SIAM J. COMPUT.
Vol. 18, No. 1, pp. 1-11, February 1989

AMORTIZED ANALYSIS OF ALGORITHMS FOR SET UNION WITH BACKTRACKING*

JEFFERY WESTBROOK† AND ROBERT E. TARJAN‡

Abstract. Mannila and Ukkonen [*Lecture Notes in Computer Science* 225, Springer-Verlag, New York, 1986, pp. 236-243] have studied a variant of the classical disjoint set union (equivalence) problem in which an extra operation, called de-union, can undo the most recently performed union operation not yet undone. They proposed a way to modify standard set union algorithms to handle de-union operations. In this paper several algorithms are analyzed based on their approach. The most efficient such algorithms have an amortized running time of $O(\log n/\log\log n)$ per operation, where n is the total number of elements in all the sets. These algorithms use $O(n \log n)$ space, but the space usage can be reduced to $O(n)$ by a simple change. The authors prove that any separable pointer-based algorithm for the problem requires $\Omega(\log n/\log\log n)$ time per operation, thus showing that our upper bound on amortized time is tight.

Key words. amortization, set union, data structures, algorithms, backtracking, logic programming

AMS(MOS) subject classifications. 68P05, 68Q25, 68R10

1. Introduction. The classical disjoint set union problem is that of maintaining a collection of disjoint sets whose union is $U=\{1, 2, \cdots, n\}$ subject to a sequence of m intermixed operations of the following two kinds:

find(x): Return the name of the set currently containing element x.

union(A, B): Combine the sets named A and B into a new set, named A.

The initial collection consists of n singleton sets, $\{1\}, \{2\}, \cdots, \{n\}$. The name of initial set $\{i\}$ is i. For simplicity in stating bounds we assume $m=\Omega(n)$. This assumption does not significantly affect any of the results, and it holds in most applications.

Several fast algorithms for this problem are known [10], [13]. They all combine a rooted tree set representation with some form of path compaction. The fastest such algorithms run in $O(\alpha(m, n))$ amortized time[1] per operation, where α is a functional inverse of Ackermann's function [10], [13]. No better bound is possible for any pointer-based algorithm that uses a separable set representation [11]. For the special case of the problem in which the subsequence of union operations is known in advance, the use of address arithmetic techniques leads to an algorithm with an amortized time bound of $O(1)$ per operation [2].

Mannila and Ukkonen [7] studied a generalization of the set union problem called *set union with backtracking*, in which the following third kind of operation is allowed:

de-union: Undo the most recently performed union operation that has not yet been undone.

This problem arises in Prolog interpreter memory management [6]. Mannila and Ukkonen showed how to extend path-compaction techniques to handle backtracking. They posed the question of determining the inherent complexity of the problem, and

* Received by the editors June 29, 1987; accepted for publication (in revised form) April 4, 1988. This research was partially supported by National Science Foundation grant DCR-8605962 and Office of Naval Research contract N00014-87-K-0467.

† Computer Science Department, Princeton University, Princeton, New Jersey 08544.

‡ Computer Science Department, Princeton University, New Jersey 08544 and AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

[1] The *amortized time* is the time of an operation averaged over a worst-case sequence of operations. See Tarjan's survey paper [12].

they claimed an $O(\log \log n)$ amortized time bound per operation for one algorithm based on their approach. Unfortunately, their upper bound argument is faulty.

In this paper we derive upper and lower bounds on the amortized efficiency of algorithms for set union with backtracking. We show that several algorithms based on the approach of Mannila and Ukkonen run in $O(\log n/\log \log n)$ amortized time per operation. These algorithms use $O(n \log n)$ space, but the space can be reduced to $O(n)$ by a simple change. We also show that any pointer-based algorithm that uses a separable set representation requires $\Omega(\log n/\log \log n)$ amortized time per operation. All the algorithms we analyze are subject to this lower bound. Improving the upper bound of $O(\log n/\log \log n)$, if it is possible, will require the use of either a non-separable pointer-based data structure or of address arithmetic techniques.

The remainder of this paper consists of four sections. In § 2 we review six algorithms for set union without backtracking and discuss how to extend them to handle backtracking. In § 3 we derive upper bounds for the amortized running times of these algorithms. In § 4 we derive a lower bound on amortized time for all separable pointer-based algorithms for the problem. Section 5 contains concluding remarks and open problems.

2. Algorithms for set union with backtracking. The known efficient algorithms for set union without backtracking [10], [13] use a collection of disjoint rooted trees to represent the sets. The elements in each set are the nodes of a tree, whose root contains the set name. Each element contains a pointer to its parent. Associated with each set name is a pointer to the root of the tree representing the set. Each initial (singleton) set is represented by a one-node tree.

To perform union(A, B), we make the tree root containing B point to the root containing A, or alternatively make the root containing A point to the root containing B and swap the names A and B between their respective elements. (This not only moves the name A to the right place but also makes undoing the union easy, as we shall see below.) The choice between these two alternatives is governed by a *union rule*. To perform find(x), we follow the path of pointers from element x to the root of the tree containing x and return the set name stored there. In addition, we apply a *compaction rule*, which modifies pointers along the path from x to the root so that they point to nodes farther along the path.

We shall consider the following possibilities for the union and compaction rules:

Union Rules:

Union by weight: Store with each tree root the number of elements in its tree. When doing a union, make the root of the smaller tree point to the root of the larger, breaking a tie arbitrarily.

Union by rank: Store with each tree root a nonnegative integer called its *rank*. The rank of each initial tree root is zero. When doing a union, make the root of smaller rank point to the root of larger rank. In the case of a tie, make either root point to the other, and increase the rank of the root of the new tree by one.

Compaction Rules (see Fig. 1):

Compression: After a find, make every element along the find path point to the tree root.

Splitting: After a find, make every element along the find path point to its grandparent, if it has one.

Halving: After a find, make every other element along the find path (the first, third, etc.) point to its grandparent, if it has one.

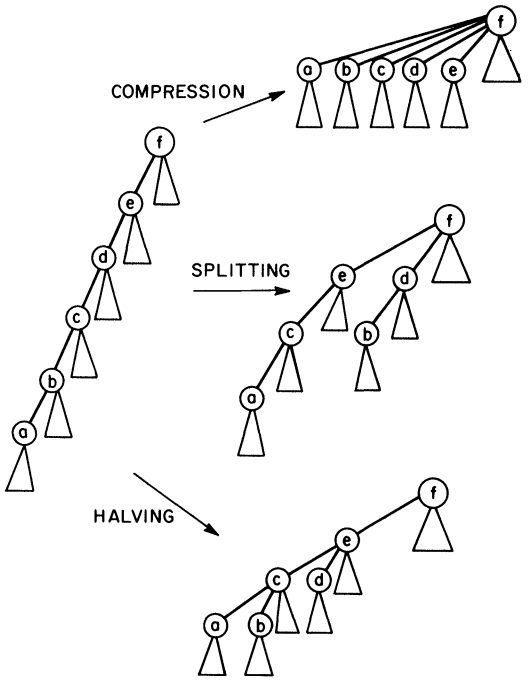


FIG. 1. *Path compression, path splitting, and path halving. The element found is "a."*

The two choices of a union rule and three choices of a compaction rule give six possible set union algorithms. Each of these has an amortized running time of $O(\alpha(m, n))$ per operation [13].

We shall describe two ways to extend these and similar algorithms to handle de-union operations. The first method is the one proposed by Mannila and Ukkonen; the second is a slight variant.

We call a union operation that has been done but not yet undone *live.* We denote a pointer from a node x to a node y by (x, y). Suppose that we perform finds without doing any compaction. Then performing de-unions is easy: to undo a set union we merely make null the pointer added to the data structure by the union. To facilitate this, we maintain a *union stack,* which contains the tree roots made nonroots by live unions. To perform a de-union, we pop the top element on the union stack and make the corresponding parent pointer null.

This method works with either of the two union rules. Some bookkeeping is needed to maintain set names and sizes or ranks. Each entry on the union stack must contain not only an element but also a bit that indicates whether the corresponding union operation swapped set names. If union by rank is used, each such entry must contain a second bit that indicates whether the union operation incremented the rank of the new tree root. The time to maintain set names and sizes or ranks is $O(1)$ per union or de-union; thus each union or de-union takes $O(1)$ time, worst-case. Either union rule guarantees a maximum tree depth of $O(\log n)$ [13]; thus the worst-case time per find is $O(\log n)$. The space needed by the data structure is $O(n)$.

Mannila and Ukkonen's goal was to reduce the time per find, possibly at the cost of increasing the time per union or de-union and increasing the space. They developed the following method for allowing compaction in the presence of de-unions. Let us call the forest maintained by the noncompacting algorithm described above the *reference*

forest. In the compacting method, each element x has an associated *pointer stack* $P(x)$, which contains the outgoing pointers that have been created during the course of the algorithm but have not yet been destroyed. The bottommost pointer on this stack is one created by a union. Such a pointer is called a *union pointer.* The other pointers on the stack are ones created by compaction. They are called *find pointers.* Each pointer (x, y) of either type is such that y is a proper ancestor of x in the reference forest.

Each pointer has an *associated union operation*, which is the one whose undoing would invalidate the pointer. To be more precise, for a pointer (x, y) the associated union operation is the one that created the pointer (z, y) such that z is a child of y and an ancestor of x in the reference forest. As a special case of this definition, if (x, y) is a union pointer, then $z = x$ and the associated union operation is the one that created (x, y). A pointer is *live* if its associated union is live.

Unions are performed as in the noncompacting method. Compactions are performed as in the set union algorithm without backtracking, except that each new pointer (x, y) is pushed onto $P(x)$ instead of replacing the old pointer, leaving x. When following a find path from an element x, the algorithm pops dead pointers from the top of $P(x)$ until $P(x)$ is empty or a live pointer is on top. In the former case, x is the root of its tree; in the latter case, the live pointer is followed.

This algorithm requires a way to determine whether a pointer is live or dead. For this purpose the algorithm assigns each union operation a distinct number as it is performed. Each entry on the union stack contains the number of the corresponding union. Each pointer on a pointer stack contains the number of the associated union and a pointer to the position on the union stack where the entry for this union was made. This information can be computed in $O(1)$ time for any pointer (x, y) when it is created. If (x, y) is a union pointer, the information is computed as part of the union. If (x, y) is a find pointer, then the last pointer on the find path from x to y when (x, y) was created has the same associated union as (x, y) and has stored with it the needed information. To test whether a pointer is live or dead, it is merely necessary to access the union stack entry whose position is recorded with the pointer and test first, if the entry is still on the stack, and second, whether its union number is the same as that stored with the pointer. If so, the pointer is live; if not, dead.

The implementation of de-union must be changed slightly, to preserve the invariant that in every pointer stack all the dead pointers are on top. To perform a de-union, the algorithm pops the top entry on the union stack. Let x be the element in this entry. The algorithm pops $P(x)$ until it contains only one pointer, which is the union pointer created by the union that is to be undone. The algorithm restores the set names and sizes or ranks as necessary, and pops the last pointer from $P(x)$. Because of the compaction, the state of the data structure after a de-union will not in general be the same as its state before the corresponding union.

We call this method the *lazy method* since it destroys dead pointers in a lazy fashion. Either of the union rules and any of the compaction rules can be used with the method. The total running time is proportional to m plus the total number of pointers created. (With any of the compaction rules, a compaction of a find path containing $k \geqq 2$ pointers results in the creation of $\Omega(k)$ pointers, $k-1$ in the case of compression or splitting and $\lfloor k/2 \rfloor$ in the case of halving.)

An alternative to the lazy method is the *eager method*, which pops pointers from pointer stacks as soon as they become dead. To make this popping possible, each union stack entry must contain a list of the pointers whose associated union is the one corresponding to the entry. When a union stack entry is popped, all the pointers on its list are popped from their respective pointer stacks as well. Each such pointer will

be on top of its stack when it is to be popped. To represent such a pointer, say (x, y), in a union stack entry, it suffices to store x. With this method, numbering the union operations is unnecessary, as is popping pointer stacks during finds.

The time required by the eager method for any sequence of operations is only a constant factor greater than that required by the lazy method, since both methods create the same pointers but the eager method destroys them earlier. With either union rule, the eager method uses $O(n \log n)$ space in the worst case, since the maximum tree depth is $O(\log n)$ and all pointers on any pointer stack point to distinct elements. (From bottom to top, the pointers on $P(x)$ point to shallower and shallower ancestors of x.)

The lazy method also has an $O(n \log n)$ space bound [3]. For any node x, consider the top pointer on $P(x)$, which is to a node, say y. Even if the pointer from x to y is currently dead, it must once have been live, and all pointers currently on $P(x)$ point to distinct nodes on the tree path from x to y as it existed when the pointer from x to y was live. Thus there can be only $O(\log n)$ such pointers. The total number of pointers therefore is $O(n \log n)$. The total number of numbers needed to distinguish relevant union operations is also $O(n \log n)$, which implies that the total space needed is $O(n \log n)$, as claimed.

The choice between the lazy and eager methods is not clear-cut. As we shall see at the end of § 3, a small change in the compaction rules reduces the space needed by either method to $O(n)$.

3. Upper bounds on amortized time. The analysis to follow applies to both the lazy method and the eager method. If we ignore the choice between lazy and eager pointer deletion, there are six versions of the algorithm, depending on the choice of a union rule and a compaction rule.

As a first step on the analysis, we note that compression with either union rule is no better in the amortized sense than doing no compaction at all, i.e., the amortized time per operation is $\Omega(\log n)$. The following class of examples shows this. For any k, form a tree of 2^k elements by doing unions on pairs of elements, then on pairs of pairs, and so on. This produces a tree called a *binomial tree* B_k, whose depth is k. (See Fig. 2.) Repeat the following three operations any number of times: do a find on the deepest element in B_k, undo the most recent union, and redo the union. Each find creates $k-1$ pointers, which are all immediately made dead by the subsequent de-union. Thus the amortized time per operation is $\Omega(k) = \Omega(\log n)$.

Both splitting and halving perform better; each has an $O(\log n / \log \log n)$ amortized bound per operation, in combination with either union rule. To prove this, we need a definition. For an element x, let size(x) be the number of descendants of x (including itself) in the reference forest. The *logarithmic size* of x, lgs(x), is $\lfloor \lg \text{size}(x) \rfloor$.[2]

We need the following lemma concerning logarithmic sizes when union by weight is used.

LEMMA 1 [10]. *Suppose union by weight is used. If node v is the parent of node w in the reference forest, then* $\text{lgs}(w) < \text{lgs}(v)$. *Any node has logarithmic size between* 0 *and* $\lg n$ *(inclusive).*

Proof. When a node v becomes the parent of another node w, $\text{size}(w) \leqq 2\,\text{size}(v)$ by the union by weight rule. Later unions can only increase size(v) and cannot increase size(w) (unless the union linking v and w is undone). The lemma follows. □

[2] For any x, $\lg x = \log_2 x$.

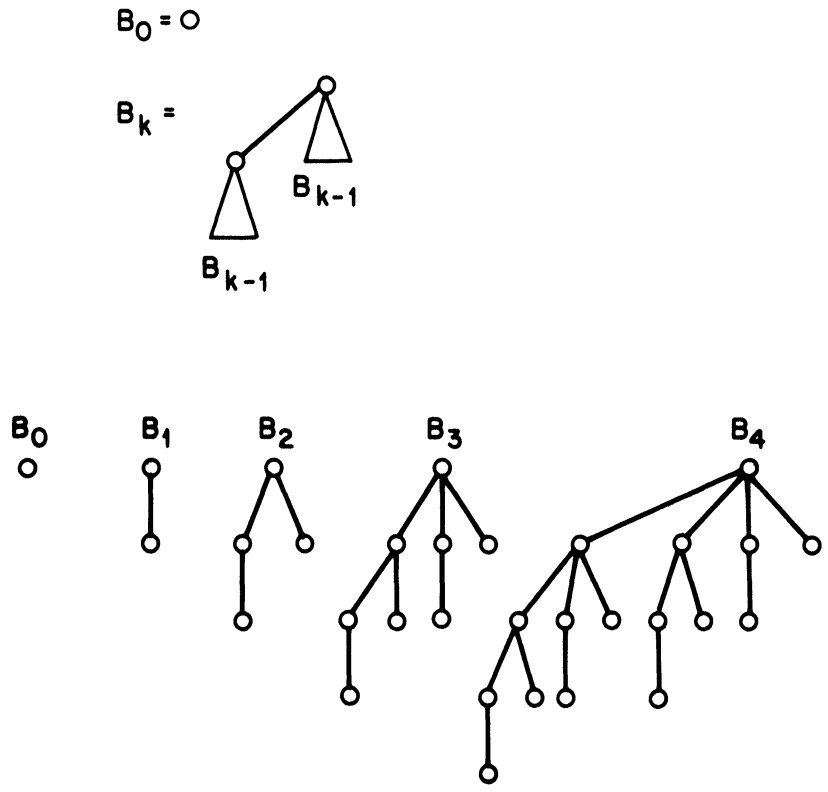


FIG. 2. *Binomial trees.*

THEOREM 1. *Union by weight in combination with either splitting or halving gives an algorithm for set union with backtracking running in* $O(\log n/\log\log n)$ *amortized time per operation.*

Proof. We shall charge the pointer creations during the algorithm to unions and finds in such a way that each operation is charged for $O(\log n/\log\log n)$ pointer creations. For an arbitrary positive constant $c<1$, we call a pointer (x, y) *short* if $\lg s(y)-\lg s(x)\leqq c\lg\lg n$ and *long* otherwise. (The logarithmic sizes in this definition are measured at the time (x, y) is created.) We charge the creation of a pointer (x, y) as follows:

(i) If y is a tree root, charge the operation (union or find) that created (x, y).
(ii) If y is not a tree root and (x, y) is long, charge the find that created (x, y).
(iii) If y is not a tree root and (x, y) is short, charge the union that most recently made y a nonroot.

A find with splitting creates two new paths of pointers, and a find with halving creates one new path of pointers. Thus $O(1)$ pointers are charged to each operation by (i). The number of long pointers along any path can be estimated as follows. For any long pointer (x, y), $\lg s(y)-\lg s(x)>c\lg\lg n$. Logarithmic sizes strictly increase along any path and are between 0 and $\lg n$ by Lemma 1. Thus if there are k long pointers on a path, $\lg n\geqq kc\lg\lg n$, which implies $k\leqq\lg n/(c\lg\lg n)$. Thus a find with either splitting or halving can create only $O(\log n/\log\log n)$ long pointers, which means that $O(\log n/\log\log n)$ pointers are charged to each find by (ii).

It remains for us to bound the number of pointers charged by (iii). Consider a union operation that makes an element x a child of another element y. Let I be the

time interval during which pointers are charged by (iii) to this union. During I, the sizes, and hence the logarithmic sizes, of all descendants of x remain constant. Interval I ends with the undoing of the union.

For each descendant w of x, at most one pointer (w, x) can be charged by (iii) to the union, since the creation of another such pointer charged by (iii) cannot occur at least until x again becomes a root and then becomes a nonroot, which can only happen after the end of I. Thus the number of pointers charged by (iii) to the union is at most one per descendant w of x such that $\text{lgs}(x) - \text{lgs}(w) \leqq c \lg \lg n$.

Since logarithmic sizes strictly increase along tree paths, any two elements u and v with $\text{lgs}(u) = \text{lgs}(v)$ must be unrelated, i.e., their sets of descendants are disjoint. This means that the number of descendants w of x with $\text{lgs}(w) = i$ is at most $\text{size}(x)/2^i \leqq 2^{\text{lgs}(x)+1-i}$, and the number of descendants w of x with $\text{lgs}(x) - \text{lgs}(w) \leqq c \lg \lg n$ is at most

$$\sum_{i=\text{lgs}(x)-\lfloor c \lg \lg n \rfloor}^{\text{lgs}(x)} 2^{\text{lgs}(x)+1-i} \leqq 2^{\lfloor c \lg \lg n \rfloor + 2} = O((\log n)^c) = O(\log n / \log \log n),$$

since $c < 1$. Thus there are $O(\log n / \log \log n)$ pointers charged to the union by (iii). □

The same result holds if union by rank is used instead of union by weight, but in this case the proof becomes a little more complicated because logarithmic sizes need not strictly increase along tree paths. We deal with this by slightly changing the definition of short and long pointers. We need the following lemma.

LEMMA 2 [13]. *Suppose union by rank is used. If node v is the parent of node w in the reference forest, then $0 \leqq \text{lgs}(w) \leqq \text{lgs}(v) \leqq \lg n$ and $0 \leqq \text{rank}(w) < \text{rank}(v) \leqq \lg n$.*

Proof. The first group of inequalities is immediate. The definition of union by rank implies $\text{rank}(w) < \text{rank}(v)$. A proof by induction on the rank of v shows that $\text{size}(v) \geqq 2^{\text{rank}(v)}$, which implies that $\text{rank}(v) \leqq \lg n$. □

THEOREM 2. *Union by rank in combination with either splitting or halving gives an algorithm for set union with backtracking running in $O(\log n / \log \log n)$ amortized time per operation.*

Proof. We define a pointer (x, y) to be *short* if $\max\{\text{lgs}(y) - \text{lgs}(x), \text{rank}(y) - \text{rank}(x)\} \leqq c \lg \lg n$ and *long* otherwise, where $c < 1$ is a positive constant. We charge the creation of pointers to unions and finds exactly as in the proof of Theorem 1 (rules (i), (ii), and (iii)). The number of pointers charged by rule (i) is $O(1)$ per union or find, exactly as in the proof of Theorem 1. A long pointer (x, y) satisfies at least one of the inequalities $\text{lgs}(x) - \text{lgs}(x) > c \lg \lg n$ and $\text{rank}(y) - \text{rank}(x) > c \lg \lg n$. Along any tree path only $O(\log n / \log \log n)$ long pointers can satisfy the former inequality and only $O(\log n / \log \log n)$ long pointers can satisfy the latter, by Lemma 2. It follows that only $O(\log n / \log \log n)$ pointers can be charged per find by rule (ii).

To count short pointers, we have one additional definition. For a nonroot element x, let $p(x)$ be the parent of x in the reference forest. A nonroot x is *good* if $\text{lgs}(x) < \text{lgs}(p(x))$ and *bad* otherwise, i.e., if $\text{lgs}(x) = \text{lgs}(p(x))$. The definition of lgs implies that any element can have at most one bad child. The bad elements thus form paths called *bad paths* of length $O(\log n)$; all elements on a bad path have the same logarithmic size. We call the element of largest rank on a bad path the *head* of the path. The head of a bad path is a bad element whose parent is either a good element or a tree root.

Consider a union operation that makes an element x a child of an element y. We count short pointers charged to this union as follows:

(1) *Short pointers leading from good elements.* If v and w are good elements such that $\text{lgs}(v) = \text{lgs}(w)$, then v and w are unrelated in the reference forest, i.e., they have

disjoint sets of descendants. The analysis that yielded the count of short pointers in the proof of Theorem 1 applies to the good elements here to yield a bound of $O((\log n)^c) = O(\log n/\log\log n)$ short pointers leading from good elements that are charged to the union by (iii).

(2) *Short pointers leading from bad elements.* Consider the number of bad paths from which short pointers can lead to x. The head of such a path is an element w such that $p(w)$ is either good or a tree root, and $\lg s(x) - \lg s(w) \leqq c \lg\lg n$. Heads of different bad paths have different parents. The analysis that counts short pointers in the proof of Theorem 1 yields an $O((\log n)^c)$ bound on the number of bad paths from which short pointers can lead to x. Along such a bad path, rank strictly increases, and the definition of shortness implies that only the $c \lg\lg n$ elements of largest rank along the path can have short pointers leading to x. The total number of short pointers leading from bad nodes that are charged to the union by (iii) is thus $O(c \log\log n\ (\log n)^c) = O(\log n/\log\log n)$. □

We conclude this section by discussing how to reduce the space bound for both the lazy method and the eager method to $O(n)$. This is accomplished by making the following simple changes in the compaction rules. If union by size is used, the compaction of a find path is begun at the first node along the path whose size is at least $\lg n$. If union by rank is used, the compaction of a find path is begun at the first node whose rank is at least $\lg\lg n$. With this modification, only $O(n/\log n)$ nodes have find pointers leaving them, and the total number of pointers in the data structure at any time is $O(n)$. The analysis in Theorems 1 and 2 remains valid, except that there is an additional time per find of $O(\log\log n)$ to account for the initial, noncompacted part of each find path.

4. A general lower bound on amortized time. We shall prove that the bound in Theorems 1 and 2 is best possible for a large class of algorithms for set union with backtracking. Our computational model is the *pointer machine* [4], [5], [9], [11] with an added assumption about the data structure called *separability.* Related results follow. Tarjan [11] derived an amortized bound in this model for the set union problem without backtracking. Blum [1] derived a worst-case-per-operation lower bound for the same problem. Mehlhorn, Näher, and Alt [8] derived an amortized lower bound for a related problem. Their result does not require separability.

The algorithms to which our lower bound applies are called *separable pointer algorithms.* Such an algorithm uses a linked data structure that can be regarded as a directed graph, with each pointer represented by an edge. The algorithm solves the set union with backtracking problem according to the following rules:

(i) The operations are presented on-line, i.e., each operation must be completed before the next one is known.

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

(iii) (Separability.) After any operation, the data structure can be partitioned into node-disjoint subgraphs, one corresponding to each currently existing set and containing all the elements in the set. The name of the set occurs in exactly one node in the subgraph. *No edge leads from one subgraph to another.*

(iv) The cost of an operation find(x) is the length (number of edges) of the shortest path from x to the node that holds the name of the set containing x. This length is measured at the beginning of the find, i.e., before the algorithm changes the structure as specified in (v).

(v) During any find, union, or de-union operation, the algorithm can add edges to the data structure at a cost of one per edge, delete edges at a cost of zero, and move,

add, or delete set names at a cost of zero. The only restriction is that separability must hold after each operation.

The eager method of § 2 obeys rules (i)-(v). This is also true of the lazy method, if we regard pointers as disappearing from the model data structure as soon as they become dead. This does not affect the performance of the algorithm in the model, since once a pointer becomes dead it is never followed.

THEOREM 3. *For any n, any $m = \Omega(n)$, and any separable pointer algorithm, there is a sequence of m find, union, and de-union operations whose cost is $\Omega(m \log n/\log\log n)$.*

Proof. We shall prove the theorem for n of the form 2^{2^k} for some $k \geqq 1$ and for $m \geqq 4n$. The result follows for all n and $m = \Omega(n)$ by padding the expensive problem instances constructed below with extra singleton sets on which no operations take place and with extra finds.

In estimating the cost of a sequence of operations, we shall charge the cost of adding an edge to the data structure to the deletion of the edge. Since this postpones the cost, it cannot increase the total cost of a sequence.

We construct an expensive sequence as follows. The first $n-1$ operations are unions that build a set of size n by combining singletons in pairs, pairs in pairs, and so on. The remaining operations occur in groups, each group containing between 1 and $2n-2$ operations. Each group begins and ends with all the elements in one set. We obtain a group of operations by applying the appropriate one of the following two cases (if both apply, either may be selected). Let $b = \lfloor \lg n/(2 \lg\lg n) \rfloor$.

(1) If some element in the (only) set is at distance at least b away from the set name, do a find on this element.

(2) If some sequence of ℓ de-unions will force the deletion of ℓb edges from the data structure (to maintain separability), do these de-unions. Then do the corresponding unions in the reverse order, restoring the initial set of size n.

We claim that if there is only one set, formed by repeated pairing, then case (1) or case (2) must apply. If this is true, we can obtain an expensive sequence of operations by generating successive groups of operations until more than $m-2n+2$ operations have occurred, and then padding the sequence with enough additional finds to make a total of m operations. The cost of such a sequence is at least $(m-3n+3)b = \Omega(m \log n/\log\log n)$.

It remains to prove the claim. Suppose case (2) does not apply. We shall show that case (1) does. Let $f = (\lg n)^2$. For $0 \leqq i \leqq \lg n/\lg f$ we define a partition P_i of the nodes of the data structure as follows:

$$P_i = \{X \mid X \text{ is the collection of nodes in the subgraph corresponding to one of the sets that would be formed by doing } f^i - 1 \text{ de-unions}\}.$$

Observe that $|P_i| = f^i$. Also $f^{\lg n/\lg f} = n$, so P_i is defined for $i \leqq \lg n/\lg f$. In particular P_b is defined, since $b = \lfloor \lg n/(2 \lg\lg n) \rfloor = \lfloor \lg n/\lg f \rfloor$.

For $0 \leqq i \leqq \lg n/\lg f$, we define the collection D_i of *deep sets* in P_i as follows:

$$D_i = \{X \in P_i \mid \text{all elements in } X \text{ are at distance at least } i \text{ from the name of the single set}\}.$$

Let $d_i = |D_i|$. We shall show that $d_b > 0$, which implies the existence of an element at distance at least b away from the name of the single set; hence case (1) applies.

Let ℓ_i be the number of edges leading from one set in P_i to another. We have $\ell_i \leqq bf^i$, since otherwise performance of $f^i - 1$ de-unions would force the deletion of bf^i edges from the data structure, and case (2) would apply.

Now we derive a recursive bound on d_i. We have $d_1 = f - 1$, since only one of the f sets in P_1 can contain the only set name. We claim that $d_{i+1} \geqq fd_i - \ell_i$. To verify the claim, let us consider D_i. Since $n = 2^{2^k}$ and the union structure of the only set forms

a binomial tree, each set X in D_i consists of f sets in P_{i+1}, all of whose elements are at distance at least i from the name of the only set. For an element $x \in X$ to be at distance exactly i from the set name, some edge must lead from x to a set in P_i other than X; otherwise X would not be in D_i. There are ℓ_i such edges. Each such edge can eliminate one set in P_{i+1} from being in D_{i+1}. But this leaves $fd_i - \ell_i$ sets in D_{i+1}, namely the fd_i sets into which the sets in D_i divide, minus at most ℓ_i eliminated by edges between different sets in P_i. That is, $d_{i+1} \geqq fd_i - \ell_i$, as claimed.

Applying the bound $\ell_i \leqq bf^i$ gives $d_{i+1} \geqq fd_i - bf^i$. Using $d_1 = f - 1$, a proof by induction shows that $d_i \geqq f^{i-1}(f-(i-1)b-1)$.

We wish to show that $d_b > 0$. This is true provided that $(f-(b-1)b-1) > 0$. But $f = (\lg n)^2$ and $b = \lfloor \lg n/(2\lg\lg n) \rfloor$, giving $(f-(b-1)b-1) = (f-b^2+b-1) \geqq \frac{3}{4}(\lg n)^2 > 0$, since we are assuming $n \geqq 4$, which implies $b^2 \leqq (\lg n)^2/4$ and $b \geqq 1$. Thus $d_b > 0$, which implies that some element is at distance at least b from the set name, i.e., case (1) applies. □

5. Remarks. Our bound of $\Theta(\log n/\log\log n)$ on the amortized time per operation in the set union problem with backtracking is the same as Blum's worst-case bound per operation in the set union problem without backtracking [1]. Perhaps this is not a coincidence. Our lower bound proof resembles his. Furthermore the data structure he uses to establish his upper bound can easily be extended to handle de-union operations; the worst-case bound per operation remains $O(\log n/\log\log n)$ and the space needed is $O(n)$.

The compaction methods have the advantage over Blum's method that as the ratio of finds to unions and de-unions increases, the amortized time per find decreases. The precise result is that if the ratio of finds to unions and de-unions in the operation sequence is γ and the amortized time per union and de-union is defined to be $\Theta(1)$, then the amortized time per find is $\Theta(\log n/(\max\{1, \log(\gamma \log n)\}))$. This bound is valid for any value of γ, and it holds for splitting or halving with either union rule, and it is the best bound possible for any separable pointer algorithm. This can be proved using straightforward extensions of the arguments in §§ 3 and 4. The space bound can be made $O(n)$ by an extension of the idea proposed at the end of § 3. If the de-union operations occur in bursts, the time per operation decreases further, but we have not attempted to analyze this situation.

Perhaps the most interesting open problem is whether the lower bound in § 4 can be extended to nonseparable pointer algorithms. (In place of separability, we require that the out-degree of every node in the data structure be constant.) We conjecture that the bound in Theorem 3 holds for such algorithms. The techniques of Mehlhorn, Näher, and Alt [8] suggest an approach to this question, which might yield at least an $\Omega(\log\log n)$ bound if not an $\Omega(\log n/\log\log n)$ bound on the amortized time.

REFERENCES

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

[2] 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.

[3] G. GAMBIOSI, G. F. ITALIANO, AND M. TALAMO, *Getting back to the past in the union-find problem*, in 5th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 294, Springer-Verlag, Berlin, 1988, pp. 8–17.

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

[5] A. N. KOLMOGOROV, *On the notion of algorithm*, Uspekhi Mat. Nauk, 8 (1953), pp. 175–176.

[6] H. MANNILA AND E. UKKONEN, *On the complexity of unification sequences*, in 3rd International Conference on Logic Programming, July 14–18, 1986, Lecture Notes in Computer Science 225, Springer-Verlag, New York, 1986, pp. 122–133.

[7] ———, *The set union problem with backtracking*, in Proc. 13th International Colloquium on Automata, Languages, and Programming (ICALP 86), Rennes, France, July 15–19, 1986, Lecture Notes in Computer Science 226, Springer-Verlag, New York, 1986, pp. 236–243.

[8] K. MEHLHORN, S. NÄHER, AND H. ALT, *A lower bound for the complexity of the union-split-find problem*, in Proc. 14th International Colloquium on Automata, Languages, and Programming (ICALP 87), Karlsruhe, Federal Republic of Germany, July 13–17, 1987, Lecture Notes in Computer Science 267, Springer-Verlag, New York, 1987, pp. 479–488.

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

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

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

[12] ———, *Amortized computational complexity*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 306–318.

[13] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 12-29, February 1989

002

THE BIT COMPLEXITY OF RANDOMIZED LEADER ELECTION ON A RING*

KARL ABRAHAMSON†, ANDREW ADLER‡, RACHEL GELBART†, LISA HIGHAM†, AND DAVID KIRKPATRICK†

Abstract. The inherent bit complexity of leader election on asynchronous unidirectional rings of processors is examined under various assumptions about global knowledge of the ring. If processors have unique identities with a maximum of m bits, then the expected number of communication bits sufficient to elect a leader with probability 1, on a ring of (unknown) size n is $O(nm)$. If the ring size is known to within a multiple of 2, then the expected number of communication bits sufficient to elect a leader with probability 1 is $O(n \log n)$.

These upper bounds are complemented by lower bounds on the communication complexity of a related problem called solitude verification that reduces to leader election in $O(n)$ bits. If processors have unique identities chosen from a sufficiently large universe of size s, then the average, overall choices of identities, of the communication complexity of verifying solitude is $\Omega(n \log s)$ bits. When the ring size is known only approximately, then $\Omega(n \log n)$ bits are required for solitude verification. The lower bounds address the complexity of certifying solitude. This is modelled by the best-case behaviour of nondeterministic solitude-verification algorithms.

Key words. bit complexity, leader election, asynchronous distributed computation, randomized algorithms, processor rings, communication complexity, attrition, solitude verification, lower bounds

AMS(MOS) subject classifications. 68Q10, 68Q25

1. Introduction.

Studies in the complexity of distributed computation typically ask two questions. (i) What is the complexity, under some chosen measure(s), of a chosen problem or family of problems, on distributed networks formalized in a chosen model? (The model might include both the network topology and assumptions about the processors and their interprocessor communication.) (ii) How is this complexity affected by changes in the model? This paper addresses these questions for the problem of electing a leader using randomized distributed algorithms running on asynchronous unidirectional rings of processors, where the measure of complexity is the expected number of bits transmitted.

Leader-election results in a unique processor, from among a specified subset of the processors, entering a distinguished final state. This problem is one of a small number of problems which are fundamental in that their solutions form the building blocks of many more involved distributed computations. Earlier work in the study of distributed computation has established the importance of the ring topology as a test-bed for the design and analysis of distributed algorithms. It is the chosen model here because it is a simple topology which exhibits many important attributes of distributed computations.

A unidirectional ring can be viewed as a sequence $P_1, \cdots, P_n$ of processors where each processor P_i sends messages to P_{i+1} and receives messages from P_{i-1} (subscripts are implicitly reduced in the obvious way). A number of variants of this basic model are distinguished by supplementary properties. Communication between processors is either synchronous or asynchronous. Processors may be indistinguishable or may have

* Received by the editors February 24, 1986; accepted for publication (in revised form) January 4, 1988.
† Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada.
‡ Department of Mathematics, University of British Columbia, Vancouver, British Columbia, Canada.

distinct identifiers. Processors may or may not know the size of the ring at the start of computation. The complexity, measured by the number of communication messages, of leader election on models with various combinations of these properties has been well studied.

On an asynchronous unidirectional ring of processors with distinct identifiers, a leader can be elected by a deterministic algorithm which operates using pairwise comparisons of processor identifiers, using $O(n \log n)$ messages each of $O(m)$ bits [DKR], [Pe], where m is the number of bits in the largest processor identifier. If interprocessor communication is synchronous and identifiers are drawn from some known countable universe, then $O(n)$ messages suffice to elect a leader in the worst case [FL]. On the other hand, in the asynchronous case, if the universe of identifiers is unbounded, any deterministic leader-election algorithm must exchange $\Omega(n \log n)$ messages (of arbitrary length) in the worst case [B], [PKR] or average case [PKR], even if bidirectional communication is possible. Even if the ring size n is known to all processors and messages are transmitted synchronously, algorithms which are restricted to operate either on the basis of comparisons of processor identifiers or within a bounded number of rounds, must transmit $\Omega(n \log n)$ messages in the worst case [FL].

If processors are not endowed with distinct identifiers then, as was first observed by Angluin [A], deterministic algorithms are unable to elect leaders, even if n is known to all processors. Itai and Rodeh [IR] propose the use of randomized algorithms to skirt this limitation. They present a randomized algorithm that elects a leader in an asynchronous ring of known size n using $O(n \log n)$ expected messages of $O(\log n)$ bits each. The lower bound results of [Pa] show that even if processors have distinct identifiers drawn from some sufficiently large universe, the expected number of messages (of arbitrary length) communicated by a randomized leader election algorithm is $\Omega(n \log n)$. However, $O(n)$ expected messages suffice for randomized leader election on a synchronous ring without identifiers [IR], provided the ring size n is known to all processors.

In this paper, it is shown that with respect to bit complexity, the algorithms cited above for asynchronous rings are not optimal. The relationship between leader election and two subproblems, called attrition and solitude verification, is explored in § 2. Efficient reductions are established which motivate the development of procedures for these two subproblems (§ 3) and lower bounds for solitude verification (§ 4).

It follows from the results of § 3 that when each processor in a unidirectional ring of n processors has a distinct m-bit identifier, it is possible to elect a leader by a randomized algorithm using $O(mn)$ expected bits of communication. In addition, when the processors are indistinguishable but each knows the ring size n to within a factor of 2, then it is possible to elect a leader by a randomized algorithm using $O(n \log n)$ expected bits of communication.

These upper bounds are complemented by the lower bounds of § 4. It follows from the results of that section that when processors have unique identifiers drawn from a sufficiently large universe of size s then any algorithm must transmit $\Omega(n \log s)$ bits to elect a leader, even in the best case. If processors are indistinguishable but each knows the ring size only to within some interval of size Δ, then any algorithm must transmit $\Omega(n \log \Delta)$ bits to elect a leader, even in the best case.

The lower bounds are proved for successful computations of algorithms in a very general model. Algorithms may be nondeterministic and nonuniform and may deadlock. Moreover computations need only terminate in the weak nondistributive sense. An algorithm terminates distributively if whenever processors enter a decision state the

decision is irrevocable, that is, it will not change in light of subsequent messages received. The weaker nondistributive termination refers to the situation in which the cessation of message traffic is not necessarily detectable by individual processors and all decisions are predicated on this undetectable condition.

The upper and lower bounds are tight to within constant factors, except when the ring size is known to lie within some relatively small interval. This gap is reminiscent of earlier results concerning knowledge of ring size [FL]. The special case when the ring size is known exactly is the subject of a companion paper [AAHK2]. Another direction for investigation is uncertain leader election, that is, probabilistic leader election that terminates correctly with probability greater than $1-\varepsilon$ for some fixed $\varepsilon > 0$. A further companion paper [AAHK1] considers this problem on rings of indistinguishable processors. Pachl [Pa] studies the problem when processors have distinct identifiers. Some of these results together with an overview of the results of the present paper are briefly described in § 5.

2. Leader election, attrition, and solitude verification. This section sets out a general framework for the study of leader election on rings. Two fundamental problems are introduced and their relationship to leader election is established. This relationship motivates the algorithms and lower bound results of the next two sections.

Leader election requires that a single processor be chosen from among some nonempty subset of processors called *candidates*. Initially each candidate is a *contender*. Intuitively, a leader election algorithm must (i) eliminate all but one contender by converting some of the contenders to *noncontenders*, and (ii) confirm that only one contender remains. This separation was pointed out and exploited earlier by Itai and Rodeh [IR]. These subtasks are called attrition and solitude verification, respectively. More formally, a procedure solves the *attrition problem* if, when initiated by every candidate, it eventually takes all but exactly one of these candidates into a permanent state of noncontention. Typically an attrition procedure does not terminate but rather enters an infinite loop in which the remaining contender continues to send messages to itself. An algorithm solves the *solitude-verification problem* if, when initiated by a set of processors, it eventually terminates with an initiator in state "yes" if and only if it was the sole initiator. The more stringent *solitude-detection problem* requires, in addition, that all initiators be left in state "no" if there was more than one initiator.

Both attrition and solitude detection can be reduced to leader election with $O(n)$ bits of communication on rings of size n. For the attrition reduction, a leader is elected from among the attrition contenders. For the solitude-verification reduction, the processors wishing to detect their solitude first use $O(n)$ bits to alert the whole ring to contend for leadership.[1] Once a leader is elected, it is easy to see how to solve attrition (no additional communication) or to detect solitude (an additional $O(n)$ bits of communication). Hence, nonlinear lower bounds on the complexity of either attrition or solitude verification translate to lower bounds on the complexity of leader election. Conversely, attrition and solitude verification can be interleaved to solve leader election by annotating attrition messages with solitude-verification messages. Whenever a contender enters a state of noncontention, it forwards a solitude-verification restart message to alert remaining contenders that they were not previously alone. When attrition has reduced the set of contenders to one, solitude verification will proceed uninterrupted,

[1] The reduction outlined here is from solitude verification to a version of leader election in which all processors are candidates for leadership. In fact, this can be generalized to a reduction to an arbitrary set of candidates as is shown in § 4.2.

eventually verifying that only one contender, the "leader," remains. If the solitude-verification algorithm terminates distributively, so does the resulting leader-election algorithm.

The efficiency of the leader-election algorithm described above depends not only on the efficiency of the attrition and solitude verification procedures from which it is constructed, but also on the cost of interleaving. If solitude verification is *conservative* in the sense that every message is bounded in length by some fixed constant number of bits, then annotated attrition messages are at worst a constant factor longer than unannotated messages. So the cost of premature attempts to verify solitude is dominated by the cost of attrition. The only remaining cost attributable to interleaving involves the transmission of restart messages. In general this cost can also be subsumed by the cost of attrition. This is shown in the next section for the attrition procedure used in this paper. That section describes a randomized attrition procedure, two different conservative solitude-verification algorithms (each exploiting different possible properties of the ring of processors), and an interleaving strategy that combines the procedures to yield efficient leader-election algorithms.

3. Procedures for attrition and solitude verification. The previous section argues that leader election can be efficiently reduced to attrition and solitude verification. This section describes and analyses efficient procedures for these two subproblems. The attrition procedure is randomized but completely general in that it makes no assumptions about the host ring. Two solitude-verification algorithms are described which exploit different assumptions about the ring, specifically the existence of distinct identifiers or at least partial knowledge of the ring size. Both solitude-verification algorithms are deterministic and terminate distributively.

3.1. The attrition procedure.

3.1.1. Informal description. The attrition procedure is initiated by all candidates for leadership. The number of candidates is denoted by c. The candidates are the initial contenders; all other processors are noncontenders. The procedure uses coin tosses to eliminate some contenders while ensuring that it is not possible for all contenders to be eliminated. A noncontender never converts to a contender, but behaves entirely passively—simply forwarding any messages received. Contenders create messages which are propagated to the next contender. Since contenders (respectively, noncontenders) have active (respectively, passive) roles in the algorithm, they are referred to as *active* (*passive*) processors in the following description.

Like the randomized attrition procedure of [IR], the procedure here can be thought of as proceeding in *phases*. However, these phases are implicit only. They are not enforced by counters, but will be justified in the subsequent analysis of the procedure.

At the beginning of each phase, each active processor tosses an unbiased coin yielding h or t, sends the outcome to its successor, and waits to receive a message from its predecessor. Suppose an active processor P sends and receives the same coin toss. It is possible that this is true for every active processor, so no decision can be taken by P. P continues alternately to send and receive coin tosses, remaining in the same phase, until P receives a message different from what it most recently sent. Suppose P eventually sends t and receives h. Then some other active processor Q must have sent h and received t. If only one of P and Q becomes passive, the possibility of losing all active processors is avoided. The convention used is that sending t and receiving h changes an active processor to a passive processor in the next phase, while the opposite results in a processor remaining active in the next phase. Once any active processor has decided its stat. (active or passive) for the next phase, it sends a *

message to signal the end of its phase. Any undecided processor Q that receives a * message while waiting for a coin flip becomes passive in the next phase, since it is assured that there will remain at least one active processor. Q forwards the * message to announce that it has ended its phase. * messages continue to be forwarded until received by a decided processor (which has already propagated a * message).

These ideas are formalized in the transition diagram of Fig. 1. The notation "x/y" is read "receive x, then send y." λ is used where no reception or transmission is to occur. Each processor begins at START. The transition leaving state FLIP is chosen by an unbiased coin toss.

3.1.2. Formal description and correctness. This section establishes the following properties of the attrition algorithm of Fig. 1:

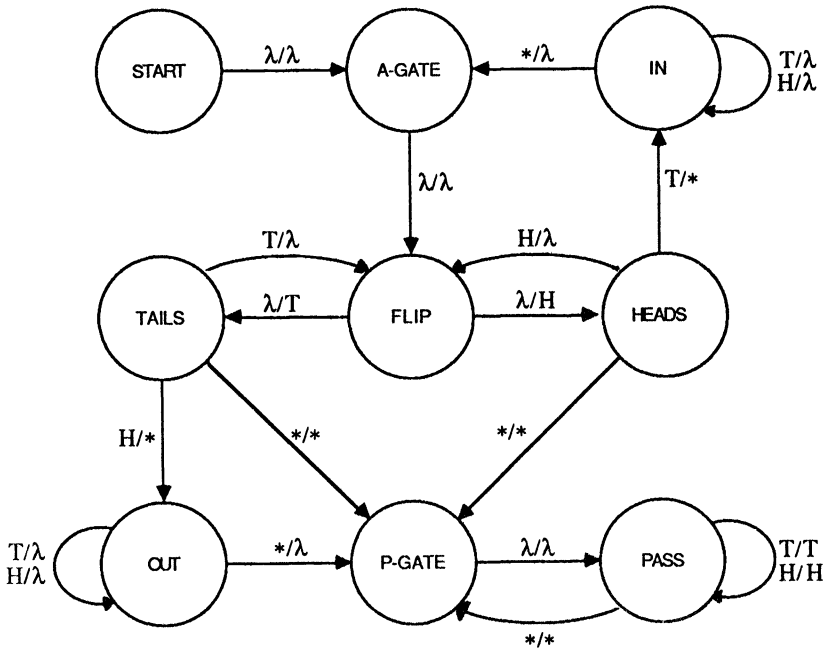


FIG. 1. *The attrition procedure.*

(1) There is always at least one active processor. (A processor is active if it is in any state other than *P*-GATE or PASS.)
(2) The attrition procedure cannot deadlock.
(3) The number of active processors never increases and eventually decreases to one.

Some definitions are introduced to facilitate the proofs. Let each processor P_x maintain an internal *phase counter*, ρ_x. ρ_x is initialized to 0 and incremented each time P_x enters a gate state. When $\rho_x = k$, P_x is in *phase k*. The following variables are defined relative to an arbitrary computation of the attrition process:

$s_{x,j}^{(k)}$ = the jth message sent by P_x while $\rho_x = k$.
$r_{x,j}^{(k)}$ = the jth message received by P_x while $\rho_x = k$.
$q_{x,j}^{(k)}$ = the state of P_x immediately after P_x sends its jth message of phase k.
$g_{x,j}^{(k)}$ = the state of P_x immediately before P_x receives its jth message of phase k.

If P_x does not receive (send) j messages in phase k, then $r_{x,j}^{(k)}$ and $g_{x,j}^{(k)}$ ($s_{x,j}^{(k)}$ and $q_{x,j}^{(k)}$) are undefined. Note that the state variables are parameterized by messages sent and received, not by transitions made. For example, if $q_{x,j}^{(k)} = \text{IN}$, then $q_{x,j-1}^{(k)} = \text{HEADS}$.

The following lemma establishes that * messages effectively delimit phase boundaries.

LEMMA 3.1. $s_{x,j}^{(k)} = r_{x+1,j}^{(k)}$.

Proof. Since messages sent by P_x are exactly those received by P_{x+1}, it suffices to show that P_x and P_{x+1} agree on phase boundaries. Notice from Fig. 1 that P_{x+1} enters a gate (a phase boundary) precisely when it receives a * message. But P_x, having sent the * message, cannot send any further message without first passing through a gate. Nor can P_x enter a gate without sending a * message. □

The following lemma is immediate from Fig. 1. It points out that within a phase, as long as a processor is undecided, its communication alternates between output and input messages.

LEMMA 3.2. *If P_x is active in phase k and $q_{x,j}^{(k)}$ is defined, then $q_{x,j}^{(k)} = g_{x,j}^{(k)}$.*

It follows from Lemma 3.1 that each phase can be considered in isolation. Consider an arbitrary phase k with $m > 1$ active processors. Since passive processors merely forward messages, they can be ignored for the following lemmas, and it can be assumed that the ring has only m processors, $P_0, \cdots, P_{m-1}$. In the remainder of this section, the superscript (k) is omitted, and variables are assumed to describe the kth phase.

The next lemma establishes that the processors cannot all become passive.

LEMMA 3.3. *If $q_{x,j} = \text{OUT}$, then there exists w and i such that $q_{w,i} = \text{IN}$.*

Proof. Choose the smallest i such that $s_{v,i} = *$, over all processors P_v. Let P_y be some processor for which i is minimized. Now either $q_{y,i} = \text{IN}$ or $q_{y,i} = \text{OUT}$. In the first case nothing remains to be proved. In the second case, it follows that $q_{y,i-1} = \text{TAILS}$, $s_{y,i-1} = t$, and $r_{y,i-1} = h$. Let C be the class of t-messages which were the $(i-1)$th messages sent by some processor. P_y sends a message of class C, but does not receive one, since $r_{y,i-1} = s_{y-1,i-1} = h$. Since there can be at most n messages in class C, and all are eventually delivered, some processor P_w must receive a class C message without sending one. The transition from OUT to OUT cannot correspond to the reception of a class C message, by the minimality of i. The transition from TAILS to FLIP pairs the absorption of one message in C with the production of one on the previous transition. The only remaining transitions compatible with receiving a t-message lead to state IN. Therefore P_w must be in state IN after receiving its ith message, and hence $q_{w,i} = \text{IN}$. □

The safety properties of the attrition procedure follow from Lemma 3.3 and the next lemma. At any point during the execution, let $N(*)$ be the number of * messages awaiting delivery. Let $N(\text{OUT})$ and $N(\text{IN})$ be the number of processors in states OUT and IN, respectively.

LEMMA 3.4. $N(*) = N(\text{IN}) + N(\text{OUT})$ *at every point during the execution.*

Proof. The equation holds initially and is preserved by every transition. □

COROLLARY 3.5. *If any processor reaches phase $k+1$, then some processor is active in phase $k+1$.*

Proof. In order for any processor to reach a gate, some processor must reach IN or OUT. By Lemma 3.3, some processor P_w reaches IN. As long as P_w remains at IN, there is an undelivered * message which must move around the ring, eventually reaching P_w and causing P_w to enter the active gate. □

COROLLARY 3.6. *The attrition procedure cannot deadlock.*

The next lemma leads to a bound on the number of phases in any computation of the attrition procedure.

LEMMA 3.7. *Suppose $q_{x,j} = \text{IN}$. Let y be the first number in the list $x-1, x-2, \cdots$ (counting modulo m), such that $q_{y,j} = \text{IN}$. (Such a y must exist since x occurs in the list.) Then there is a $w \in \{y+1, y+2, \cdots, x-1\}$ such that $q_{w,j} \in \{\text{OUT}, P\text{-GATE}\}$.*

Proof. If $q_{x,j} = \text{IN}$, then $r_{x,j-1} = s_{x-1,j-1} = t$. So $q_{x-1,j-1} = g_{x-1,j-1} = \text{TAILS}$. Suppose $r_{x-1,j-1} \neq t$. After receiving $r_{x-1,j-1}$, P_{x-1} moves from TAILS to either OUT or P-GATE, and $w = x-1$ satisfies the lemma.

Suppose, instead, that $r_{x-1,j-1} = t$. Then $q_{x-2,j-1} = \text{TAILS}$. Again, if $r_{x-2,j-1} \neq t$, then $w = x-2$ satisfies the lemma. Otherwise the search continues through $x-3$, $x-4, \cdots, y+1$ until a w is found such that $q_{w,j-1} = \text{TAILS}$ and $r_{w,j-1} \neq t$. Such a w must exist since

$$q_{y,j} = \text{IN} \Rightarrow q_{y,j-1} = \text{HEADS} \Rightarrow s_{y,j-1} = h \Rightarrow r_{y-1,j-1} = h$$

so some first non-t message $r_{w,j-1}$ must be encountered among $r_{x-1,j-1}$, $r_{x-2,j-1}, \cdots, r_{y+1,j-1}$. That w satisfies the lemma. □

COROLLARY 3.8. *At least half of the active processors at phase k are passive at phase $k+1$.*

Proof. Lemma 3.7 associates with each processor P_x which remains active, a distinct processor P_w which becomes passive. The same P_w is not associated with two different P_{x_1} and P_{x_2}, since $q_{w,j}, q_{w,j'} \in \{\text{OUT}, P\text{-GATE}\}$ implies $j = j'$. □

COROLLARY 3.9. *There are at most $\lfloor \log c \rfloor$ phases during which more than one processor is active, where c is the number of candidates for leadership.*

If there remain more than one active processor in phase k, then with probability 1, at least two of them will eventually produce opposite coin flips, thus resulting in a transition to phase $k+1$. This observation, together with Corollary 3.9 ensures that the final requirement for correctness is satisfied.

3.1.3. Complexity analysis. Recall that the last phase of the attrition procedure, when only one active processor remains, is an infinite loop broken by the intervention of the solitude-verification algorithm. Therefore the complexity of concern for the attrition procedure is the expected number of bits sent, up to but not including the last phase. Corollary 3.9 establishes that there can be at most $\lfloor \log c \rfloor + 1$ phases. It remains to bound the expected number of messages sent per phase.

The following random variables over computations are needed:

$$M_x^{(k)} = \begin{cases} 0 & \text{if only one processor is active in phase } k, \text{ or if phase } k \text{ is not reached,} \\ m & \text{if more than one processor is active in phase } k, \text{ and processor } P_x \\ & \text{sends } m \text{ messages during phase } k. \end{cases}$$

$M_x^{(k)}$ is defined for both active and passive processors x and for all integers $k \geqq 1$.

LEMMA 3.10. $\Pr(M_x^{(k)} \geqq m) \leqq 2^{2-m}$.

Proof. The lemma is trivial if k is greater than or equal to the number of the last phase. Therefore suppose that there are two or more active processors in phase k. It is sufficient to consider the active processors only, since each passive processor sends the same number of messages as its nearest active predecessor.

The lemma is proved by induction on m. The case $m = 2$ is trivial. If, in phase k, P_x reaches state FLIP s times, then P_x sends $s+1$ messages. But whenever P_x reaches FLIP, there is a probability of at least $\frac{1}{2}$ that it will not return to FLIP in the same phase, since its coin toss is random relative to that of its nearest active predecessor. The induction follows immediately. □

COROLLARY 3.11. $E(M_x^{(k)}) \leqq 3$.

COROLLARY 3.12. *The expected number of bits communicated by the attrition procedure up to the last phase is at most $6n\lfloor \log c \rfloor$.*

Proof. The total number of messages sent up to the last phase is given by $\sum_{x=0}^{n-1} \sum_{k \geqq 1} M_x^{(k)}$. By Corollaries 3.9 and 3.11 this has an expected value of at most

$3n\lfloor \log c \rfloor$. Since there are only three distinct messages used, each can be encoded in two bits yielding an expected bit complexity of at most $6n\lfloor \log c \rfloor$. □

3.1.4. Interleaving properties. The attrition procedure lends itself naturally to being interleaved with a solitude-verification algorithm. Attrition messages can simply be annotated with solitude-verification messages. The * messages which serve to delimit phase boundaries are interpreted as solitude-verification restart signals. The interleaved algorithm proceeds exactly like attrition until the last phase, at which point the absence of * messages allows solitude verification to run to completion. This is summarized by the following theorem.

THEOREM 3.13. *Any conservative solitude-verification algorithm of complexity $f(n)$ can be combined with the attrition procedure to yield a leader-election algorithm of complexity $O(n \log c + f(n))$.*

Note that under the right conditions, a nonconservative solitude-verification algorithm can be interleaved with attrition to achieve an efficient leader-election algorithm, but Theorem 3.13 suffices here.

3.2. Solitude verification algorithms. In the absence of any information about the ring, solitude verification with certainty is impossible. Therefore solitude-verification algorithms must use specific ring information to verify that there is a sole active processor. Two cases are considered here.

(1) Each processor P_x has a distinct identifier I_x, consisting of a string such that if $x \neq w$, then I_x is not a prefix of I_w.

(2) Each processor knows the size of the ring to within a factor of 2.

Individually, these assumptions are as weak as possible in the sense that if processors' identities can appear at most twice or the size of the ring is not known to within a factor of 2, solitude verification remains impossible. Though solitude verification is all that is required for leader election, the algorithms are, in fact, solitude-detection algorithms.

3.2.1. Solitude detection with distinct identifiers. Suppose each processor, P_x, has a distinct identifier, I_x, which is not the prefix of any other identifier in the ring. Processor P_x uses an internal string variable J_x, which is initialized to the empty string. Each initiator alternately sends the jth bit of its own identifier and receives the jth bit of its nearest active predecessor, which it appends to J_x. Thus P_x builds up in J_x the identifier of its nearest active predecessor. If I_x contains m bits then after receipt of at most m bits P_x can declare, by comparing J_x and I_x, whether or not it is alone. Because no identifier is a prefix of any other identifier, P_x can never falsely claim solitude.

THEOREM 3.14. *If processors have distinct m bit identifiers, then conservative and deterministic solitude detection can be achieved with distributive termination using at most $O(mn)$ bits.*

3.2.2. Solitude detection when the size of the ring is known approximately. Suppose that distinct identifiers are not available, but each processor knows the size n of the ring. In this case, a nonconservative algorithm for determining solitude has each initiator send a counter which is incremented and forwarded by each passive processor until it reaches an initiator, P_x. By comparing the received counter with n, P_x knows whether or not it is alone. This algorithm can be transformed into a conservative solitude-detection algorithm without any increase in bit-communication complexity.

Each processor P_x, whether active or passive, maintains a counter c_x, initialized to 0. Let $d_x > 0$ denote the distance from P_x to its nearest active predecessor. The algorithm maintains the invariant:

$$\text{if } P_x \text{ has received } j \text{ bits then } c_x = d_x \bmod 2^j.$$

Then if P_x reaches a state where $c_x = n$, there must be $n-1$ passive processors preceding P_x, so P_x can conclude that it is the sole initiator. It remains to describe the strategy necessary to maintain the invariant.

Initiators first send 0. Thereafter, all processors alternately send and receive bits. If P_x is passive, then P_x is required to send the jth low-order bit of d_x as its jth message. Initiators continue to send 0. Suppose a processor P_y has the lowest order j bits of d_y in c_y. A simple inductive argument shows that when P_y receives its $(j+1)$st message (by assumption the $(j+1)$st bit of $d_y - 1$), it can compute the first $(j+1)$ bits of d_y and thus can update the value of c_y to satisfy $c_y = d_y \bmod 2^j$.

In the previous algorithm, it was assumed that n is known exactly. Suppose instead, that each processor knows a quantity N, such that $N \leqq n \leqq 2N-1$. Then there can be at most one gap of length N or more between neighbouring active processors. Thus any gap of less than N confirms nonsolitude and any processor detecting a gap of N or more can determine solitude by initiating a single checking round. (For the purposes of leader election, it is sufficient for any active processor that detects a gap of N or more to declare itself the leader, since it has confidence that no other processor can do the same.) Thus, the algorithm can be used when n is known to within a factor of less than 2.

THEOREM 3.15. *If each processor knows a value N such that the ring size n satisfies $N \leqq n \leqq 2N-1$, then conservative and deterministic solitude detection can be achieved using at most $O(n \log n)$ bits.* □

3.3. Time complexity of the leader-election algorithm. The usual notion of the time complexity of an asynchronous algorithm becomes particularly simple for a unidirectional ring since message delays cannot influence computation sequences. The *time complexity* of a randomized algorithm on a unidirectional ring is the expected number of unit time intervals before the algorithm terminates, under the assumption that messages travel each communication link in at most unit time, and local processing is instantaneous.

The time complexity of [DKR] is $O(n)$. Our leader-election algorithm, as described, requires $O(n \log n)$ time steps even for just the final phase, when verification bits are sent and received one at a time by the sole remaining active processor. It therefore might appear that the algorithm achieves an improvement by a factor of $O(\log n)$ in bits, only at the expense of an additional factor of $O(\log n)$ in time over other leader-election algorithms. But a slight alteration in the attrition algorithm reduces the time complexity of the final phase from $O(n \log n)$ to almost linear. Suppose that the ith message of a phase of attrition contains a package of $f(i)$ annotated coin flips rather than a single coin flip as previously described. Let $f(1) = 1$ and $f(i) = 2^{f(i-1)}$ for $i > 1$. With this packaging, in the final phase of leader election, the remaining active processor will initiate only $O(\log^* n)$ messages which propagate around the ring. $O(n \log^* n)$ time steps are used to confirm solitude when ring size is known. Similar packaging yields $O(n \log^* m)$ timesteps when identifiers of length m are used to confirm solitude. Clearly this packaging does not alter the bit complexity of the solitude-verification stage of leader election, nor does it significantly affect the bit complexity of the earlier phases of attrition. The probability that a processor sends at least k packages of coin flips in a given phase is no more than $2^{-(f(1)+\cdots+f(k-1))}$. Hence

the expected number of coin-flip bits sent by an arbitrary processor in a given phase is no more than $\sum_{k=1}^{\infty} f(k)/2^{f(1)+\cdots+f(k-1)} = \sum_{k=1}^{\infty} 1/2^{f(1)+\cdots+f(k-2)} = O(1)$. So the expected number of annotated coin-flip bits sent by a processor in a phase is also constant as it was without the packaging.

It remains to analyse the expected time for attrition up to the last phase. Imagine that each initiator creates an envelope and sends its first message in it. Since the algorithm is message-driven, it can be thought to proceed as follows. Each processor waits to receive an envelope. Upon reception of an envelope, a processor examines its contents and either forwards the envelope (with a possibly new message) or destroys the envelope. The nature of the algorithm guarantees that at the beginning of each phase, the number of active processors is equal to the number of envelopes. Thus in order to bound the expected time for attrition, it is sufficient to bound the expected time until the number of envelopes is reduced to one. The analysis of the bit complexity of attrition implies that attrition has the following property *P*.

> *P*: If at some time there are ω envelopes on the ring, then the expected number of remaining messages exchanged until one envelope is left is at most $cn \log \omega$ for some constant c.

Let the random variable X_l be the number of envelopes at time nl. Then nX_{l+1} is no greater than the total number of messages from time nl until time $n(l+1)$. Given that there are ω envelopes at time nl, it follows from property *P* that this total has expected value at most $cn \log \omega + n$. Therefore $E(X_{l+1}|X_l = \omega) \leqq c \log \omega + 1$. Now

$$\begin{aligned} E(X_{l+1}) &= \sum_{\omega} E(X_{l+1}|X_l = \omega) \Pr(X_l = \omega) \\ &\leqq \sum_{\omega} (c \log \omega \Pr(X_l = \omega)) + 1 \\ &\leqq c \log \sum_{\omega} \omega \Pr(X_l = \omega) + 1 \quad \text{by the convexity of log,} \\ &= c \log E(X_l) + 1. \end{aligned}$$

Thus at time $n \log^* I$, where I is the number of initiators, the expected number of envelopes is reduced to a constant, say c'. Again by property *P*, given that $X_{\log^* I} = \omega$, the expected number of messages remaining to the completion of attrition following $n \log^* I$ timesteps is at most $cn \log \omega < cn\omega$. The expected remaining time, given $X_{\log^* I} = \omega$, is therefore also less than $cn\omega$. So the expected remaining time is less than $cn \sum_{\omega} \omega \Pr(X_{\log^* I} = \omega) = cnc'$. Thus the expected total time for attrition with I initiators is less than $n \log^* I + cnc'$. Combining these results for attrition and solitude verification yields the following theorem.

THEOREM 3.16. *The leader-election algorithm which results from combining the attrition procedure of* § 3.1 *and the solitude-verification algorithm of* § 3.2.1 (§ 3.2.2) *can be adapted to have time complexity* $O(n \log^* m)$ ($O(n \log^* n)$) *without any increase in the order of the communication complexity.*

4. Lower bounds for solitude verification. This section provides lower bounds on the number of bits of communication required for solitude verification. Two cases, paralleling those considered in § 3.2, are studied. In each case lower bounds are developed which show that the algorithms discussed earlier are essentially optimal.

The lower bounds apply even to nondeterministic distributed algorithms. In order to understand the generality of the lower bounds, a formal model of nondeterministic

algorithms and their computations are presented in § 4.1. In § 4.2, the solitude-verification and leader-election problems are defined in terms of the model. That section also contains an $O(n)$ reduction from solitude verification to leader election within the model, implying that even the general nondeterministic lower bounds for solitude verification translate to the same lower bounds for leader election. Techniques and tools common to the proofs are grouped together in § 4.3. Sections 4.4 and 4.5 study, respectively, the cases where processors have distinct identifiers, and processors know the approximate size of the ring.

4.1. Model of computation. In order to describe a computation, § 4.1.1 defines a process and states some useful properties of rings of processes. Section 4.1.2 clarifies the relationship between distributed algorithms and rings of processes.

4.1.1. Processes. The following description of a process incorporates two non-restrictive assumptions [PR], namely, that messages are self-delimiting, and that communication is message-driven with at most one message sent in response to receipt of a message.

A *message* is an element of $M = \{0, 1\}^* \cdot \square$. The symbol $\square$ is called the end-of-message marker. A *history* is a (possibly infinite) sequence of messages. If h is a finite history, then $\|h\|$ denotes the length of the binary encoding of h using some fixed encoding scheme to encode each symbol in $\{0, 1, \square\}$. Note that any history has a unique parse into a sequence of messages.

A *process* π is modelled as a pair of mappings which describe the next output message (possibly null) and the next state of π as a function of its current state and current input message. Notice that this models a deterministic process. A process' state encodes its entire history so far, and consequently the state set is not bounded. Process π is an *initiating process* if it produces an output message from its initial state (that is, before the arrival of the first message). Otherwise it is a *noninitiating process.*

Let $\pi_1, \cdots, \pi_n$ be a sequence of processes. We use $\pi_{1,n}$ to abbreviate $\pi_1, \cdots, \pi_n$. There is a sequence of histories, $C = h_1, \cdots, h_n$, called a *computation* associated with $\pi_{1,n}$ in the natural way. Each history h_i is composed of a sequence of messages $m_1^i \cdots m_{r_i}^i$. If π_i is an initiating process, then m_1^i is the message produced by π_i from its initial state. The computation is then determined inductively by applying the mappings defined by π_1 through π_n and letting successive non-null output messages of π_i be successive input messages of π_{i+1}. (Indices are reduced in the obvious way.) If the h_i are finite, then C is said to *terminate.* A terminating computation $C = h_1, \cdots, h_n$ has *complexity* equal to $\sum_{i=1}^{n} \|h_i\|$.

We distinguish a subset $M_a \subseteq M$ called *accepting* messages, and a subset $M_r \subseteq M$ called *rejecting* messages. A history is an *accepting* history (respectively, *rejecting* history) if and only if its *last* message is an accepting message (respectively, rejecting message). An accepting or rejecting history is a *decisive* history. All other histories are *indecisive* histories. If a computation $C = h_1, \cdots, h_n$ terminates, then each h_i has a last message and $f(h_i) \in \{\text{accepting, rejecting, indecisive}\}$ denotes its type. By extension, $f(C)$, the *final state* of C, abbreviates the sequence $f(h_1), \cdots, f(h_n)$.

Two types of termination for a distributed computation may be considered. A weak form of termination, *nondistributive termination*, allows processors to reach tentative conclusions which are firm only if all message traffic has terminated for the entire ring (possibly an undetectable condition.) This conclusion is open to revision upon receipt of additional messages. Under the usual, stronger type of termination called *distributive termination*, it is required that processors come to irrevocable

decisions and halt computation. Distributive termination is modelled by insisting that processes never output another message after sending a decisive message. Since the lower bounds for this paper hold even for nondistributively terminating algorithms, the model permits accepting and rejecting messages to occur throughout the histories of a computation.

4.1.2. Algorithms. A *labelled ring* is a unidirectional n-ring of processors $R = P_1, \cdots, P_n$ where each processor P_i has associated with it a *label* l_i from ID $\times$ {initiator, noninitiator} where ID denotes some fixed universe of identifiers. Depending upon the assumptions, a processor's identifier in ID may or may not be distinct. The second label field is used to distinguish processors that initiate a computation from those that only participate in response to other initiators. The ring R is said to be *labelled* by $\mathscr{L} = l_1, \cdots, l_n$.

A distributed algorithm can be viewed as an assignment of processes to processors. A *deterministic* algorithm for a labelled ring is a mapping from labels to (deterministic) processes where initiating (respectively, noninitiating) processes are assigned to initiators (respectively, noninitiators). A more general notion of an algorithm can be obtained by relaxing the constraint that the assignment be deterministic. Let $\mathscr{A}$ be the set of all processes as described in the preceding section. A (*distributed*) *algorithm* α is a mapping from labels to nonempty subsets of $\mathscr{A}$ such that sets of initiating processes are assigned to initiators and sets of noninitiating processes are assigned to noninitiators. The set $\alpha(l)$ is the collection of processes available to processors with label l. A sequence $\pi_{1,n} = \pi_1, \cdots, \pi_n$ with $\pi_i \in \alpha(l_i)$ corresponds to an arbitrary assignment of processes to processors on the n-ring labelled with the sequence $l_1, \cdots, l_n$. A *computation* of α on a ring labelled by $\mathscr{L} = l_1, \cdots, l_n$ is a computation induced by any process sequence $\pi_1, \cdots, \pi_n$ where $\pi_i \in \alpha(l_i)$.

The generalization from deterministic to arbitrary assignments gives algorithms a nondeterministic attribute. Like conventional nondeterministic algorithms, an algorithm is said to solve a problem efficiently on a ring labelled by $\mathscr{L} = l_1, \cdots, l_n$ is for *some* choice of process assignments consistent with $\mathscr{L}$, the resulting computation provides a solution and has low complexity.

Notice that the usual notion of a randomized algorithm is subsumed by this general definition of an algorithm. In the natural description of a randomized distributed algorithm, a process' next state and output message are determined by its current state, its last input message, and the result of a random experiment. Random choices occur throughout the run of the algorithm. But these random choices can be simulated as a single random choice by each processor at the beginning of the algorithm. A processor randomly chooses a function from internal state input message pairs to internal state output message pairs. (Essentially, the processor pre-selects all its random coin tosses.) Hence a randomized distributed algorithm can be modelled as a *random* assignment of deterministic processes to labels. This is further generalized by permitting an *arbitrary* assignment of processes to labels. Therefore lower bounds on the complexity of a problem under this model apply to lower bounds for randomized algorithms and in fact address the complexity of the *best-case* computation.

The following definition captures the notion of what it means for an algorithm to solve a given problem. Let T be a predicate defined over elements of $N \times F$, where N is the set of all label sequences in (ID $\times$ {initiator, noninitiator})^n, and F is the set of all final states in {accepting, rejecting, indecisive}^{n}. Let R be a ring labelled by $\mathscr{L} = l_1, \cdots, l_n$ and C be the computation of some process sequence $\pi_{1,n} = \pi_1, \cdots, \pi_n$, where $\pi_i \in \alpha(l_i)$. Then $\pi_{1,n}$ *respects* T *for label sequence* $\mathscr{L}$ if $T(\mathscr{L}, f(C))$. An algorithm

α computes T for label sequence $\mathscr{L}$ if every $\pi_{1,n}$, with $\pi_i \in \alpha(l_i)$, respects T for label sequence $\mathscr{L}$. An algorithm α *computes* T if it computes T for every element of N.

4.2. The problems. Although the algorithms of § 3 always solve leader election, the corresponding lower bounds even apply to a weak version of leader election which requires that at most one processor be left, at termination, in a distinguished final state. Formally, the *weak leader-election* predicate, LE, is defined over $N \times F$ by: $LE(\mathscr{L}, f(C)) =$ (if $\mathscr{L}$ contains at least one initiator then there exists at most one "accept" in $f(C)$). A weak version of solitude verification requires that if any initiator is left in an accepting final state at termination, then that initiator is the only initiator of the computation. Formally, the *weak solitude-verification* predicate, SV, is defined over $N \times F$ by: $SV(\mathscr{L}, f(C)) =$ (if there exists i such that $f(h_i)$ is "accept," then l_i is the only initiating label in $\mathscr{L}$).

Notice that an algorithm has to meet only a rather weak requirement in order that it be said to solve one of these problems. For example, deadlocking computations, leaving all processors undecided, are tolerated. This only serves to strengthen the lower-bound results that follow. Conclusions are drawn about the number of bits that must be transmitted in any computation of an algorithm which succeeds in verifying solitude (a *successful* computation), even if the algorithm solves the problem in only this weak sense. A *successful* solitude verification computation has exactly one initiating process π_i and its history h_i in $C = h_1, \cdots, h_n$ is an accepting history. A *successful* leader-election computation has exactly one of the initiators left in final state "accept." Nonsolitude can be ascertained with low expected communication complexity, but we wish to focus on the cost of verifying that there is only one initiator. Therefore the complexity of solitude verification is defined to be the expected complexity when solitude is asserted. Let α be an algorithm that computes SV. Say that α *has complexity* at least $f(n)$ on rings of size n if every successful computation of α has complexity at least $f(n)$.

Section 2 contains a description of how a randomized, distributively terminating algorithm for electing a leader from among all processors on the ring can be converted, using an additional $O(n)$ bits of communication, to an algorithm for solitude detection (and hence solitude verification). In fact this $O(n)$ reduction holds even for the nondeterministic, nondistributively terminating model, and for the weak versions of the two problems. Initiators, wishing to determine their solitude, alert the ring by propagating a wake-up message. All processors, having been alerted, nondeterministically choose whether or not to be candidates for leadership, and run the weak leader-election algorithm. A candidate remaining in contention guesses if and when the leader-election algorithm terminates with itself as leader. At this time the elected leader circulates a single message of constant length to determine whether one or more than one original initiator was present. A final round announces the result. Because the portion of this algorithm following leader election is deterministic, the reduction converts successful computations of leader election to correct computations of solitude detection. Unsuccessful leader election can happen if either no processor chose to be a candidate or if the weak leader election algorithm left all processors in a nonleader state. But in both of these cases the corresponding solitude-detection computation deadlocks, satisfying weak solitude verification. Finally, if a candidate guesses erroneously that it is the sole remaining contender before weak leader election has terminated (nondistributively), then eventually the leader-election algorithm must correct this error. So the resulting solitude-verification computation is also eventually alerted to the error, and correctly achieves nondistributive termination. Thus nonlinear

lower bounds for computations that verify solitude translate to lower bounds for computations that elect a leader even for this general model. Notice that nondeterminism allows this reduction to hold for the general version of leader election when a leader is elected from c candidates rather than from the fixed configuration of all processors on the ring. Thus the lower bounds for solitude verification imply lower bounds for a best-case configuration of candidates for leadership.

4.3. Tools. In the lower-bound results that follow, two techniques are used to convert low-complexity computations of an algorithm α on a ring R labelled by $\mathscr{L} = l_1, \cdots, l_n$ to new computations of α on a related ring R'. Let $C = h_1, \cdots, h_n$ be the computation of $\pi = \pi_1, \cdots, \pi_n$ where $\pi_i \in \alpha(l_i)$. Suppose $h_i = h_j$ and $h_{i+1}, \cdots, h_j$ contains no initiators. Consider the new sequence $C' = h_1, \cdots, h_i, h_{j+1}, \cdots, h_n$ formed by removing $h_{i+1}, \cdots, h_j$ from C. Then C' is a computation of $\pi_1, \cdots, \pi_i, \pi_{j+1}, \cdots, \pi_n$. Thus C' is a computation of α on the ring R' labelled by $\mathscr{L}' = l_1, \cdots, l_i, l_{j+1}, \cdots, l_n$. This process of forming C' on R' from C on R when $h_i = h_j$, while retaining all initiators, is referred to as *collapsing.*

Let $C_1 = h_1^1, \cdots, h_n^1$ and $C_2 = h_1^2, \cdots, h_m^2$ be computations of α on the two rings R_1 and R_2 labelled by $\mathscr{L}_1 = l_1^1, \cdots, l_n^1$ and $\mathscr{L}_2 = l_1^2, \cdots, l_m^2$. Suppose that $h_i^1 = h_j^2$ for some i and j. Consider the sequence $C' = h_1^1, \cdots, h_i^1, h_{j+1}^2, \cdots, h_m^2, h_1^2, \cdots, h_j^2, h_{i+1}^1, \cdots, h_n^1$ formed by combining C_1 and C_2. Then C' is a computation of α on R' labelled by $\mathscr{L}' = l_1^1, \cdots, l_i^1, l_{j+1}^2, \cdots, l_m^2, l_1^2, \cdots, l_j^2, l_{i+1}^1, \cdots, l_n^1$. This process of forming C' on R' by combining C_1 on R_1 and C_2 on R_2 is referred to as *splicing.* The special case of splicing a computation to itself is called *doubling.*

The following lemma takes advantage of the fact that any collection of k distinct binary strings contains at least $(k \log k)/2$ bits for $k > 3$.

LEMMA 4.1. *If $C = h_1, \cdots, h_n$ is a computation with complexity less than $cm \log m$, and $m \leqq n$, then there exist i and j, $0 < j - i < m^{4c}$ such that $h_i = h_j$.*

Proof. Let k be the maximum integer such that every subsequence of C containing k histories has each history distinct. The encoding of a subsequence of k distinct histories must contain a total of at least $(k \log k)/2$ bits. But C can be decomposed into $\lfloor n/k \rfloor \geqq \lceil n/(2k) \rceil$ disjoint subsequences each with at least k histories. Since the complexity of C is less than $cm \log m$, at least one of these subsequences must have less than $(cm \log m)/\lceil n/(2k) \rceil \leqq 2ck \log m$ bits. Hence, $(k \log k)/2 < 2ck \log m$. Thus $k < m^{4c}$. □

The lower-bound arguments both have similar structure. It is assumed that an algorithm α exists that solves solitude verification on a ring R with nondistributive termination and that α has a successful computation C with small communication complexity. It follows, by some combination of collapsing and splicing, that C can be transformed into a computation C' of α on a different ring R', in which more than one initiator terminates in state "accept." Thus α does not solve solitude verification on R'.

4.4. Distinct identifiers. The objective of this section is to characterize the complexity of algorithms that solve the solitude-verification problem on rings of processors with distinct identifiers chosen (otherwise arbitrarily) from a set ID of size s. Let α be any algorithm that solves weak solitude verification on all n-rings with distinct identifiers chosen from ID. There are $\binom{s}{n}$ possible identifier sets for such an n-ring. Suppose that for each of these sets, there is at least one permutation of the identifier set, such that for an n-ring labelled with this permutation and with exactly one initiator, α has some successful computation. α is then said to *nontrivially* compute solitude verification for n-rings with identifiers chosen from ID.

THEOREM 4.2. *Let α be any algorithm that nontrivially computes weak solitude verification on rings of between N and $2N$ processors with distinct identifiers chosen from a universe* ID *of size $s \geqq 2N$. Then for at least half of the $\binom{s}{N}$ identifier sets L of size N, the complexity of α on every ring labelled by some permutation of L is $\Omega(N \log (s/2N))$.*

Proof. Let L be any subset of ID of size N and let $C = h_1, \cdots, h_N$ be a successful computation of $\pi_{1,N} = \pi_1, \cdots, \pi_N$ where the identifiers associated with $\pi_{1,N}$ are distinct identifiers chosen from L. C is a *cheap computation* if it has complexity less than $N \log (s/2N)$. If a cheap C exists, the process sequence $\pi_{1,N}$ and the identifier set L are also said to be cheap. If C is cheap, then at least one of the histories $h_1, \cdots, h_N$ must have length less than $l = \log (s/2N)$. Choose any such history and call it the cheap history associated with C and indirectly associated with $\pi_{1,N}$ and L. Now suppose that α has the property that for at least one half of the $\binom{s}{N}$ possible choices for L, there exists a cheap successful computation of α and therefore an associated cheap history. Among all the partitions of ID into s/N subsets of size N, at least one such partition must have among its subsets at least $s/2N$ cheap identifier sets. Therefore there exist $s/2N$ disjoint, cheap labellings with corresponding cheap successful computations. But there are fewer than $2^l = s/2N$ distinct cheap histories in total. So some cheap history must be associated with successful computations of α on two process sequences with disjoint sets of identifiers. If these computations are spliced at their common history, the result is a decisive computation of α on a ring of size $2N$, whose processors all have distinct identifiers. However, this computation leaves two processors in the final state "accept" contradicting the correctness of α. □

COROLLARY 4.3. *Let α be an algorithm that meets the conditions of Theorem* 4.2. *If the size s of the universe* ID *is $\Omega(N^{1+\varepsilon})$ for some $\varepsilon > 0$, then for at least half of the identifier sets L of size N, the complexity of α on any ring labelled by some permutation of L is $\Omega(N \log s)$.*

4.5. Ring size known approximately. If ring size n is to be used to verify solitude, it must be known to within a factor of two. The objective of this subsection is to characterize the complexity of computations that verify solitude, as a function of processors' uncertainty of ring size within this limit.

THEOREM 4.4. *Let α be any nondistributively terminating algorithm that computes solitude verification on the class of all rings of size n, where $n \in [N, N+\Delta]$ for some N and $0 < \Delta < N$. Then the complexity of α on any ring in this class is $\Omega(N \log \Delta)$.*

Proof. Let $C = h_1, \cdots, h_n$ be any successful computation of α on a ring of size n, where $N \leqq n \leqq N + \Delta$. Then $f(h_i) =$ "accept" for some initiator π_i. Without loss of generality, suppose $i = 1$. Let $\varepsilon = \log \Delta / \log N$. Suppose that the communication complexity of C is less than $(N \log \Delta - 2N)/4 = (\varepsilon N \log N - 2N)/4$. By Lemma 4.1, there exist i and j such that $1 < j - i < N^{\varepsilon} = \Delta$ and $h_i = h_j$. Consider the new computation $h_1, \cdots, h_i, h_{j+1}, \cdots, h_n$ formed by removing $S = h_{i+1}, \cdots, h_j$ and apply repeated collapsing to each subsequence $H_1 = h_1, \cdots, h_i$ and $H_2 = h_{j+1}, \cdots, h_n$ separately until each of these subsequences is composed of distinct histories. Let the resulting computation be $C' = h'_1, \cdots, h'_l, h'_{l+1}, \cdots, h'_m$, where $h'_1 = h_1$, $h'_l = h_i$, and $h'_m = h_n$. Since $h'_1, \cdots, h'_l$ and $h'_{l+1}, \cdots, h'_m$ are sequences of distinct histories, their combined communication complexity is at least $l \log l/2 + (m-l) \log (m-l)/2$, which is at least $(m/2) \log (m/2)$. Thus their combined length m must not exceed $N/2$, since otherwise, the assumption on the complexity of C is violated. Therefore C' can be doubled to form a new computation, C'', of α on a ring of size $2m \leqq N$ processors. But since the subsequence S, which was originally removed, has length $s < \Delta$, there exists an integer $k \geqq 0$ such that $N \leqq 2m + ks \leqq N + \Delta$. Thus k copies of S can be spliced into C'', after

either history identical to h_i, forming a new computation of α on a ring of size $n' \in [N, N+\Delta]$ which contains two histories identical to h_1. Thus two processors conclude solitude, contradicting the correctness of α for rings of size $n \in [N, N+\Delta]$. □

It can be shown that Theorem 4.4 holds even if the algorithm α is required to work correctly only on the class of rings in which processors have identifiers, and no processor identifier appears more than twice. If all processor identifiers are known to be distinct then the results are different, as was shown in § 4.4.

5. Conclusions.

5.1. Technical results. The inherent communication complexity, measured in terms of the expected number of bits, of electing a leader in a ring of processors has been identified to within constant factors for two cases. When all processors know the ring size to be within some interval $[N_l, N_u]$ and all processors have distinct identifiers drawn from some set of size $s \geqq N_u^{1+\varepsilon}$, where $\varepsilon > 0$, then for all n satisfying $N_l \leqq n \leqq N_u/2$, the average, over all n-rings, of the expected bit complexity of randomized leader election is $\Theta(n \log s)$. On the other hand, if the ring size is known to be in some interval $[N_l, N_u]$ where $N_l + N_l^{\varepsilon} \leqq N_u < 2N_l$, for some $\varepsilon > 0$, and processor identifiers are not necessarily distinct then, for all n satisfying $N_l \leqq n \leqq N_u$, the expected bit complexity of randomized leader election is $\Theta(n \log n)$.

The results for leader election stem from bounds on the complexity of two more primitive processes called attrition and solitude verification. The identification of these subproblems and the clarification of their relationship to leader election is one of the important contributions of this paper. Efficient conservative solitude verification algorithms that exploit known properties of a ring can be combined with the randomized attrition procedure described in § 3.1 to provide new efficient leader-election algorithms. Solitude verification is of equal interest for its role in the proof of lower bounds for leader election. For all solitude-verification computations of concern there is only one initiator, which considerably simplifies the analysis. This is reflected in the strong lower bounds of § 4.

5.2. Related issues. In addition to the specific technical contributions cited above the results of this paper shed light on a number of important issues in distributed computing. These are summarized under three general headings below.

5.2.1. Global knowledge of ring. Suppose that all processors know that the ring size n lies in the interval $[N_l, N_u]$. If the processors are indistinguishable then deterministic leader election is impossible [A], even if $N_l = N_u$. Furthermore, if $N_u \geqq 2N_l$ then even randomized algorithms cannot elect a leader among indistinguishable processors with certainty. However, if $N_u < 2N_l$, then randomized leader election can be achieved in $O(n \log n)$ expected bits. If, in addition, $N_l + N_l^{\varepsilon} \leqq N_u$, for some $\varepsilon > 0$ (i.e., the interval is not too small), then $\Omega(n \log n)$ bits are required to elect a leader among indistinguishable processors.

On the other hand, even if $N_l = 1$ and $N_u = \infty$, if processors have distinct identities chosen from a universe S of size s (which need not be known explicitly) then a leader can be elected with $O(n \log s)$ expected bits. In fact, assuming $N_u \geqq 2N_l$ and $n \leqq s/2$, $\Omega(n \log (s/n))$ bits are required to elect a leader with distinct identities from S.

5.2.2. Type of algorithm. The leader-election algorithms described in this paper are all randomized. In fact, the solitude-verification process is deterministic. The algorithms cannot deadlock. They all terminate distributively with probability 1 and elect a leader (or detect solitude) with certainty. Finally, with the exception of those modifications described in § 3.3, the algorithms are all conservative.

In contrast to the above, the lower bounds on solitude verification (and hence leader election) are proved on a nondeterministic model of computation. The model admits algorithms that may deadlock. Furthermore algorithms may communicate nonconservatively, may terminate nondistributively, and may, in the case of solitude verification, tolerate errors when there is only one initiator. The lower bounds state a minimum bit complexity of any computation that provides a certificate of solitude.

The juxtaposition of the algorithms and model of computation highlight a remarkable insensitivity for the problems and complexity measure studied in this paper, to the details of the underlying model of computation. This insensitivity is not preserved when the focus shifts to certain closely related problems [AAHK1], [AAHK2].

5.2.3. Type of analysis. The solitude-verification algorithms are analysed with respect to the worst-case number of bits of communication. The lower bounds refer to the best-case number of bits communicated by computations of algorithms that certify solitude.

The bulk of earlier results on leader election are concerned with message complexity. The leader-election algorithms of this paper are competitive in this measure while improving upon earlier results by a factor of $\log n$ in the number of bits transmitted. While the obvious implementation of the leader-election algorithms of this paper on a synchronous model makes them somewhat unattractive in terms of communication time, implementations exist, as described in § 3.3, which for all practical purposes make the algorithms comparable with earlier algorithms in this measure as well.

5.3. Extensions. The results of the present paper can be extended in two natural directions. First, the case where the ring size n is known exactly—a situation where the upper and lower bounds of this paper do not agree—can be explored in more detail. The solitude-verification problem when n is known exactly is examined in [AAHK2]. In this case number-theoretic properties of n can be exploited to improve upon the $O(n \log n)$ algorithm contained in this paper. With exact knowledge of ring size, there is a distinction between the complexity of distributively and nondistributively terminating versions of solitude verification. $\Theta(n\sqrt{\log n})$ bits are necessary and sufficient to achieve solitude verification with distributive termination. This becomes $\Theta(n \log \log n)$ bits for nondistributive termination. The upper bounds in this case are achieved by nondeadlocking, deterministic algorithms, and the lower bounds apply on the same general models as used in this paper. The algorithms are nonconservative. (If conservative solitude verification is required then $\Theta(n \log n)$ bits are necessary and sufficient [H].)

This paper is concerned with leader-election and solitude-verification when enough information is available to achieve certainty. When processor information is insufficient to confirm solitude with certainty, it is still possible to solve these problems probabilistically. Reference [AAHK1] examines probabilistic solitude verification, that is, algorithms that are correct with probability at least $1-\varepsilon$. When there is no knowledge of ring size, the communication complexity of solitude verification with nondistributive termination is $\Theta(n \log 1/\varepsilon)$ bits. (Distributive termination with probability $1-\varepsilon$ of correctness is impossible.) When ring size is known to be less than a bound N, then distributive termination can be achieved with complexity $O(n\sqrt{\log (N/n)}+n \log 1/\varepsilon)$ bits. A matching lower bound is shown for rings of actual size no larger than $N/2$.

Finally, probabilistic solitude detection with exact knowledge of ring size combines the two extensions above. In this case number-theoretic properties of n and error tolerance can be simultaneously exploited to reduce the complexity of probabilistic

solitude detection still further. The results for this version of the problem are more elaborate than those quoted here. The reader is directed to [AAHK2] for details.

REFERENCES

[A] D. ANGLUIN, *Local and global properties in networks of processors*, in Proc. 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 82-93.

[AAHK1] K. ABRAHAMSON, A. ADLER, L. HIGHAM, AND D. KIRKPATRICK, *Probabilistic solitude detection* I: *Ring size known approximately*, TR-87-8, University of British Columbia, Vancouver, British Columbia, Canada, 1987.

[AAHK2] ———, *Probabilistic solitude detection* II: *Ring size known exactly*, TR-87-11, University of British Columbia, Vancouver, British Columbia, Canada 1987.

[B] J. BURNS, *A formal model for message passing systems*, TR-91, Indiana University, September 1980.

[DKR] D. DOLEV, M. KLAWE, AND M. RODEH, *An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle*, J. Algorithms, 3 (1982), pp. 245-260.

[FL] G. FREDRICKSON AND N. LYNCH, *The impact of synchronous communication on the problem of electing a leader in a ring*, in Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 493-503.

[H] L. HIGHAM, Ph.D. thesis, University of British Columbia, Vancouver, British Columbia, Canada, in preparation.

[IR] A. ITAI AND M. RODEH, *Symmetry breaking in distributed networks*, in Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science, 1981, pp. 150-158.

[Pa] J. PACHL, *A lower bound for probabilistic distributed algorithms*, Res. Report CS-85-25, University of Waterloo, Waterloo, Ontario, Canada, August 1985.

[PKR] J. PACHL, E. KORACH, AND D. ROTEM, *Lower bounds for distributed maximum-finding algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 905-918.

[PR] J. PACHL AND D. ROTEM, *Notes on distributed algorithms in unidirectional rings*, in Proc. 1st International Workshop on Distributed Algorithms, 1985, pp. 115-122.

[Pe] G. PETERSON, *An $O(n \log n)$ unidirectional algorithm for the circular extrema problem*, Trans. Prog. Lang. Sys., 4 (1982), pp. 758-762.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 30-55, February 1989

003

A FAST PARAMETRIC MAXIMUM FLOW ALGORITHM AND APPLICATIONS*

GIORGIO GALLO†, MICHAEL D. GRIGORIADIS‡, AND ROBERT E. TARJAN§

Abstract. The classical maximum flow problem sometimes occurs in settings in which the arc capacities are not fixed but are functions of a single parameter, and the goal is to find the value of the parameter such that the corresponding maximum flow or minimum cut satisfies some side condition. Finding the desired parameter value requires solving a sequence of related maximum flow problems. In this paper it is shown that the recent maximum flow algorithm of Goldberg and Tarjan can be extended to solve an important class of such parametric maximum flow problems, at the cost of only a constant factor in its worst-case time bound. Faster algorithms for a variety of combinatorial optimization problems follow from the result.

Key words. algorithms, data structures, communication networks, complexity, flow sharing, fractional programming, graphs, knapsack constraint, linear programming, maximum flow, maximum-density subgraphs, network flows, network vulnerability, networks, nonlinear zero-one programming, ratio closure, record segmentation, parallel computations, parametric programming, provisioning, pseudoforest, scheduling, selection, sequencing

AMS(MOS) subject classifications. 68Q20, 68Q25, 68R05, 68R10, 05C20

1. Introduction.

The well-known *maximum flow problem* calls for finding a maximum flow (or alternatively a minimum cut) in a capacitated network. This problem arises in a variety of situations in which the arc capacities are not fixed but are functions of a single parameter, and the goal is to find the value of the parameter such that the corresponding maximum flow (or minimum cut) meets some side condition. The usual approach to solving such problems is to use a maximum flow algorithm as a subroutine and to use either binary search, monotonic search, or some other technique, such as Megiddo's [29], to find the desired value of the parameter.

Existing methods take no advantage of the similarity of the successive maximum flow problems that must be solved. In this paper, we address the question of whether this similarity can lead to computational efficiencies. We show that the answer to this question is yes: an important class of parametric maximum flow problems can be solved by extending the new maximum flow algorithm devised by Goldberg and Tarjan [13]. This algorithm is the fastest among all such algorithms for real-valued data, uniformly for all graph densities. The resulting algorithm for the parametric problem has a worst-case time bound that is only a constant factor greater than the time bound to solve a nonparametric problem of the same size. The parametric problems we consider are those in which the capacities of the arcs leaving the source are nondecreasing functions of the parameter, those of arcs entering the sink are nonincreasing functions of the parameter, and those of all other arcs are constant. Our parametric maximum flow algorithm has a variety of applications in combinatorial optimization.

* Received by the editors July 20, 1987; accepted for publication (in revised form) February 16, 1988. This research was partially supported by the National Science Foundation under grants MCS-8113503 and DCR-8605962, and by the Office of Naval Research under contracts N00014-84-K-0444 and N00014-87-K-0467.

† Departimento di Informatica, Università di Pisa, Pisa, Italy. Some of this work was done while this author was on leave at the Department of Computer Science, Rutgers University, New Brunswick, New Jersey from April to July 1986.

‡ Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903.

§ Department of Computer Science Princeton University, Princeton, New Jersey 08544 and AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

This paper consists of four sections in addition to the Introduction. In § 2 we extend the Goldberg-Tarjan algorithm to find maximum flows in an n-vertex, m-arc network for $O(n)$ ordered values of the parameter in $O(nm \log (n^2/m))$ time. In § 3 we use the algorithm of § 2 to compute information about the minimum cut capacity as a function of the parameter, assuming that each arc capacity is a linear function of the parameter. In this case, the minimum cut capacity is a piecewise linear concave function of the parameter. We describe successively more complicated algorithms for finding the smallest (or largest) breakpoint, finding a maximum, and finding all the breakpoints of this function. Each of these algorithms runs in $O(nm \log (n^2/m))$ time.

In § 4 we discuss applications of our parametric maximum flow algorithm to various combinatorial optimization problems. Depending on the application, our method is faster than the best previously known method by a factor of between $\log n$ and n. The applications include flow sharing problems [3], [18], [21], [22], [27], [28], zero-one fractional programming problems [4], [5], [11], [12], [24], [25], [30], [33]-[35], [39], and others [9], [43]. Section 5 contains a summary of our results and some final remarks.

2. Parametric maximum flows and the preflow algorithm. We begin in this section by reviewing the maximum flow algorithm of Goldberg and Tarjan [13], here called the *preflow algorithm.* Then we extend their method to the parametric maximum flow problem and we analyze three versions of the parametric preflow algorithm. We conclude with some remarks about the parametric problem and our algorithm for solving it.

2.1. Flow terminology. A *network* (see [10], [44]) is a directed graph $G=(V, E)$ with a finite vertex set V and arc set E, having a distinguished *source vertex* s, a distinguished *sink vertex* t, and a nonnegative *capacity* $c(v, w)$ for each arc (v, w). We denote the number of vertices by n and the number of arcs by m. We assume that for each vertex v, there is a path from s through v to t; this implies $n=O(m)$, since every vertex other than t has at least one exiting arc. We extend the capacity function to arbitrary vertex pairs by defining $c(v, w)=0$ if $(v, w) \notin E$. A *flow* f on G is a real-valued function on vertex pairs satisfying the following three constraints:

$$f(v, w) \leqq c(v, w) \quad \text{for } (v, w) \in V \times V \quad \text{(capacity)}, \tag{2.1}$$

$$f(v, w) = -f(w, v) \quad \text{for } (v, w) \in V \times V \quad \text{(antisymmetry)}, \tag{2.2}$$

$$\sum_{v \in V} f(u, v) = 0 \quad \text{for } v \in V - \{s, t\} \quad \text{(conservation)}. \tag{2.3}$$

The *value* of the flow f is $\sum_{v \in V} f(v, t)$. A *maximum flow* is a flow of maximum value.

If A and B are two disjoint vertex subsets, the *capacity* of the pair A, B is $c(A, B)=\sum_{v \in A, w \in B} c(v, w)$. A *cut* $(X, \bar{X})$ is a two-part vertex partition $(X \cup \bar{X}=V, X \cap \bar{X}=\emptyset)$ such that $s \in X$ and $t \in \bar{X}$. A *minimum cut* is a cut of minimum capacity. If f is a flow, the *flow across the cut* $(X, \bar{X})$ is $f(X, \bar{X})=\sum_{v \in X, w \in \bar{X}} f(v, w)$. The conservation constraint implies that the flow across any cut is equal to the flow value. The capacity constraint implies that for any flow f and any cut $(X, \bar{X})$, we have $f(X, \bar{X}) \leqq c(X, \bar{X})$, which in turn implies that the value of a maximum flow is no greater than the capacity of a minimum cut. The *max-flow min-cut theorem* of Ford and Fulkerson [10] states that these two quantities are equal.

2.2. The preflow algorithm. The preflow algorithm computes a maximum flow in a given network. To describe the algorithm we need two additional concepts, those of a *preflow* and a *valid labeling.*

A *preflow* f on G is a real-valued function on vertex pairs satisfying the capacity constraint (2.1), the antisymmetry constraint (2.2), and the following relaxation of the conservation constraint (2.3):

$$\sum_{u \in V} f(u, v) \geqq 0 \quad \text{for all } v \in V - \{s\} \quad \text{(nonnegativity)}. \tag{2.4}$$

For a given preflow, we define the *excess* $e(v)$ of a vertex v to be $\sum_{u \in V} f(u, v)$ if $v \neq s$, or infinity if $v = s$. The *value* of the preflow is $e(t)$. We call a vertex $v \notin \{s, t\}$ *active* if $e(v) > 0$. A preflow is a flow if and only if (2.4) is satisfied with equality for all $v \notin \{s, t\}$, i.e., $e(v) = 0$ for all $v \notin \{s, t\}$. A vertex pair (v, w) is a *residual arc* for f if $f(v, w) < c(v, w)$; the difference $c(v, w) - f(v, w)$ is the *residual capacity* of the arc. A pair (v, w) that is not a residual arc is *saturated*. A path of residual arcs is a *residual path*.

A *valid labeling* d for a preflow f is a function from the vertices to the nonnegative integers and infinity, such that $d(t) = 0$, $d(s) = n$, and $d(v) \leqq d(w) + 1$ for every residual arc (v, w). The *residual distance* $d_f(v, w)$ from a vertex v to a vertex w is the minimum number of arcs on a residual path from v to w, or infinity if there is no such path. A proof by induction shows that if d is a valid labeling, $d(v) \leqq \min \{d_f(v, t), d_f(v, s) + n\}$ for any vertex v.

The preflow algorithm maintains a preflow f, initially equal to the arc capacities on arcs leaving s and zero on arcs not incident to s. It improves f by pushing flow excess toward the sink along arcs estimated (by using d) to be on shortest residual paths. The value of f gradually becomes larger, and f eventually becomes a flow of maximum value. As a distance estimate, the algorithm uses a valid labeling d, initially defined by $d(s) = n$, $d(v) = 0$ for $v \neq s$. This labeling increases as flow excess is moved among vertices; such movement causes residual arcs to change.

To implement this approach, the algorithm uses an *incidence list* $I(v)$ for each vertex v. The elements of such a list, called *edges*, are the unordered pairs $\{v, w\}$ such that $(v, w) \in E$ or $(w, v) \in E$. Of the edges on $I(v)$, one, initially the first, is designated the *current edge* of v. The incidence lists $I(v)$ for all $v \in V$ can be generated from an arbitrarily ordered arc list E in $O(m)$ time.

The algorithm consists of repeating the following steps until there are no active vertices:

Push/Relabel. Select any active vertex v. Let $\{v, w\}$ be the current edge of v. Apply the appropriate one of the following three cases:

Push. If $d(v) > d(w)$ and $f(v, w) < c(v, w)$, send $\delta = \min \{e(v), c(v, w) - f(v, w)\}$ units of flow from v to w. This is done by increasing $f(v, w)$ and $e(w)$ by δ, and by decreasing $f(w, v)$ and $e(v)$ by δ. (The push is *saturating* if $\delta = c(v, w) - f(v, w)$ and *nonsaturating* otherwise.)

Get Next Edge. If $d(v) \leqq d(w)$ or $f(v, w) = c(v, w)$, and $\{v, w\}$ is not the last edge on $I(v)$, replace $\{v, w\}$ as the current edge of v by the next edge on $I(v)$.

Relabel. If $d(v) \leqq d(w)$ or $f(v, w) = c(v, w)$, and $\{v, w\}$ is the last edge on $I(v)$, replace $d(v)$ by $\min \{d(w) | \{v, w\} \in I(v) \text{ and } f(v, w) < c(v, w)\} + 1$ and make the first edge on $I(v)$ the current edge of v.

When the algorithm terminates, f is a maximum flow. A minimum cut can be computed as follows. For each vertex v, replace $d(v)$ by $\min \{d_f(v, s) + n, d_f(v, t)\}$ for each $v \in V$. (This replacement cannot decrease any distance label. The values $d_f(v, s)$ for all v and $d_f(v, t)$ for all v can be computed in $O(m)$ time by breadth-first searches backward

from s and t, respectively.) Then, the cut $(X, \bar{X})$ defined by $X = \{v \mid d(v) \geqq n\}$ is a minimum cut whose sink side $\bar{X}$ is of minimum size, a property that follows from Theorem 5.5 of Ford and Fulkerson [10]. If desired, a cut $(X', \bar{X}')$ of minimum-size source side can be computed as follows. For each $v \in V$ let $d'(v) = \min \{d_f(s, v), d_f(t, v) + n\}$, and let $X' = \{v \mid d'(v) < n\}$.

The efficiency of this "generic" form of the preflow algorithm depends on the order in which active vertices are selected for *push/relabel* steps. We shall consider this selection issue after we have extended the algorithm to the parametric problem. For the moment, we merely note the bounds derived by Goldberg and Tarjan for the generic algorithm (with any selection order).

LEMMA 2.1 [13]. *Any active vertex v has $d_f(v, s) < \infty$, which implies $d(v) \leqq 2n - 1$. The value of $d(v)$ never decreases during the running of the algorithm. The total number of relabel steps is thus $O(n^2)$; together they and all the get next edge steps take $O(nm)$ time.*

LEMMA 2.2 [13]. *The number of saturating push steps through any particular residual arc (v, w) is at most one per value of $d(v)$. The total number of saturating push steps is thus $O(nm)$; each such step takes $O(1)$ time.*

LEMMA 2.3 [13]. *The total number of nonsaturating push steps is $O(n^2 m)$; each such step takes $O(1)$ time.*

In all variants of the algorithm, the running time is $O(nm)$ plus $O(1)$ time per nonsaturating *push* step; making the algorithm more efficient requires reducing the number of such steps. This is also true in the parametric extension, as we shall see.

2.3. Extension to parametric networks. In a *parametric network*, the arc capacities are functions of a real-valued parameter λ. We denote the capacity function by c_λ and make the following assumptions:

(i) $c_\lambda(s, v)$ is a nondecreasing function of λ for all $v \neq t$.

(ii) $c_\lambda(v, t)$ is a nonincreasing function of λ for all $v \neq s$.

(iii) $c_\lambda(v, w)$ is constant for all $v \neq s$, $w \neq t$.

When speaking of a maximum flow or minimum cut in a parametric network, we mean maximum or minimum for some particular value of the parameter λ.

We shall address the problem of computing maximum flows (or minimum cuts) for each member of an increasing sequence of parameter values $\lambda_1 < \lambda_2 < \cdots < \lambda_l$. Successive values are given *on-line*; that is, λ_{i+1} need not be known until after the maximum flow for λ_i has been computed. In stating time bounds we shall assume that the capacity of an arc can be computed in constant time given the value of λ. (Such is the case if, for example, the arc capacities are linear functions of λ.) The algorithm we shall describe computes no more than $n - 1$ distinct minimum cuts, no matter how many values of λ are given. For all of our applications, $l = O(n)$.

We shall now extend the preflow algorithm to the parametric maximum flow problem. Suppose that for some value λ_i of the parameter we have computed a maximum flow f and a valid labeling d for f. What is the effect of changing the value of the parameter to λ_{i+1}? The capacity of each arc (s, v) may increase; that of each arc (v, t) may decrease. Suppose we modify f by replacing $f(v, t)$ by $\min \{c_{\lambda_{i+1}}(v, t), f(v, t)\}$ for each arc $(v, t) \in E$, and replacing $f(s, v)$ by $\max \{c_{\lambda_{i+1}}(s, v), f(s, v)\}$ for each arc $(s, v) \in E$ such that $d(v) < n$. The modified f is a preflow, since $e(v)$ for $v \notin \{s, t\}$ can only have increased. Furthermore, d is a valid labeling for the modified f, since the only new residual arcs are of the form (s, v) for $d(v) \geqq n$ and (v, s) for $d(v) < n$. This means that we can compute a maximum flow and a minimum cut for λ_{i+1} by applying the preflow algorithm beginning with the modified f and the current d.

This idea leads to the following *parametric preflow algorithm.* The algorithm finds a maximum flow f_i and a minimum cut $(X_i, \bar{X}_i)$ for each value λ_i of the parameter. It consists of initializing $f=0$, $d(s)=n$, $d(v)=0$ for $v \neq s$, and $i=0$, and repeating the following three steps l times:

Step 1. (Update preflow.) Replace i by $i+1$. For $(v, t) \in E$, replace $f(v, t)$ by $\min\{c_{\lambda_i}(v, t), f(v, t)\}$. For $(s, v) \in E$ with $d(v) < n$, replace $f(s, v)$ by $\max\{c_{\lambda_i}(s, v), f(s, v)\}$.

Step 2. (Find maximum flow.) Apply the preflow algorithm to the network with arc capacities corresponding to λ_i, beginning with the current f and d. Let f and d be the resulting flow and final valid labeling.

Step 3. (Find minimum cut.) Redefine $d(v) = \min\{d_f(v, s)+n, d_f(v, t)\}$ for each $v \in V$. The cut $(X_i, \bar{X}_i)$ is then given by $X_i = \{v \mid d(v) \geqq n\}$.

The minimum cuts produced by the algorithm have a *nesting property* that was previously observed in the context of various applications by Eisner and Severance [9], Stone [43], and perhaps others. Megiddo [27] has also noted a similar property in a related problem. Here, the result follows directly from our algorithm.

LEMMA 2.4. *For a given on-line sequence of parameter values* $\lambda_1 < \lambda_2 < \cdots < \lambda_l$, *the parametric preflow algorithm correctly computes maximum flows* $f_1, f_2, \cdots, f_l$ *and minimum cuts* $(X_1, \bar{X}_1), (X_2, \bar{X}_2), \cdots, (X_l, \bar{X}_l)$ *such that* $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_l$.

Proof. The correctness of the algorithm is immediate. For any vertex v, the label $d(v)$ cannot decrease in Step 3, which implies that $d(v)$ never decreases during the running of the algorithm. This means that $X_1 \subseteq X_2 \subseteq \cdots \subseteq X_l$, which in turn implies that there can be at most $n-1$ distinct sets X_i. □

2.4. Analysis of the parametric preflow algorithm. In view of Lemma 2.4, Lemmas 2.1 and 2.2 hold without change for the parametric preflow algorithm. Furthermore, the time spent in Steps 1 and 3 is $O(m)$ per iteration, for a total of $O(lm)$ time. Thus the parametric preflow algorithm runs in $O((n+l)m)$ time plus $O(1)$ time per nonsaturating push.

The number of nonsaturating pushes depends on the order in which *push/relabel* steps are performed. We shall analyze three versions of the parametric preflow algorithm. For each, we show that the time bound for the nonparametric case extends to the parametric case with an increase of at most a constant factor. The proofs of the following three theorems are analogous to those given in [13].

We first analyze the generic version, in which *push/relabel* steps take place in any order.

THEOREM 2.5. *If the order of push/relabel steps is arbitrary, then the number of nonsaturating push steps is* $O(n^2(l+m))$. *The total running time of the parametric preflow algorithm is* $O((n^2(l+m))$, *or* $O(n^2m)$ *if* $l = O(n)$.

Proof. Let $\Phi = \sum\{d(v) \mid v \text{ is active}\}$ if some vertex is active, and $\Phi = 0$ otherwise. A nonsaturating *push* step decreases Φ by at least one. The function Φ is always in the range 0 to $2n^2$. Step 1 can increase Φ by at most $2n^2$, for a total over all iterations of Step 1 of $O(ln^2)$. A relabeling step increases Φ by the amount by which the label changes. Thus the total increase in Φ due to relabeling steps is $O(n^2)$. A saturating *push* step can increase Φ by at most $2n$. Thus the total increase in Φ due to such steps is $O(n^2m)$. These are the only ways in which Φ can increase. The total number of nonsaturating *push* steps is bounded by the total increase in Φ over the algorithm, which is $O(n^2m)$. □

Next we consider the *first-in first-out* (FIFO) version of the preflow algorithm, which solves the nonparametric problem in $O(n^3)$ time [13]. In this version, a queue

Q is used to select vertices for *push/relabel* steps. Initially Q is empty. At the beginning of Step 2 of the parametric preflow algorithm, every active vertex is appended to Q. The FIFO algorithm consists of repeating the following step until Q is empty:

Discharge. Remove the vertex v on the front of Q. Apply *push/relabel* steps to v until v is no longer active or v is relabeled. If a *push* from v to another vertex w makes w active, add w to the rear of Q.

THEOREM 2.6. *For the* FIFO *version of the parametric preflow algorithm, the number of nonsaturating push steps is* $O(n^3)$. *The total running time is* $O(n^3+ln^2)$, *or* $O(n^3)$ *if* $l=O(n)$.

Proof. We define *passes* over the queue Q as follows. The first pass during an iteration of Step 2 consists of the *discharge* steps applied to the vertices initially on Q. Each pass after the first in an iteration of Step 2 consists of the *discharge* steps applied to the vertices added to Q during the previous pass. There is at most one nonsaturating *push* step per vertex v per pass, since such a step reduces $e(v)$ to zero. We claim that the total number of passes is $O(n^2+ln)$, from which the theorem follows.

To establish the claim, we define $\Phi=\max\{d(v)\mid v\in Q\}$ if $Q\neq\emptyset$, and $\Phi=0$ if $Q=\emptyset$. Consider the effect on Φ of a pass over Q. If $v\in Q$ at the beginning of a pass and $d(v)=\Phi$, then $v\notin Q$ at the end of the pass unless a *relabel* step occurs during the pass. Thus, if Φ is the same at the end as at the beginning of the pass, some vertex label must have increased by at least one. If Φ increases over the pass, some $d(v)$ must increase by at least the amount of the increase in Φ. From the end of one iteration of Step 2 to the beginning of the next, Φ can increase by $O(n)$. Thus (i) the total number of passes in which Φ can increase or stay the same is $O(n^2+ln)$; (ii) the total number of passes in which Φ can decrease is at most the total increase in Φ between passes and during passes in which it increases, which is also $O(n^2+ln)$. We conclude that the total number of passes is at most $O(n^2+ln)$, verifying the claim and hence the theorem. □

A more elaborate version of the preflow algorithm [13] uses the dynamic tree data structure of Sleator and Tarjan [40], [41] to reduce the running time to $O(nm\log(n^2/m))$ if $l=O(n)$. The corresponding version of the parametric preflow algorithm also runs in $O(nm\log(n^2/m))$ time. It uses a queue Q for vertex selection, and it performs *discharge* steps exactly as does the FIFO algorithm, but in place of *push/relabel* steps it uses more complicated *tree-push/relabel* steps. A *tree-push/relabel step* can move an amount of flow excess through several arcs at once. Extending the analysis in [13] to the parametric case is straightforward. We shall merely summarize the results.

The dynamic tree algorithm uses a parameter k, the *maximum tree size*, which can be chosen freely in the range from 2 to n. An easy extension of the analysis in [13] shows that the parametric preflow algorithm runs in $O(nm\log k)$ time plus $O(\log k)$ time per addition of a vertex to Q. Furthermore, the number of additions of a vertex to Q is $O(nm)$ plus $O(n/k)$ per pass over Q, where passes are defined as in the proof of Theorem 2.6. The $O(n^2+ln)$ bound on the number of passes in the proof of Theorem 2.6 remains valid if the dynamic tree algorithm is used in place of the FIFO algorithm. Combining these estimates, we obtain an $O((nm+(n^3+ln^2)/k)\log k)$ bound on the total running time. Choosing $k=\max\{2, n^2/m\}$, we have the following theorem.

THEOREM 2.7. *The parametric preflow algorithm implemented using dynamic trees runs in* $O((n+l)m\log(n^2/m))$ *time. If* $l=O(n)$, *the time bound is* $O(nm\log(n^2/m))$.

2.5. Additional observations. We conclude this section with several observations about the parametric maximum flow problem and our algorithm for solving it. Our

first observation concerns variants of the parametric maximum flow problem. Our algorithm remains valid if the arc capacity functions are nonincreasing on arcs out of s and nondecreasing on arcs into t, and the values of the parameter λ are given in decreasing order. To see this, merely substitute $-\lambda$ for λ. The algorithm also applies if the arc capacity functions are nondecreasing on arcs out of s and nonincreasing on arcs into t, and the values of λ are given in decreasing order. In this case, reverse the directions of all the arcs, exchange the source and sink, and apply the original algorithm to this reversed network, which we shall denote by G^R. Each minimum cut $(\bar{X}, X)$ generated for G^R will correspond to a minimum cut $(X, \bar{X})$ in the original (nonreversed) network that will have the source side of minimum size instead of the sink side. Successively generated minimum cuts in G^R will correspond to cuts with successively smaller source sides in the original network.

If we are only interested in computing minimum cuts, there is a variant of the preflow algorithm that does less computation, although it has the same asymptotic time bound [13]. This variant, here called the *min-cut preflow algorithm*, computes a preflow of maximum value and a minimum cut, but not a maximum flow. It differs from the original algorithm in that a vertex v is considered to be active only if $e(v) > 0$ and $d(v) < n$. The algorithm terminates having computed a *maximum preflow*. (A preflow f is *maximum* if and only if for every vertex v, $v \neq t$ or $e(v) = 0$ or $d_f(v, t) < \infty$.) If this variant is used to compute minimum cuts, a maximum flow for a desired parameter value can be computed by beginning with the corresponding maximum preflow and converting it into a maximum flow using the original preflow algorithm. Most of the applications we shall consider only require the computation of a sequence of minimum cuts or even minimum cut values and not maximum flows. We shall refer to the variant of our parametric preflow algorithm that computes minimum cuts and maximum preflows as the *min-cut parametric algorithm.*

So far we have required all arc capacities to be nonnegative, but if we are only interested in computing minimum cuts, we can allow negative capacities on arcs out of s and on arcs into t. This is because there is a simple transformation that makes such arc capacities nonnegative without affecting minimum cuts [35]. For a given vertex v, suppose we add a constant $\Delta(v)$ to $c(s, v)$ and $c(v, t)$. Then the minimum cuts do not change since the capacity of every cut is increased by $\Delta(v)$.

By adding a suitably large $\Delta(v)$ to $c(s, v)$ and $c(v, t)$ for each v, we can make all the arc capacities positive. In the parametric problem, we can choose a new function Δ on the vertices of G for each new value λ_i of λ without affecting the $O((n+l)m \log (n^2/m))$ time bound for our algorithm. It suffices to choose

$$\Delta_{\lambda_1}(v) = \max \{0, -c_{\lambda_1}(s, v)\} + \max \{0, -c_{\lambda_1}(v, t)\},$$

$$\Delta_{\lambda_{i+1}}(v) = \Delta_{\lambda_i}(v) + c_{\lambda_i}(v, t) - c_{\lambda_{i+1}}(v, t) \quad \text{for } i \geqq 1.$$

With this choice, the transformed arc capacities are nondecreasing functions of λ on arcs leaving s and constant on arcs that enter t. Although the same effect could be obtained by adding a sufficiently large constant to the capacities of these arcs, the modification we have described has the additional advantage of keeping capacities as small as possible. In subsequent sections, when discussing minimum cut problems, we shall allow arbitrary capacities, positive or negative, on arcs leaving s and arcs entering t.

The minimum cuts corresponding to various parameter values have a *nesting property* that is a strengthening of Lemma 2.4. The following lemma is an extension of known results [10, pp. 13], [43] that we shall need in the next section. To prove the lemma, we shall use the min-cut preflow algorithm discussed above.

LEMMA 2.8. *Let* $(X, \bar{X})$ *be any minimum cut for* $\lambda = \lambda_1$, *and let* $(Y, \bar{Y})$ *be any minimum cut for* $\lambda = \lambda_2$ *such that* $\lambda_1 \leqq \lambda_2$. *Then,* $(X \cap Y, \bar{X} \cup \bar{Y})$ *is a minimum cut for* $\lambda = \lambda_1$ *and* $(X \cup Y, \bar{X} \cap \bar{Y})$ *is a minimum cut for* $\lambda = \lambda_2$.

Proof. Run the min-cut parametric algorithm for $\lambda = \lambda_1$, followed by $\lambda = \lambda_2$. At the beginning of the computation for $\lambda = \lambda_2$, all vertices $v \in X - \{s\}$ have $d(v) \geqq n$. Thus, after a maximum preflow is computed for $\lambda = \lambda_2$, all arcs (v, w) with $v \in X$, $w \in \bar{X}$ are saturated (their flow has not changed during the computation for $\lambda = \lambda_2$). Since $(Y, \bar{Y})$ is a minimum cut for $\lambda = \lambda_2$, all arcs (v, w) with $v \in Y$, $w \in \bar{Y}$ are saturated. Furthermore, if $v \in \bar{Y} - \{t\}$ then $e(v) = 0$, since the net flow across $(Y, \bar{Y})$ must be equal to the excess at t.

Now consider the cut $Z = (X \cup Y)$, $\bar{Z} = (\bar{X} \cap \bar{Y})$. Any arc (v, w) with $v \in Z$, $w \in \bar{Z}$ must be saturated. Since $v \in \bar{Z} - \{t\}$ implies $e(v) = 0$, the cut $(Z, \bar{Z})$ must have capacity $e(t)$, and hence it must be a minimum cut.

The proof for $(X \cap Y, \bar{X} \cup \bar{Y})$ is similar: proceed on G^R, and run the min-cut parametric algorithm for $\lambda = \lambda_2$, followed by $\lambda = \lambda_1$. □

A direct consequence of this lemma is the following corollary, which we shall need in the next section.

COROLLARY 2.9. *Let* $(X_1, \bar{X}_1)$ *be a minimum cut for* $\lambda = \lambda_1$, *let* $(X_2, \bar{X}_2)$ *be a minimum cut for* $\lambda = \lambda_2$ *such that* $X_1 \subseteq X_2$ *and* $\lambda_1 \leqq \lambda_2$, *and let* λ_3 *be such that* $\lambda_1 \leqq \lambda_3 \leqq \lambda_2$. *Then there is a cut* $(X_3, \bar{X}_3)$ *minimum for* λ_3 *such that* $X_1 \subseteq X_3 \subseteq X_2$.

Proof. Let $(X_3', \bar{X}_3')$ be any minimum cut for $\lambda = \lambda_3$. Take $X_3 = (X_3' \cup X_1) \cap X_2$, $\bar{X}_3 = V - X_3$, and apply Lemma 2.8 twice. □

Our last observation is that if the graph G is bipartite, the $O((n+l)m \log (n^2/m))$ time bound for computing parametric maximum flows can be improved slightly. Suppose $V = A \cup B$, $A \cap B \neq \emptyset$, and every arc in G has one vertex in A and one in B. Let $n_A = |A|$ and $n_B = |B|$ and suppose that $n_A \leqq n_B$. Then the time to compute maximum flows for l ordered values of λ can be reduced to $O(n_A m \log (n_A^2/m + 2))$ if $l = O(n_A)$. This requires modifying the preflow algorithm so that only vertices in A are active, and modifying the use of the dynamic tree data structure so that such a tree contains as many vertices in A as in B. The bound is slightly worse if $l = \omega(n_A)$. The details can be found in [42].

3. The min-cut capacity function of a parametric network. For a parametric network, we define the *min-cut capacity function* $\kappa(\lambda)$ to be the capacity of a minimum cut as a function of the parameter λ. We shall assume throughout this section that the arc capacities are *linear* functions of λ satisfying the conditions (i)-(iii) of § 2. It is well known [9], [43] and follows from the results of § 2 that under this assumption $\kappa(\lambda)$ is a piecewise-linear concave function with at most $n-2$ breakpoints. (By a *breakpoint* we mean a value of λ at which the slope of $\kappa(\lambda)$ changes.) The $n-1$ or fewer line segments forming the graph of $\kappa(\lambda)$ correspond to $n-1$ or fewer distinct cuts. We shall develop three algorithms for computing information about $\kappa(\lambda)$. The first computes the smallest (or equivalently the largest) breakpoint. The second computes a value of λ at which $\kappa(\lambda)$ is maximum. The third computes all the breakpoints. Each of these algorithms uses the algorithm of § 2 as a subroutine and runs in $O(nm \log (n^2/m))$ time. Although the algorithm for computing all breakpoints solves all three problems, we shall present all three algorithms since each is more complicated than the preceding one and since the resulting difference in constant factors may be important in practice.

We shall assume that the capacities $c_\lambda(s, v)$ and $c_\lambda(v, t)$ are given in the form $c_\lambda(s, v) = a_0(v) + \lambda a_1(v)$ and $c_\lambda(v, t) = b_0(v) - \lambda b_1(v)$, with arbitrary coefficients a_0, b_0 and nonnegative coefficients a_1, b_1. A minimum cut $(X_0, \bar{X}_0)$ for some

$\lambda = \lambda_0$ gives an equation for a line that contributes a line segment to the function $\kappa(\lambda)$ at $\lambda = \lambda_0$. This line is $L_{X_0}(\lambda) = \alpha_0 + \lambda\beta_0$, where $\alpha_0 = c_{\lambda_0}(X_0, \bar{X}_0) - \lambda_0\beta_0$ and $\beta_0 = \sum_{v \in \bar{X}_0} a_1(v) - \sum_{v \in X_0} b_1(v)$. (Recall from § 2 that $c_\lambda(X, \bar{X}) = \sum_{v \in X, w \in \bar{X}} c_\lambda(v, w)$. Also note that $a_1(v) = 0$ if $(s, v) \notin E$, and $b_1(v) = 0$ if $(v, t) \notin E$.)

3.1. Computing the smallest breakpoint of $\kappa(\lambda)$. To compute the smallest breakpoint of $\kappa(\lambda)$ we use an algorithm stated by Gusfield [18] for an application involving scheduling transmissions in a communication network, discussed in more detail in § 4.1.

The algorithm, an adaption of Newton's method, consists of the following two steps.

Step 0. Compute λ_1, λ_2 such that the smallest breakpoint λ_0 satisfies $\lambda_1 \leqq \lambda_0 \leqq \lambda_2$. Compute a cut $(X_1, \bar{X}_1)$ that is a minimum for λ_1. Go to Step 1.

Step 1. Compute a cut $(X_2, \bar{X}_2)$ that is a minimum for λ_2. If $L_{X_1}(\lambda_1) = L_{X_2}(\lambda_2)$, stop: λ_2 is the smallest breakpoint. Otherwise, replace λ_2 by the value of λ such that $L_{X_1}(\lambda) = L_{X_2}(\lambda)$ and repeat Step 1. (The appropriate value of λ is $(\alpha_2 - \alpha_1)/(\beta_1 - \beta_2)$.)

The values of λ_2 generated by this algorithm are strictly decreasing; thus the parametric preflow algorithm of § 2 applied to G^R performs all iterations of Step 1 in $O(nm \log (n^2/m))$ time, saving a factor of n over Gusfield's algorithm [18].

In Step 0, it suffices to select λ_1 sufficiently small so that for each vertex v such that (s, v) or (v, t) is of nonconstant capacity, $c_{\lambda_1}(s, v) + \sum_{u \in V - \{s,t\}} c(u, v) < c_{\lambda_1}(v, t)$. A simple computation shows that a suitable value of λ_1 is

$$\min_{v \in V - \{s,t\}} \left\{ \frac{b_0(v) - a_0(v) - \sum_{u \in V - \{s,t\}} c(u, v)}{a_1(v) + b_1(v)} \,\middle|\, a_1(v) + b_1(v) > 0 \right\} - 1. \tag{3.1}$$

Similarly, it suffices to select λ_2 sufficiently large so that for each vertex v such that (s, v) or (v, t) is of nonconstant capacity, $c_{\lambda_2}(v, t) + \sum_{w \in V - \{s,t\}} c(v, w) < c_{\lambda_2}(s, v)$. A suitable value of λ_2 is

$$\max_{v \in V - \{s,t\}} \left\{ \frac{b_0(v) - a_0(v) + \sum_{w \in V - \{s,t\}} c(v, w)}{a_1(v) + b_1(v)} \,\middle|\, a_1(v) + b_1(v) > 0 \right\} + 1. \tag{3.2}$$

Essentially the same algorithm can be used for computing the largest breakpoint; instead of successively decreasing λ_2 and using G^R, successively increase λ_1 and use G.

3.2. Finding a maximum of $\kappa(\lambda)$. Our algorithm for finding the value of λ that maximizes $\kappa(\lambda)$ is based on a simple method of iterative interval contraction for computing the maximum $f(\lambda^*)$ of a strictly concave and continuously differentiable function $f(\lambda)$ on a nonempty interval $[\lambda_1, \lambda_3]$ of the real line. The method is as follows. First, compute the function values and the tangents of $f(\lambda)$ at each end of the given interval. Second, compute the point $\lambda_2 \in [\lambda_1, \lambda_3]$ where the two tangent lines intersect, and also compute $f'(\lambda_2)$. Then, if $f'(\lambda_2) > 0$ replace λ_3 by λ_2 and repeat; if $f'(\lambda_2) > 0$ replace λ_1 by λ_2 and repeat; if $f'(\lambda_2) = 0$ accept λ_2 as the solution. Of course this algorithm need not terminate, but it will converge to the maximum.

The method is seldom used in this general setting because it is inferior to several other algorithms for one-dimensional maximization. But it can be specialized in the obvious way to handle the piecewise-linear concave function $\kappa(\lambda)$ efficiently. A maximum of $\kappa(\lambda)$ can be computed in as many function evaluations as there are linear segments that comprise $\kappa(\lambda)$, namely $n - 1$ or fewer. Using the notation introduced above, $\lambda_2 = (\alpha_3 - \alpha_1)/(\beta_1 - \beta_3)$ if the line segments of $\kappa(\lambda)$ at λ_1 and at λ_3 are distinct.

Otherwise, the search terminates with a line segment of zero slope and $\lambda^* = \lambda_1$ (or $\lambda^* = \lambda_3$). The algorithm will compute a maximum of $\kappa(\lambda)$ in $O(n^2 m \log(n^2/m))$ time since at most $n-1$ minimum cut problems must be solved.

The running time of this algorithm can be improved by partitioning the sequence of successive values of λ_2 into two subsequences, one increasing and the other decreasing. It is then possible to use two concurrent invocations of the parametric preflow algorithm: *invocation I* that starts with λ_1 and computes minimum cuts of G for an increasing sequence of λ values, and *invocation D* that starts with λ_3 and computes minimum cuts of G^R for a decreasing sequence of λ values. A new value λ_2 is a member of the increasing sequence if $\beta_2 > 0$, and a member of the decreasing sequence otherwise. The initial values of λ_1 and λ_3 must be such that all breakpoints lie in the interval $[\lambda_1, \lambda_3]$, a property that is assured by using (3.1) to give the initial value of λ_1 and (3.2) to give the initial value of λ_3.

The algorithm to compute a maximum of $\kappa(\lambda)$ consists of the following four steps.

Step 0. Compute the initial values λ_1 and λ_3 from (3.1) and (3.2). Start concurrent invocations (I and D) of the parametric preflow algorithm of §2: For $\lambda = \lambda_1$, invocation I computes a minimum cut $(X_1, \bar{X}_1)$ having $|X_1|$ maximum; for $\lambda = \lambda_3$, invocation D computes a minimum cut $(X_3, \bar{X}_3)$ having $|X_3|$ minimum.

Step 1. Compute $\lambda_2 = (\alpha_3 - \alpha_1)/(\beta_1 - \beta_3)$, pass λ_2 to both invocations I and D, and run them concurrently. If invocation I finds a minimum cut $(X_2^I, \bar{X}_2^I)$ first, suspend invocation D and go to Step 2 (the other case is symmetric). Compute $\beta_2 = \sum_{v \in \bar{X}_2} a_1(v) - \sum_{v \in X_2} b_1(v)$.

Step 2. If $\beta_2 = 0$, stop: $\lambda^* = \lambda_2$. Otherwise, if $\beta_2 > 0$, replace λ_1 by λ_2, back up invocation D to its state before it began processing λ_2, and go to Step 1. Otherwise, go to Step 3.

Step 3 ($\beta_2 < 0$). Finish running invocation D on λ_2. This produces a minimum cut $(X_2^D, \bar{X}_2^D)$, not necessarily the same as $(X_2^I, \bar{X}_2^I)$. If $\beta_2 \geqq 0$, stop: $\lambda^* = \lambda_2$. Otherwise, replace λ_3 by λ_2, back up invocation I to its state before processing λ_2, and go to Step 1.

Backing up invocation D or I as required in Steps 2 and 3 is merely a matter of restoring the appropriate flow and valid labeling, which takes $O(m)$ time. The total time spent during one iteration of Steps 1, 2, and 3 is proportional to the time spent in invocation I or D, whichever one is run to completion on λ_2 and not backed up. The total number of values of λ_2 processed is $O(n)$. Thus the total time is proportional to the time necessary to run the parametric preflow algorithm of § 2.3 twice, once on an increasing sequence of values and once on a decreasing sequence of values; i.e., $O(nm \log(n^2/m))$.

3.3. Finding all breakpoints of $\kappa(\lambda)$. In some applications it is necessary to produce all the line segments or breakpoints of $\kappa(\lambda)$, possibly along with the corresponding minimum cuts. To do this we extend the maximum-finding algorithm of the previous section. This algorithm uses iterative contraction of the interval $[\lambda_1, \lambda_3]$; it ignores breakpoints that lie in the discarded portion of the interval. We can find all the breakpoints by proceeding as in the algorithm of § 3.2 but using a divide-and-conquer strategy that recursively examines both of the subintervals $[\lambda_1, \lambda_2]$ and $[\lambda_2, \lambda_3]$ into which the current interval is split by the new value λ_2. This method was proposed by Eisner and Severance [9] for bipartite graphs in the context of a database record-segmentation problem (see § 4.4). Unfortunately, a straightforward implementation of this idea yields an $O(n^2 m \log(n^2/m))$-time algorithm. To obtain a better bound it is necessary to use two concurrent invocations of the parametric preflow algorithm, and

also use graph contraction so that recursive invocations of the method compute cuts on smaller and smaller graphs.

If G is a network and X is a set of vertices such that at most one of s and t is in X, we define $G(X)$, the *contraction of* G *by* X, to be the network formed by shrinking the vertices in X to a single vertex, eliminating loops, and combining multiple arcs by adding their capacities. The algorithm we present reports only the breakpoints of $\kappa(\lambda)$, although it computes cuts corresponding to the line segments of the graph of $\kappa(\lambda)$. If the actual cuts are needed, they can either be saved as the computation proceeds or computed in a postprocessing step using one application of the method in § 2.3.

Our algorithm uses a recursive procedure called *slice*. With each network G to which *slice* is applied, we associate four pieces of information: Two values of λ, denoted by λ_1 and λ_3, and two flows f_1 and f_3, such that f_1 is a maximum flow for λ_1, f_3 is a maximum flow for λ_3, the cut $(\{s\}, V-\{s\})$ is the unique minimum cut for λ_1, the cut $(V-\{t\}, \{t\})$ is the unique minimum cut for λ_3, and $\lambda_1 < \lambda_3$. The initial values for λ_1 and λ_3 are computed from (3.1) and (3.2) as before. The *breakpoint algorithm* consists of the following two steps.

Step 1. Compute λ_1 according to (3.1) and λ_3 according to (3.2). Compute a maximum flow f_1 and minimum cut $(X_1, \bar{X}_1)$ for λ_1 such that $|X_1|$ is maximum by applying the preflow algorithm to G. Compute a maximum flow f_3 and minimum cut $(X_3, \bar{X}_3)$ for λ_3 such that $|X_3|$ is minimum by applying the preflow algorithm to G^R. Form G' from G by shrinking the vertices in X_3 to a single vertex, shrinking the vertices in $\bar{X}_1$ to a single vertex, eliminating loops, and combining multiple arcs by adding their capacities. (Note that $X_3 \cap \bar{X}_1 = \emptyset$.)

Step 2. If G' contains at least three vertices, let f_1' and f_3' be the flows in G' corresponding to f_1 and f_3, respectively; perform *slice* $(G', \lambda_1, \lambda_3, f_1', f_3')$, where *slice* is defined as follows:

Procedure *slice* $(G, \lambda_1, \lambda_3, f_1, f_3)$.

Step S1. Let λ_2 be the value of λ such that $c_{\lambda_2}(\{s\}, V-\{s\}) = c_{\lambda_2}(V-\{t\}, \{t\})$. (This value will satisfy $\lambda_1 \leqq \lambda_2 \leqq \lambda_3$.)

Step S2. Run the preflow algorithm for the value λ_2 on G starting with the preflow f_1' formed by increasing f_1 on arcs (s, v) to saturate them and decreasing f_1 on arcs (v, t) to meet the capacity constraints. As an initial valid labeling, use $d(v) = \min\{d_{f_1'}(v, t), d_{f_1'}(v, s) + n\}$. Concurrently, run the preflow algorithm for the value λ_2 on G^R starting with the preflow f_3' formed by increasing f_3 on arcs (v, t) to saturate them and decreasing f_3 on arcs (s, v) to meet the capacity constraints. As an initial valid labeling, use $d(v) = \min\{d_{f_3'}(s, v), d_{f_3'}(t, v) + n\}$. Stop when one of the concurrent applications stops, having computed a maximum flow f_2. Suppose the preflow algorithm applied to G stops first. (The other case is symmetric.) Find the minimum cuts $(X_2, \bar{X}_2)$ and $(X_2', \bar{X}_2')$ for λ_2 such that $|X_2|$ is minimum and $|X_2'|$ is maximum. If $|X_2| > n/2$, complete the execution of the preflow algorithm on G^R and let f_2 be the resulting maximum flow.

Step S3. If $c_\lambda(X_2, \bar{X}_2) \neq c_\lambda(X_2', \bar{X}_2')$ for some λ, report λ_2 as a breakpoint.

Step S4. If $X_2 \neq \{s\}$, perform *slice* $(G(\bar{X}_2), \lambda_1, \lambda_2, f_1, f_2)$. If $\bar{X}_2' \neq \{t\}$, perform *slice* $(G(X_2'), \lambda_2, \lambda_3, f_2, f_3)$.

Remarks. Step 1 is an initialization step that guarantees that the graph G', on which *slice* is called, has unique minimum cuts $(\{s\}, V-\{s\})$ and $(V-\{t\}, \{t\})$ for λ_1

and λ_3, respectively. The flows f_1 and f_3 are needed as input parameters to *slice* to guarantee that the initial labeling is such that the time for a sequence of calls to *slice* is subject to the bound of § 2.

The correctness of this algorithm follows from Corollary 2.9. Note that the minimum cuts computed in Step S2 correspond to minimum cuts for λ_2 in the original network, with the correspondence obtained by expanding the contracted vertex sets. Since each vertex of G is in at most one of the two subproblems in Step S4, there are $O(n)$ invocations of *slice*.

3.4. Analysis of the breakpoint algorithm. Two ideas underlay the efficiency of the breakpoint algorithm. To explain them, we need to develop a framework for the analysis of the algorithm. We shall charge to an invocation of slice the time spent in the invocation, not including the time spent in nested invocations. The time charged to one invocation is then $O(m)$ plus the time spent running the preflow algorithm in Step S2. Summing $O(m)$ over all $O(n)$ invocations of *slice* gives a bound of $O(nm)$. It remains to estimate the time spent running the preflow algorithm.

In our analysis we shall denote by n_0 and m_0 the number of vertices and edges in the original (unshrunken) graph, and by n and m the number of vertices and edges in one of the shrunken graphs passed to *slice*. We use the dynamic tree version of the preflow algorithm, with the maximum tree size k chosen globally. Specifically, let $k_0 = \max\{2, n_0^2/m_0\}$. For an invocation of *slice* on a graph with n vertices and m edges, we let the maximum tree size for this subproblem be $k = \min\{n, k_0\}$. Then the running time of this invocation of the preflow algorithm is $O((nm + n^3/k)\log k) = O((nm + n^3/k_0)\log k_0)$.

The first idea contributing to the speed of the algorithm is that the results of § 2 allow us to bound the time of a sequence of preflow algorithm applications, not just a single one, by $O((nm + n^3/k_0)\log k_0)$. That is, if we charge this much time for an invocation of *slice*, we can regard certain of the nested invocations as being free. The second idea is that running the preflow algorithm concurrently on G and on G^R allows us to regard the larger of the nested invocations in Step S4 as being free, since the time spent on the larger subproblem invocation is no more than that spent on the smaller, which implies that the total time is at most twice the time spent on all the smaller subproblem invocations. This leads to a recurrence bounding the total running time for all nested invocations whose solution is $O((nm + n^3/k_0)\log k_0)$. This gives a total time bound for the breakpoint algorithm of $O(n_0 m_0 \log(n_0^2/m_0))$.

Consider an invocation of *slice* $G(\lambda_1, \lambda_3, f_1, f_3)$. Let $G_1 = G(\bar{X}_2)$ as computed in Step S4, and let $G_3 = G(X_2')$; let n_1, m_1 and n_2, m_2 be the numbers of vertices and arcs in G_1 and G_2, respectively. We regard this invocation of *slice* as being a continuation of the algorithm of § 2.3 applied to G, with λ_1 the most recently processed value of λ and f_1 the resulting maximum flow. Simultaneously, we regard the invocation as being a continuation of the algorithm of § 2 applied to G^R, with λ_3 the most recently processed value of λ and f_3 the resulting flow.

With this interpretation we can regard the preflow algorithm applications in Step S2 as being free, but if $|X_2| \leqq n/2$ we must account for new applications of the algorithm in § 2 on $G(\bar{X}_2)$ and $G^R(\bar{X}_2)$, and otherwise (i.e., $|\bar{X}_2'| \leqq n/2$) we must account for new applications of the algorithm in § 2 on $G(X_2')$ and $G^R(X_2')$. Thus we obtain the following bound on the time spent in invocations of the preflow algorithm. If G has n vertices and m arcs, the time spent in such invocations during the computation of $\kappa(\lambda)$ is at most $T(n, m) + O((nm + n^3/k_0)\log k_0)$, where $T(n, m)$ is defined recursively as follows:

$$T(n, m) = \begin{cases} 0 & \text{if } n \leqq 3, \\ \max \{T(n_1, m_1) + T(n_2, m_2) + O((n_1 m_1 + n_1^3/k_0) \log k_0): \\ \qquad n_1, n_2 \geqq 3;\ n_1 + n_2 \leqq n + 2; \\ \qquad n_1 \leqq n_2;\ m_1, m_2 \geqq 1;\ m_1 + m_2 \leqq m + 1\} & \text{if } n > 3. \end{cases}$$

Remark. In this analysis, the sequence of preflow algorithm invocations associated with a particular application of the algorithm of § 2.3 is on a sequence of successively smaller graphs, but the analysis in § 2.4 remains valid. In the definition of $T(n, m)$, the constraint $m_1 + m_2 \leqq m + 1$ allows for the existence of an arc (s, t), which will appear in both subproblems.

A straightforward proof by induction shows that $T(n, m) = O((nm + n^3/k_0) \log k_0)$. By setting $n = n_0$ and $m = m_0$, we obtain the following theorem.

THEOREM 3.1. *The breakpoint algorithm runs in $O(nm \log (n^2/m))$ time on a graph with n vertices and m edges.*

3.5. Additional observations. We conclude this section with two observations. First, as noted by Stone [43], a complete set of minimum cuts for all values of λ can be represented in $O(n)$ space: store with each vertex $v \notin \{s, t\}$ the breakpoint at which v moves from the sink side to the source side of a minimum cut, for a set of minimum cuts whose source sides are nested. The breakpoint algorithm can be augmented to compute this information without affecting its asymptotic time bound. Second, the time bound of the three algorithms in Sections 3.1–3.3 can be improved to $O(n_A m \log (n_A^2/m + 2))$ if G is bipartite and $\kappa(\lambda)$ has $O(n_A)$ breakpoints. Here n_A is the size of the smaller half of the bipartite partition of V. This bound follows using the bipartite variant of the preflow algorithm mentioned at the end of § 2.5.

4. Applications. In this section, we give a number of applications of the algorithms in §§ 2 and 3. For each application, we obtain an algorithm running in $O(nm \log (n^2/m))$ time, where n is the number of vertices and m is the number of arcs in the graph involved in the problem. For applications in which the graph is bipartite, the bound is $O(n_A m \log (n_A^2/m + 2))$, where n_A is the size of the smaller half of the bipartite partition of the vertex set. (When the latter bound is applicable, we shall state it within square brackets.) Depending on the application, our bound is a factor of from $\log n$ to n better than the best previously known bound. Our applications fall into four general categories: Flow-sharing problems, zero-one fractional programming problems, parametric zero-one polynomial programming problems, and miscellaneous applications.

4.1. Flow sharing. Consider a network with a set of sources $S = \{s_1, s_2, \cdots, s_k\}$ and a single sink t, in which we want to find a flow from the sources in S to t. We require flow conservation at vertices not in $S \cup \{t\}$. We can model this problem as an ordinary one-source, one-sink problem by adding a supersource s and an arc (s, s_i) of infinite capacity for each $i \in \{1, \cdots, k\}$. The resulting network can have many different maximum flows, with different utilizations of the various sources; we define the utilization u_i of source s_i to be the flow through the arc (s, s_i). The question arises of how to compare the quality of such flows. Suppose each source s_i has a positive weight w_i. Several figures of merit have been proposed, leading to the following optimization problems:

(i) *Perfect sharing.* Among flows with u_i/w_i equal for all $i \in \{1, \cdots, k\}$, find one that maximizes the flow value $e(t)$.

(ii) *Maximin sharing.* Among maximum flows, find one that maximizes the smallest u_i/w_i, $i \in \{1, \cdots, k\}$.

(iii) *Minimax sharing.* Among maximum flows, find one that minimizes the largest u_i/w_i, $i \in \{1, \cdots, k\}$.

(iv) *Optimal sharing.* Among maximum flows, find one that simultaneously maximizes the smallest u_i/w_i and minimizes the largest u_i/w_i, $i \in \{1, \cdots, k\}$.

(v) *Lexicographic sharing.* Among maximum flows, find one that lexicographically maximizes the k-component vector whose jth component is the jth smallest u_i/w_i, $i \in \{1, \cdots, k\}$.

Flow-sharing problems with one source and multiple sinks are completely analogous to the multiple-source case: merely exchange source and sinks and reverse the network. For the criteria (ii)-(v), we can even allow multiple sources and multiple sinks, and simultaneously optimize one criterion for the sources and a possibly different criterion for the sinks. This is because each of problems (ii)-(v) calls for a maximum flow, and a multiple-source, multiple-sink problem can be decomposed into a multiple-source, one-sink problem and a one-source, multiple-sink problem, by finding a minimum cut of all sources from all sinks, contracting all vertices on the sink side to give a one-sink problem, and separately contracting all vertices on the source side to give a one-source problem. This observation is due to Megiddo [27].

Perfect sharing arises in a network transmission problem studied by Itai and Rodeh [22] and Gusfield [18] and in a network vulnerability model proposed by Cunningham [5]. We discuss these models below. Brown studied maximin sharing [3], Ichimori, Ishii, and Nishida [21] formulated minimax and optimal sharing, and Megiddo [27], [28] studied lexicographic sharing. Motivation for these problems is provided by the following kind of example, which gives rise to a multiple-sink problem. During a famine, relief agencies supplying food to the stricken areas want to distribute their available food supplies so that each person receives a fair share. The weight associated with each sink (famine area) is the population in that area, possibly adjusted for differences in food needs between adults and children, and other factors. A perfect sharing solution gives every person in every famine area the same amount of food, but it may be too restrictive since it need not allocate all the available and transportable food supply. A better solution will be provided by solving one of the problems (ii)-(v). There are analogous industrial interpretations of this model.

We shall show that all five flow-sharing problems can be solved in $O(nm \log (n^2/m))$ time using the algorithms of §§ 2 and 3. The lexicographic sharing problem requires computing all the breakpoints of a min-cut capacity function by the algorithm of § 3.3. The other four problems are easier, and can be solved by the algorithm for finding the smallest (or largest) breakpoint given in § 3.1. Our tool for solving all five problems is the following parametric formulation: for each $s_i \in S$, let arc (s, s_i) have capacity $w_i\lambda$, where λ is a real-valued parameter. Since all arc capacities are nonnegative, the range of interest of λ is $[0, \infty)$. There are at most k breakpoints of the min-cut capacity function $\kappa(\lambda)$, one per source s_i.

Perfect sharing. Find the smallest breakpoint λ_s of $\kappa(\lambda)$. Any maximum flow for λ_s solves the perfect sharing problem. This was observed by Gusfield [18] in the context of the network transmission-scheduling problem described below. Another application will arise in § 4.2.

Scheduling transmissions. Itai and Rodeh state the following problem of scheduling transmissions in a "circuit-switched" communication network represented by a directed graph $G = (V, E)$ with fixed positive arc capacities. The capacity $c(v, w)$ is the effective transmission rate of the communication channel (v, w) in the direction from v to w,

say in bits per second. A central vertex (sink) $t \in V$ receives all traffic that originates at a subset of vertices $S \subseteq V - \{t\}$ called *emitters* (sources). Each emitter $s_i \in S$ has $w_i > 0$ bits of information that it wishes to send to t. We assume that G has paths from each $s_i \in S$ to t. The communication protocol allows the sharing of arc capacities by several paths, but it requires that at least one path from s_i to t be established before transmission from s_i can begin. Clearly, if transmissions are scheduled from each emitter, one at a time, the entire task can be completed in $T' = \sum_{s_i \in S} w_i / c(X_i, \bar{X}_i)$ seconds, where $(X_i, \bar{X}_i)$ is a minimum cut separating s_i and t. But since arc capacities can be shared, it may be possible to obtain a lower value for T. The objective is to minimize the time T within which all transmissions can be completed.

To see that the problem is a perfect sharing multiple-source problem, let $\lambda = 1/T$, where T is in seconds, and assign a capacity of $w_i \lambda$ bits per second to each arc (s, s_i) from the supersource s to an emitter $s_i \in S$. Once λ_s and the corresponding maximum flow have been computed by the algorithm above, the actual transmission schedule can be constructed from the flow in $O(m)$ time as described in [22]. Itai and Rodeh proposed two algorithms for this problem, with running times of $O(kn^2m)$ and $O(k^2nm \log n)$. These are modifications of known maximum-flow algorithms. In comparison, our algorithm runs in $O(nm \log (n^2/m))$ time.

Maximin sharing. Find the largest breakpoint λ_l of $\kappa(\lambda)$. Any maximum flow for λ_l solves the maximin sharing problem.

Minimax sharing. Find the smallest breakpoint λ_s of $\kappa(\lambda)$. Find a maximum flow for λ_s. Construct a residual network in which each arc (v, w) with $s \notin \{v, w\}$ has capacity $c(v, w) - f(v, w)$, each arc (s, s_i) has infinite capacity, and each arc (s_i, s) has zero capacity. Find a maximum flow f' in the residual network. The flow $f + f'$ is a minimax flow in the original network.

Optimal sharing. Find the smallest breakpoint λ_s and the largest breakpoint λ_l of $\kappa(\lambda)$. Find a maximum flow f for λ_s. Construct a residual network in which each arc (v, w) with $s \notin \{v, w\}$ has capacity $c(v, w) - f(v, w)$, each arc (s, s_i) has capacity $w_i(\lambda_l - \lambda_s)$, and each arc (s_i, s) has zero capacity. Find a maximum flow f' in the residual network. The flow $f + f'$ is an optimal flow in the original network.

Lexicographic sharing. Find all the breakpoints of $\kappa(\lambda)$. For each source s_i, let λ_i be the breakpoint at which s_i moves from the sink side to the source side of a minimum cut. For each arc (s, s_i) define its capacity to be $w_i \lambda_i$. Find a maximum flow f with these upper bounds on the capacities of the arcs out of s. Flow f is a lexicographic flow, and hence an optimal flow.

The correctness of the first four algorithms above is easy to verify. We shall prove the correctness of the algorithm for the lexicographic sharing problem. Renumber the sources if necessary so that $\lambda_1 \leqq \lambda_2 \leqq \cdots \lambda_k$, and let G_∞ denote the parametric network with $\lambda = \infty$.

THEOREM 4.1. *On G_∞ there is a maximum flow f such that $f(s, s_i) = w_i \lambda_i$ for all i. Such a flow is a lexicographic flow.*

Proof. Let $i_1, i_2, \cdots, i_{l-1}$ be the values of i such that $\lambda_{i_j} < \lambda_{i_j+1}$. Let $i_0 = 0$ and $i_l = k$. Then $\lambda_{i_1}, \lambda_{i_2}, \cdots, \lambda_{i_l}$ are the distinct breakpoints in increasing order. Let $\{s\} = X_0 \subset X_1 \subset X_2 \subset \cdots \subset X_l$ be the sets such that $(X_j, \bar{X}_j)$ for $1 \leqq j \leqq l$ is the minimum cut with the smallest sink side for $\lambda = \lambda_{i_j}$. Then $s_{i_{j-1}+1}, s_{i_{j-1}+2}, \cdots, s_{i_j} \in X_j - X_{j-1}$. For $1 \leqq j \leqq l$, the cut $(X_{j-1}, \bar{X}_{j-1})$ is a minimum cut for $\lambda = \lambda_{i_j}$ as well, specifically the one with the smallest source side. Thus, $c_{\lambda_{i_j}}(X_{j-1}, \bar{X}_{j-1}) = c_{\lambda_{i_j}}(X_j, \bar{X}_j)$. It follows by induction on j that for $1 \leqq j \leqq l$,

$$c_{\lambda_{i_j}}(X_j, \bar{X}_j) = \sum_{i=1}^{i_j} w_i \lambda_i + \sum_{i=i_j+1}^{k} w_i \lambda_{i_j}, \tag{4.1}$$

which implies that

$$c_\infty(X_j - \{s\}, \bar{X}_j) = \sum_{i=1}^{i_j} w_i \lambda_{i_j}. \tag{4.2}$$

Equation (4.2) implies that any flow for G_∞ such that $f(s, s_i) = w_i\lambda_i$ for all i must be a maximum flow (choose $j = l$ in (4.2)). It must also be a lexicographic flow, since for all j, any flow that has $f(s, s_i) \geqq w_i\lambda_i$ for $1 \leqq i \leqq i_{j-1}$ must either have $f(s, s_i) = w_i\lambda_i$ for $1 \leqq i \leqq i_j$ or have some $i \in \{i_{j-1}+1, \cdots, i_j\}$, such that $f(s, s_i) < w_i\lambda_i$.

It remains to show that G_∞ admits a flow f with $f(s, s_i) = w_i\lambda_i$. Consider running the min-cut parametric preflow algorithm presented in § 2.5 on the parametric network, for the successive λ values $\lambda_{i_1}, \lambda_{i_2}, \cdots, \lambda_{i_l}$. Let $f_1, f_2, \cdots, f_l$ be the successive maximum preflows generated by the algorithm. When the min-cut preflow algorithm is restarted with a new value λ_{i_j} of λ, the flow on each arc (s, s_i) with $i \in \{i_{j-1}+1, \cdots, k\}$ is first increased from $w_i\lambda_{i_{j-1}}$ to $w_i\lambda_{i_j}$. All of this new flow successfully reaches the sink t, because of (4.1) and the fact that $(X_j, \bar{X}_j)$ is a minimum cut for $\lambda = \lambda_{i_j}$. It follows by induction on j that f_j is a flow and that $f_j(s, s_i) = w_i\lambda_i$ for $1 \leqq i \leqq i_j$. In particular, f_l is the desired flow. □

4.2. Fractional programming applications. Another class of problems that can be solved by the parametric preflow algorithm of § 2.3 arises from various discrete and network optimization problems with fractional objectives. In general, the *fractional programming* problem is defined as

$$\lambda(x^*) = \max_{x \in S} \{\lambda(x) = f(x)/g(x)\}, \tag{4.3}$$

where $f(x)$ and $g(x)$ are real-valued functions on a subset S of R^n, and $g(x) > 0$ for all $x \in S$. Isbell and Marlow [23] proposed an elegant solution method for the important case of linear f and g, but their approach has been extended to nonlinear problems (see, e.g., Dinkelbach [7]), and more recently to several classes of combinatorial problems (see, e.g., Picard and Queyranne [33], [34], Padberg and Wolsey [31], and Cunningham [5]).

A problem that is intimately related to (4.3) is

$$z(x^*, \lambda) = \max_{x \in S} \{z(x, \lambda) = f(x) - \lambda g(x)\}, \tag{4.4}$$

where λ is a real-valued constant. These two problems are related in the sense that x^* solves (4.3) if and only if (x^*, λ^*) solves (4.4) for $\lambda = \lambda^* = \lambda(x^*)$ giving the value $z(x^*, \lambda^*) = 0$. Isbell and Marlow's algorithm generates a sequence of solutions until this condition is met. We state their algorithm below in a form useful for our purposes (see, e.g., Gondran and Minoux [14, pp. 636-641]):

ALGORITHM FP.

Step 0. Select some $x^0 \in S$. Compute $\lambda_0 = f(x^0)/g(x^0)$. Set $k = 0$.

Step 1. Compute x^{k+1}, solving the problem (4.4): $z(x^{k+1}, \lambda_k) = \max_{x \in S} z(x, \lambda_k) = f(x) - \lambda_k g(x)$.

Step 2. If $z(x^{k+1}, \lambda_k) = 0$, stop: $x^* = x^k$. Otherwise, let $\lambda_{k+1} = f(x^{k+1})/g(x^{k+1})$, replace k by $k+1$ and go to Step 1.

THEOREM 4.2. *Algorithm* FP *is correct. The sequence of values* $\{\lambda_k\}$ *generated by the algorithm is increasing.*

Proof. For any particular k, $z(x^{k+1}, \lambda_k)$ is nonnegative in Step 1, since $z(x^{k+1}, \lambda_k) \geqq z(x^k, \lambda_k) = 0$. If $z(x^{k+1}, \lambda_k) = 0$, the algorithm halts with x^k, which solves (4.4) for $\lambda = \lambda_k$, and hence solves (4.3). The algorithm continues only if $z(x^{k+1}, \lambda_k) > 0$; i.e., $f(x^{k+1}) - \lambda_k g(x^{k+1}) > 0$, which implies $\lambda_{k+1} = f(x^{k+1})/g(x^{k+1}) > \lambda_k$. □

If maximization is replaced by minimization in problem (4.3), it suffices to replace maximization by minimization in (4.4) and use the same algorithm. In this case all values of $z(x^{k+1}, \lambda_k)$, except the last one, are less than zero, and a decreasing sequence $\{\lambda_k\}$ is generated. Another important observation is that in Step 1 the maximization (4.4) can be taken over a larger set $S' \supset S$, provided that $z(x, \lambda_k) \leqq 0$ for all $x \in S' - S$. In the minimization problem, the corresponding requirement is $z(x, \lambda_k) \geqq 0$ for all $x \in S' - S$. Several of our applications make use of this extension.

The following lemma can be used to bound the number of iterations of Algorithm FP in some situations.

LEMMA 4.3. *$g(x^{k+1}) < g(x^k)$ for $k \geqq 1$.*

Proof. Consider iterations $k-1$ and k of Algorithm FP, and assume $\lambda(x^k) < \lambda(x^*)$. In iteration $k-1$ we have $z(x^k, \lambda_{k-1}) > 0$ and $\lambda_k = f(x^k)/g(x^k)$. In iteration k we have

$$\begin{aligned} 0 < z(x^{k+1}, \lambda_k) &= f(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &= f(x^{k+1}) - \lambda_{k-1} g(x^{k+1}) + \lambda_{k-1} g(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &\leqq f(x^k) - \lambda_{k-1} g(x^k) + \lambda_{k-1} g(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &= \lambda_k g(x^k) - \lambda_{k-1} g(x^k) + \lambda_{k-1} g(x^{k+1}) - \lambda_k g(x^{k+1}) \\ &= (g(x^k) - g(x^{k+1}))(\lambda_k - \lambda_{k-1}), \end{aligned}$$

which implies that $g(x^k) > g(x^{k+1})$ since $\lambda_k > \lambda_{k-1}$. The inequality "$\leqq$" above follows from $z(x^{k+1}, \lambda_{k-1}) \leqq z(x^k, \lambda_{k-1})$ since x^k maximizes $z(x, \lambda_{k-1})$. □

The efficiency of Algorithm FP depends on the number of times problem (4.4) has to be solved, and on the time spent solving it. For continuous functions f and g defined on a nonempty compact set S, Schaible [38] has shown that the decreasing sequence $\{g(x^k)\}$ for $k \geqq 1$ approaches $g(x^*)$ linearly, and the increasing sequence $\{\lambda_k\}$ approaches λ^* superlinearly. Nevertheless, (4.4) may be as hard to solve as the original fractional problem unless some assumptions are made about f, g, and S. Fortunately, even the most restrictive assumptions find relevant applications in practice. For instance, if f and g are linear and S is polyhedral (the case in [23]), the algorithm consists of solving a finite number of linear programs (4.4) whose solution is implemented by *cost-parametric programming* on intervals $[\lambda_k, \lambda_{k+1}]$, for successive k. This can be specialized to network simplex parametric programming by using the primitives described by Grigoriadis [15]. If f is a negative semidefinite quadratic form and g is linear, the sequence of concave quadratic programs defined by (4.4) can be handled by the parametric algorithm of Grigoriadis and Ritter [16]. If f and g are negative- and positive-definite quadratic forms, respectively, Ritter's parametric quadratic programming method [37] can be used. Approaches for more general nonlinear problems are analyzed in [7] and [38].

If S is nonempty and finite, f is real-valued, and g is positive, integer-valued, and bounded above by some integer $p > 0$, Lemma 4.3 implies that Algorithm FP will terminate in $p+1$ or fewer iterations. This observation has been used in various applications where $g(x)$ is a set function, for which usually $p = O(n)$. Such is the case whether (4.3) is a maximization or a minimization problem.

We shall now describe a number of applications of the generic Algorithm FP. In each case the sequence of problems (4.4) that arises can be handled by our parametric preflow algorithm or its min-cut variant described in § 2.5.

Strength of a directed network. This is an application due to Cunningham [5, § 6]. Let $G = (V, E)$ be a given directed graph with n vertices, m arcs, nonnegative arc weights and nonnegative vertex weights, and a distinguished vertex $s \in V$. We assume that every $v \in V - \{s\}$ is reachable from s in G. The arc weight $c(v, w)$ represents the

cost required to "destroy" the arc $(v, w) \in E$. The node weight d_v is the "value" attributed to having v reachable from s. Destroying a set of edges $A \subseteq E$ (at a total cost of $f(A) = \sum_{(v,w)\in A} c(v, w)$) may cause some subset of vertices $V_A \subseteq V - \{s\}$ to become unreachable from s, resulting in a loss of $g(A) = \sum_{v\in V_A} d_v$ in total value. The ratio $f(A)/g(A)$ is the cost per unit reduction in value. Cunningham defines the *strength of the network* to be the minimum of this ratio taken over all subsets $A \subseteq E$ whose removal reduces the value of the network, i.e., such that $g(A) > 0$. This is a problem of the form (4.3):

$$\lambda(A^*) = \min_{A\subseteq E, g(A)>0} \{\lambda(A) = f(A)/g(A)\},$$

which leads to a sequence of problems (4.4) that Cunningham calls *attack problems*:

$$z(A^{k+1}, \lambda_k) = \max_{A\subseteq E} \{z(A, \lambda_k) = f(A) - \lambda_k g(A)\}.$$

Each such problem amounts to finding a minimum cut in an expanded network formed by adding to G a sink t and an arc (v, t) with capacity $\lambda_k d_v$ for each $v \in V - \{s\}$. If we solve the strength problem using Algorithm FP and use the algorithm of § 2.3 to compute minimum cuts for the generated parameter values, we obtain an algorithm running in $O(nm \log (n^2/m))$ time; as Cunningham notes, there can be only $O(n)$ iterations of Step 1. Alternatively, we can make use of his observation that (4.5) is zero if and only if there is flow in the expanded network such that $f(v, t) = \lambda_k d_v$ for each $v \in V$. Equivalently, $\lambda(A^*)$ is the largest value of λ for which the minimum cut is $(V, \{t\})$. That is, the strength problem is a perfect sharing problem, and it can be solved in $O(nm \log (n^2/m))$ time as described in § 4.1. Either method improves over Cunningham's method, which solves $O(n)$ minimum cut problems without making use of their similarity.

Zero-one fractional programming. An important subclass of (4.3) is the problem for which $f(x) \geqq 0$ and $g(x) > 0$ are given polynomials defined for all x in $S = \{0, 1\}^n - \{0\}^n$ as follows:

$$f(x) = \sum_{P\in A} a_P \prod_{i\in P} x_i + \sum_{i=1}^{n} a_i x_i, \tag{4.5}$$

$$g(x) = \sum_{Q\in B} b_Q \prod_{i\in Q} x_i + \sum_{i=1}^{n} b_i x_i. \tag{4.6}$$

The sets A and B are given collections of nonempty nonsingleton subsets of $\{1, \cdots, n\}$, $a_P \geqq 0$ for each $P \in A$, and $b_Q \leqq 0$ for each $Q \in B$. Since $f(x) \geqq 0$ and $g(x) > 0$ for all $x \in S$, we have $a_i \geqq 0$ and $b_i > 0$ for each $i \in \{1, \cdots, n\}$. This problem was studied by Picard and Queyranne [33]. For ease in stating time bounds we assume $n = O(|A| + |B|)$.

Algorithm FP leads to a sequence of problems of the form (4.4) for increasing values $\lambda_k \geqq 0$ of λ. Each such problem is an instance of the *selection* or *provisioning* problem, characterized by Rhys [36] and Balinski [2] as a minimum cut problem on a bipartite graph.

The entire sequence of these problems can be handled as a parametric minimum cut problem of the kind studied in § 2. We give two different formulations, one of which works for the special case of $B = \emptyset$ (i.e., $g(x)$ contains no nonlinear terms) and the other of which works for the general case. All the subsequent applications we consider fall into the case $B = \emptyset$.

If $B = \emptyset$, we define a bipartite network G whose vertex set contains one vertex for each set $P \in A$, one vertex for each $i \in \{1, \cdots, n\}$, and two additional vertices, a source s and a sink t. There is an arc (s, v) of capacity a_P for each vertex v corresponding

to a set $P \in A$, an arc (i, t) of capacity $\lambda b_i - a_i$ for every $i \in \{1, \cdots, n\}$, and an arc (v, i) of infinite capacity for every vertex corresponding to a set $P \in A$ that has i as one of its elements. Observe that the capacities of all arcs into t are nondecreasing functions of λ, and those of all other arcs are constant. The parametric preflow algorithm operates on G^R instead of G. For a given value of λ, a minimum cut $(X, \bar{X})$ in G corresponds to a solution x to (4.4) defined by $x_i = 1$ if $i \in X$, $x_i = 0$ if $i \in \bar{X}$.

In the general case ($B \neq \emptyset$), it is convenient to assume $f(x) > 0$ for some x (otherwise the solution to (4.3) is $\lambda(x) = 0$, attained for any x) and that Algorithm FP starts with an x^0 such that $\lambda_0 > 0$. Then, the entire sequence $\{\lambda_k\}$ is strictly positive. To solve (4.4) we rewrite it as follows:

$$z(x^*, \lambda) = \lambda \max_{x \in S} (f(x)/\lambda - g(x)).$$

We define the network G to have a vertex set consisting of one vertex for each set $P \in A$, one vertex for each set $Q \in B$, one vertex for each $i \in \{1, \cdots, n\}$, and a source s and a sink t. There is an arc (s, v) of capacity a_P/λ for each v corresponding to a set $P \in A$, an arc (s, v) of capacity $-b_Q$ for each v corresponding to a set $Q \in B$, an arc (v, i) of infinite capacity for each vertex v corresponding to a set $P \in A$ or $Q \in B$ that has i as one of its elements, and an arc (i, t) of capacity $b_i - a_i/\lambda$ for each $i \in \{1, \cdots, n\}$. The capacities of arcs out of s are nonincreasing functions of λ and those of arcs into t are nondecreasing functions of λ; the parametric preflow algorithm operates on G^R. Minimum cuts in G correspond to solutions exactly as described above.

Remark. This formulation differs from that in [33] because of the division by λ. The formulation of [33] gives a parametric minimum cut problem in which the capacities of arcs out of the source and of arcs into the sink are nondecreasing functions of λ, to which the results of § 2 do not apply.

The following analysis is valid for both of the above two cases. The nesting property of minimum cuts (Lemma 2.4) implies that the number of iterations of Step 1 of Algorithm FP is $O(n)$, a fact also observed by Picard and Queyranne [33]. To state time bounds, let us denote by n' and m' the number of vertices and edges, respectively, in G; $n' = n + |A| + |B| + 2$ and $m' = n + |A| + |B| + \sum_{P \in A} |P| + \sum_{Q \in B} |Q|$. Algorithm FP, in combination with the parametric preflow algorithm of § 2.3, yields a time bound of $O(n'm' \log (n'^2/m))$ [or $O(nm' \log (n^2/m' + 2))$], improving over the algorithms of Picard and Queyranne [33] and Gusfield, Martel, and Fernandez-Baca [20].

Maximum-ratio closure problem. This problem was considered by Picard and Queyranne [34] and independently by Lawler [25], who only considered acyclic graphs (see the next application). The problem can be solved by a straightforward application of Algorithm FP. Each problem in the sequence of problems (4.4) is a maximum-weight closure problem. The *maximum-weight closure problem* (Picard [32]) is the generalization to nonbipartite graphs of the selection or provisioning problem [2], [36] mentioned above.

These closure problems are defined formally as follows. Let $G = (V, E)$ be a directed graph with vertex weights a_v of arbitrary sign. A subset $U \subseteq V$ is a *closure* in G if for each arc $(v, w) \in E$ with $v \in U$ we also have $w \in U$. A closure $U^* \subseteq V$ is of *maximum weight* if the sum of its vertex weights is maximum among all closures in G. To compute a maximum-weight closure, construct the graph G^* from G as follows. Add a source s and a sink t to G. Create an arc (s, v) of capacity a_v and an arc (v, t) of zero capacity for each $v \in V$. Assign infinite capacity to all arcs in E. A minimum cut $(X, \bar{X})$ of G^* gives the desired closure $U^* = X - \{s\}$.

Now let $a_v \geqq 0$ and $b_v > 0$ be given weights on the vertices of $G = (V, E)$. The *maximum-ratio closure* problem is to find a closure U^* that maximizes the ratio $a(U)/b(U)$ over all nonempty closures $U \subseteq V$. To compute a maximum-ratio closure, Picard and Queyranne [34] suggest the use of Algorithm FP. This requires the solution of a sequence of $O(n)$ maximum-weight closure problems, each of which is a minimum-cut problem. Thus an $O(n^2 m \log (n^2/m))$-time algorithm results. Lawler's algorithm uses binary search and runs in $O(knm \log (n^2/m))$ time, where $k = \log (\max \{n, a_{\max}, b_{\max}\})$, assuming integer weights.

We can solve the entire sequence of these minimum-cut problems by the parametric preflow algorithm of § 2.3 as follows. Modify G^* so that for each vertex $v \in V$ there is an arc (s, v) of capacity $a_v - \lambda b_v$ and an arc (v, t) of capacity zero. All other arcs have infinite capacity. We start with $U^0 = V \cup \{s\}$; or, equivalently, with a sufficiently small value of λ so that the minimum cut is $(\{s\} \cup V, \{t\})$. Such a value is $\lambda_0 = \min_i a_i/b_i$. The capacities of arcs out of the source are nonincreasing functions of the parameter, and the parameter values are given on-line in increasing order. The parametric preflow algorithm operates on $(G^*)^R$ and runs in $O(nm \log (n^2/m))$ time, improving the bound of Picard and Queyranne by a factor of n and that of Lawler by a factor of k.

Remark. Negative arc capacities in the various minimum-cut problems can be made nonnegative using the transformation suggested in § 2.5. In the minimum-ratio closure problem, it suffices to assign a capacity of $\max \{0, a_v - \lambda b_v\}$ to each arc (s, v) and a capacity of $\max \{0, \lambda b_v - a_v\}$ to each arc (v, t).

A job-sequencing application. Lawler [25] applied his algorithm to a problem studied by Sidney [39] and others: there are n jobs to be scheduled for processing on a single machine subject to a partial order given as an acyclic graph $G = (V, E)$, where V is the set of jobs. Each job v has a processing time a_v and a "weight" $b_v > 0$ that describes its importance or some measure of profit. Let the completion time of job v as determined by a given feasible sequence be C_v. It is required to find a sequence that minimizes $\sum_{v \in V} b_v C_v$. This problem is NP-complete for an arbitrary partial order even when all $a_v = 1$ or all $b_v = 1$ [25]. Sidney offered the following decomposition procedure. First find a maximum-ratio closure U_1 such that $|U_1|$ is minimum. Remove the subgraph induced by U_1 from G, find a maximum-ratio closure U_2 of the reduced graph, and repeat this process until the entire vertex set is partitioned. Sidney and Lawler call closures *initial sets* of V. Once such a decomposition is found, an optimal schedule can be computed by finding an optimal schedule for each closure, for example, by a branch-and-bound method, and then concatenating the solutions. The algorithm described above can be used to find each closure. The overall time bound depends on the size of each closure. (Our algorithm will give closures of minimum cardinality, since the algorithm is applied to the graph $(G^*)^R$; see § 2.5.)

Maximum-density subgraph. A special case of the fractional programming problem (4.3) is that of finding a nonempty subgraph of maximum density in an undirected graph $G = (V, E)$ with n vertices and m edges. The *density* of a subgraph of G induced by a subset of vertices $V' \subseteq V$ is the number of its edges divided by the number of its vertices. For this application, (4.5) and (4.6) have the simpler forms $f(x) = \frac{1}{2} xAx$ and $g(x) = ex$, where e is the vector of all ones, A is the vertex-vertex incidence matrix of G, and $x_i = 1$ if vertex $i \in V'$, and $x_i = 0$ otherwise. Algorithm FP, which yields x^* and the maximum density λ^*, can be used to compute a maximum-density subgraph of G. It is not necessary to construct a bipartite network and solve minimum-cut problems on it. We can merely modify G by specializing the construction of [35]. Replace each edge of G by two oppositely directed arcs of unit capacity, add a source s and a sink t, and create an arc (s, v) of capacity $\delta_v - \lambda$ and an arc (v, t) of zero capacity for each

$v \in V$, where δ_v is one-half the degree of vertex v in G. We can also allow weights on the edges and vertices. The resulting algorithm runs in $O(nm \log (n^2/m))$ time. This bound is better than that of Picard and Queyranne [34] and of Goldberg [12] by a factor of $\log n$; Goldberg's bound is valid only for the unweighted version of the problem.

The integer $\lceil \lambda^* \rceil$ is known as the *pseudoarboricity* of G: the minimum number of edge-disjoint pseudoforests into which G can be decomposed. (A *pseudoforest* is a subgraph each of whose connected components is a tree or a tree plus an edge.) If in the above network the capacities of arcs (s, v) and (v, t) for each $v \in V$ are increased by m, the minimum cut and maximum flow for $\lambda = \lambda^*$ can be used to construct a decomposition of G into $\lceil \lambda^* \rceil$ pseudoforests by the procedure suggested in [33].

4.3. Parametric zero-one polynomial functions. We consider the problem of computing a minimum of the function

$$z(\lambda) = \max_{x \in S} \{f(x) - \lambda(dx - b)\}, \tag{4.7}$$

where $S = \{0, 1\}^n$, $f(x)$ is a polynomial in zero-one variables defined by (4.5), and $d_i > 0$, $i \in \{1, \cdots, n\}$, such that $\sum_i d_i > b > 0$.

The function $z(\lambda)$ differs from the corresponding function (4.4) that arises in the zero-one fractional programming application of § 4.2 because of the term λb in (4.7). The function $z(\lambda)$ is piecewise linear and convex, and it has at most $n-1$ linear segments and $n-2$ breakpoints. The network formulation of (4.7) is as defined for the zero-one fractional programming application. The breakpoints of $z(\lambda)$ coincide with those of the min-cut capacity function $\kappa(\lambda)$ for this network. In general, no minimum of $z(\lambda)$ coincides with a maximum of $\kappa(\lambda)$. To compute a minimum of $z(\lambda)$, we can use the algorithm of § 3.2 for finding a maximum of $\kappa(\lambda)$, modified to use the graph of $z(\lambda)$ instead of the graph of $\kappa(\lambda)$ to guide the search. We have $z(0) = \sum_{P \in A} a_P + \sum_{i=1}^{n} a_i > 0$ (for $x = e$). The slope of the leftmost line segment of $z(\lambda)$ is $b - de < 0$, and the slope of the rightmost line segment is $b > 0$. The algorithm consists of the following three steps and finds a minimum of $z(\lambda)$ in $O(n'm' \log (n'^2/m'))$ [or $O(nm' \log (n^2/m' + 2))$] time. A cut $(X, \bar{X})$ in this network defines a solution x by $x_i = 1$ if vertex $i \in X$, and $x_i = 0$ otherwise.

Step 0. Start with initial values $\lambda_1 = 0$, $x^1 = e$, $z(\lambda_1) = f(x^1)$, and $h_1 = b - dx^1$. Choose λ_3 sufficiently large so that $x^3 = 0$; let $z(\lambda_3) = \lambda_3 b$ and $\beta_3 = b$.

Step 1. Compute $\lambda_2 = (z(\lambda_3) - z(\lambda_1))/(\beta_1 - \beta_3)$, pass λ_2 to two invocations, I and D, of the parametric preflow algorithm, and run them concurrently. If invocation I finds a minimum cut $(X_2^I, \bar{X}_2^I)$ first, suspend invocation D and go to Step 2 (the other case is symmetric). Compute $\beta_2 = b - dx^2$, the slope of the line segment of $z(\lambda)$ derived from this cut.

Step 2. If $\beta_2 = 0$, stop: $\lambda^* = \lambda_2$. Otherwise, if $\beta_2 > 0$, replace λ_3 by λ_2, back up invocation D to its state before it began processing λ_2, and go to Step 1. Otherwise, go to Step 3.

Step 3 ($\beta_2 < 0$). Finish the invocation D for λ_2. This produces a minimum cut $(X_2^D, \bar{X}_2^D)$, not necessarily the same as $(X_2^I, \bar{X}_2^I)$. If $\beta_2 \geqq 0$, stop: $\lambda^* = \lambda_2$. Otherwise, replace λ_1 by λ_2, back up invocation I to its state before processing λ_2, and go to Step 1.

We now describe an application of this algorithm.

Knapsack-constrained provisioning problems. We consider the following provisioning problem with a knapsack constraint that limits the weight of the selected items:

$$z(x^*) = \max_{x \in \{0,1\}^n} \{f(x) \mid dx \leqq b\}, \tag{4.8}$$

where $f(x)$ is given by (4.5), d is a positive n-vector of item weights and b is a scalar, the knapsack size, such that $\sum_{i \in V} d_i > b > 0$, $V = \{1, \cdots, n\}$. Thus, in addition to the benefit a_i obtained for including an individual item $i \in V$ in the knapsack, the model allows the possibility of an additional reward of $a_P \geqq 0$ for including all of the items that comprise a given subset $P \in A$. The (linear) *knapsack problem* is a special case of (4.8) in which all subsets $P \in A$ are singletons.

This NP-complete problem was suggested by Lawler [24]. Because of its many practical applications there is interest in the fast computation of bounds on $z(x^*)$. To this end we consider the Lagrangian function for (4.8):

$$L(x, \lambda) = f(x) - \lambda(dx - b) \quad \text{for } \lambda \geqq 0,$$

which has a finite infimum over $x \in \{0, 1\}^n$. For each $\lambda \geqq 0$, we define the *dual function*:

$$\Phi(\lambda) = \max_{x \in \{0,1\}^n} L(x, \lambda).$$

$\Phi(\lambda)$ is a piecewise linear convex function of λ, having at most $n-1$ line segments and $n-2$ breakpoints. We wish to solve the following *Lagrangian dual problem*:

$$\Phi(\lambda^*) = \min_{\lambda \geqq 0} \Phi(\lambda).$$

This value is an upper bound on $z(x^*)$ and can be used to construct heuristics and search procedures for computing an approximate or exact solution to (4.8). It can be evaluated by the above algorithm in $O(n'm' \log (n'^2/m'))$ [or $O(nm' \log (n^2/m'+2))$] time.

A special case of considerable practical importance is the *quadratic knapsack* problem, for which $f(x) = xAx$, where $A = [a_{ij}]$ is a nonnegative real symmetric matrix having no null rows. For this case, Gallo, Hammer, and Simeone [11] proposed an $O(n^2 \log n)$-time algorithm for creating a class of "upper planes" bounding $z(x)$. Chaillou, Hansen, and Mahieu [4] showed that its Lagrangian dual can be solved as a sequence of $O(n)$ minimum-cut problems in $O(n^2 m \log (n^2/m))$ time.

The problem of evaluating $\Phi(\lambda)$ for a fixed λ can be formulated as a minimum-cut problem using a graph construction similar to that described earlier for the maximum-density subgraph problem, thereby avoiding the use of a bipartite graph. Let $G = (V, E)$ be a directed graph with vertex set $V = \{1, \cdots, n\}$, arc set $E = \{(v, w)\colon a_{vw} > 0,\ v, w \in V\}$, and arc weights $a(v, w) = a_{vw}$. We add to G a source s, a sink t, and an arc (s, v) of capacity $a_v - \lambda d_v$ and an arc (v, t) of zero capacity for each $v \in V$, where $a_v = \sum_{w \in V} a_{vw}$. Using this network formulation, the above algorithm computes the Lagrangian relaxation of a quadratic knapsack problem in $O(nm \log (n^2/m))$ time.

4.4. Miscellaneous applications. Our last two applications both use the algorithm developed in § 3.3 for computing the min-cut capacity function $\kappa(\lambda)$ of a parametric minimum-cut problem. The first application is to a problem of computing critical load factors for modules of a distributed program in a two-processor distributed system [43]. The second application is to a problem of record segmentation between primary and secondary memory in large shared databases [9].

Critical load factors in two-processor distributed systems. Stone [43] modeled this problem by a graph $G = (V, E)$ in which $V = \{1, \cdots, n\}$ is the set of program modules and E is the set of pairs of modules that need to communicate with each other. The capacity of an arc $(v, w) \in E$ specifies the communication cost between modules v and w (it is infinity if the modules must be coresident). The two processors, say A and B, are represented by the source s and the sink t, respectively, that are appended to the

network. There is an arc (s, v) of capacity $\lambda b_v > 0$, where b_v is the given cost of executing program module v on processor B. There is an arc (v, t) of capacity $(1-\lambda)a_v > 0$, where a_v is the given cost of executing program module v on processor A.

The parameter $\lambda \in [0, 1]$ is the fraction of the time processor A that delivers useful cycles, commonly known as the *load factor*. For a fixed value of λ, a minimum cut $(X, \bar{X})$ in this network gives an optimum assignment of modules to processors. For $\lambda = 0$, a minimum cut $(X, \bar{X})$ with $|X|$ minimum has $X = \{s\}$, i.e., all modules are assigned to B. For $\lambda = 1$ a minimum cut $(X, \bar{X})$ with $|X|$ maximum has $\bar{X} = \{t\}$, i.e., all modules are assigned to A. The objective is to find the best assignment of program modules to processors for various values of λ, or to generate these assignments for each breakpoint of the min-cut capacity function $\kappa(\lambda)$. Lemma 2.4 implies that, at each breakpoint, one or more modules shift from one side of the cut to the other. By listing, for each module, the breakpoint at which it shifts from one side of the minimum cut to the other, one can determine what Stone calls the *critical load factor* for each module. The operating system can then use this list of critical load factors to do dynamic assignment of modules to processors. The algorithm of § 3.3 will compute the critical load factors of the modules in $O(nm' \log((n+2)^2/m'))$ time, where $m' = m + 2n$. Stone does not actually propose an algorithm for this computation.

Record segmentation in large shared databases. Eisner and Severance [9] have stated a model for segmenting records in a large shared database between primary and secondary memory. Such a database consists of a set of data items $S = \{1, \cdots, N\}$ and serves a set of users $T = \{1, \cdots, n\}$. Each user $w \in T$ retrieves a nonempty subset $S_w \subseteq S$ of data items and receives a "value" (satisfaction) of $b_w > 0$ whenever all of the items in S_w reside in primary memory. The cost of transporting and storing a data item $v \in S$ in primary memory is $\lambda a_v > 0$, where $a_v > 0$. The scalar $\lambda > 0$ is a conversion factor such that λ units of transportation and storage costs equals one unit of user value. The objective is to find a segmentation that minimizes the total cost minus user satisfaction.

For a fixed value of λ the problem can be formulated as a selection or provisioning problem [2], [36] as follows. Construct a bipartite graph having the data items S as its source part and the users T as its sink part. Construct an arc (v, w), $v \in S$, $w \in T$ of infinite capacity if data item v belongs to the set of data items S_w retrieved by user w. Create a supersource s and a supersink t, and append an arc (s, v) of capacity λa_v for each $v \in S$ and an arc (w, t) of capacity b_w for each $w \in T$. A min-cut $(X, \bar{X})$ separating s and t in this network necessarily partitions S and T into $(S_X, \bar{S}_X)$ and $(T_X, \bar{T}_X)$, respectively. It is easy to see that

$$c(X, \bar{X}) = \min_{S_X \cup T_X, \bar{S}_X \cup \bar{T}_X} \left\{ \lambda \sum_{v \in \bar{S}_X} a_v + \sum_{w \in T_X} b_w \right\}.$$

The value of λ plays an important role in this linear performance measure, and it depends on the system load. In practice it is necessary to create a list of primary storage assignments for all critical values of λ. The database inquiry program can then select and implement the best assignment at appropriate times. This table consists of all the breakpoints of the min-cut capacity function $\kappa(\lambda)$ and, for each data item and user, the parameter value at which it moves from one side to the other of a minimum cut. This information can be computed by the breakpoint algorithm of § 3.3 in $O((n+N)m \log((n+N)^2/m))$ [or $O(\min\{n, N\}m \log((\min\{n, N\})^2/m+2))$] time. The algorithm proposed by Eisner and Severance [9] for solving the parametric problem requires the solution of $O(\min\{n, N\})$ minimum-cut problems. Our algorithm improves their method by a factor of $\min\{n, N\}$. They also consider a nonlinear performance measure,

for which an algorithm such as that in § 4.3 can be used to derive bounds on an optimum solution. This bounding method gives an approximate solution, and the method can be used in a branch-and-bound algorithm to give an exact solution.

5. Remarks. We have shown how to extend the maximum-flow algorithm of Goldberg and Tarjan to solve a sequence of $O(n)$ related maximum-flow problems at a cost of only a constant factor over the time to solve one problem. The problems must be instances of the same parametric maximum flow problem and the corresponding parameter values must either consistently increase or consistently decrease. We have further shown how to extend the algorithm to generate the entire min-cut capacity function of such a parametric problem, assuming that the arc capacities are linear functions of the parameter.

We have applied our algorithms to solve a variety of combinatorial optimization problems, deriving improved time bounds for each of the problems considered. Our list of applications is meant to be illustrative, not exhaustive. We expect that more applications will be discovered. Although we have only considered a special form of the parametric maximum-flow problem, most of the parametric maximum-flow problems we have encountered in the literature can be put into this special form.

We have discussed only sequential algorithms in this paper, but our ideas extend to the realm of parallel algorithms. Specifically, the preflow algorithm has a parallel version that runs in $O(n^2 \log n)$ time using n processors on a parallel random-access machine. This version extends to the parametric preflow algorithm in exactly the same way as the sequential algorithm. Thus we obtain $O(n^2 \log n)$-time, n-processor parallel algorithms for the problems considered in §§ 2 and 3 and for each of the applications in § 4, where n is the number of vertices in the network.

There are a number of remaining open problems. One is to find additional applications. Our methods might extend to parametric maximum-flow problems that do not have the structure considered in this paper. Such problems include computing the arboricity of a graph [30], [33], computing properties of activity-selection games [45], and computing processor assignment for a two-processor system in which the processor speeds vary independently [17]. (This last problem is a two-parameter generalization of Stone's model [43] discussed in § 4.4.) Gusfield [19] has recently found a new application, to a problem considered by Cunningham [5], of solving the sequence of attack problems involved in the computation of the strength of an undirected graph. (This problem is related to the strength problem considered in § 4.2 but is harder.)

Another area for research is investigating whether an arbitrary maximum-flow algorithm can be extended to the parametric problem at a cost of only a constant factor in running time. One algorithm that we have unsuccessfully tried to extend in this way is that of Ahuja and Orlin [1]. Working in this direction, Martel [26] has recently discovered how to modify an algorithm based on the approach of Dinic [6] so that it solves the parametric problem with only a constant factor increase in running time.

REFERENCES

[1] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Working Paper No. 1905-87, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[2] M. L. BALINSKI, *On a selection problem*, Management Sci., 17 (1970), pp. 230–231.

[3] J. R. BROWN, *The sharing problem*, Oper. Res., 27 (1979), pp. 324-340.
[4] P. CHAILLOU, P. HANSEN, AND Y. MAHIEU, *Best network flow bounds for the quadratic knapsack problem*, presented at the NETFLO 83 International Workshop, Pisa, Italy, 1983. Lecture Notes in Mathematics, Springer-Verlag, Berlin, to appear.
[5] W. H. CUNNINGHAM, *Optimal attack and reinforcement of a network*, J. Assoc. Comput. Mach., 32 (1985), pp. 549-561.
[6] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277-1280.
[7] W. DINKELBACH, *On nonlinear fractional programming*, Management Sci., 13 (1967), pp. 492-498.
[8] J. EDMONDS, *Minimum partition of a matroid into independent subsets*, J. Res. Nat. Bur. Standards, 69B (1965), pp. 67-72.
[9] M. J. EISNER AND D. G. SEVERANCE, *Mathematical techniques for efficient record segmentation in large shared databases*, J. Assoc. Comput. Mach., 23 (1976), pp. 619-635.
[10] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.
[11] G. GALLO, P. HAMMER, AND B. SIMEONE, *Quadratic knapsack problems*, Math. Programming, 12 (1980), pp. 132-149.
[12] A. V. GOLDBERG, *Finding a maximum density subgraph*, Tech. Report No. UCB CSD 84/171, Computer Science Division (EECS), University of California, Berkeley, CA, 1984.
[13] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 136-146; J. Assoc. Comput. Mach., 35 (1988).
[14] M. GONDRAN AND M. MINOUX, *Graphs and Algorithms* (translated by S. Vajda), John Wiley, New York, 1984.
[15] M. D. GRIGORIADIS, *An efficient implementation of the network simplex method*, Math. Programming Study, 26 (1986), pp. 83-111.
[16] M. D. GRIGORIADIS AND K. RITTER, *A parametric method for semidefinite quadratic programs*, SIAM J. Control, 7 (1969), pp. 559-577.
[17] D. GUSFIELD, *Parametric combinatorial computing and a program of module distribution*, J. Assoc. Comput. Mach., 30 (1983), pp. 551-563.
[18] ——— *On scheduling transmissions in a network*, Tech. Report YALEU DCS TR 481, Department of Computer Science, Yale University, New Haven, CT, 1986.
[19] ———, *Computing the strength of a network in* $O(|V|^3|E|)$ *time*, Tech. Report CSE-87-2, Department of Electrical and Computer Engineering, University of California, Davis, CA, 1987.
[20] D. GUSFIELD, C. MARTEL, AND D. FERNANDEZ-BACA, *Fast algorithms for bipartite network flow*, SIAM J. Comput., 16 (1987), pp. 237-251.
[21] T. ICHIMORI, H. ISHII, AND T. NISHIDA, *Optimal sharing*, Math. Programming, 23 (1982), pp. 341-348.
[22] A. ITAI AND M. RODEH, *Scheduling transmissions in a network*, J. Algorithms, 6 (1985), pp. 409-429.
[23] J. R. ISBELL AND H. MARLOW, *Attrition games*, Naval Res. Logist. Quart., 2 (1956), pp. 71-93.
[24] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.
[25] ———, *Sequencing jobs to minimize total weighted completion time subject to precedence constraints*, Ann. Discrete Math., 2 (1978), pp. 75-90.
[26] C. MARTEL, *A comparison of phase and non-phase network algorithms*, Tech. Report CSE-87-7, Department of Electrical and Computer Engineering, University of California, Davis, CA, 1987.
[27] N. MEGIDDO, *Optimal flows in networks with multiple sources and sinks*, Math. Programming, 7 (1974), pp. 97-107.
[28] ———, *A good algorithm for lexicographically optimal flows in multi-terminal networks*, Bull. Amer. Math. Soc., 83 (1979), pp. 407-409.
[29] ———, *Combinatorial optimization with rational objective functions*, Math. Oper. Res., 4 (1979), pp. 414-424.
[30] C. ST. J. A. NASH-WILLIAMS, *Decomposition of finite graphs into forests*, J. London Math. Soc., 39 (1964), p. 12.
[31] M. W. PADBERG AND L. A. WOLSEY, *Fractional covers and forests and matchings*, Math. Programming, 29 (1984), pp. 1-14.
[32] J.-C. PICARD, *Maximal closure of a graph and applications to combinatorial problems*, Management Sci., 11 (1976), pp. 1268-1272.
[33] J.C. PICARD AND M. QUEYRANNE, *A network flow solution to some nonlinear* 0-1 *programming problems, with applications to graph theory*, Networks, 12 (1982), pp. 141-159.
[34] ———, *Selected applications of minimum cuts in networks*, INFOR, 20 (1982), pp. 394-422.
[35] J.-C. PICARD AND H. D. RATLIFF, *Minimum cuts and related problems*, Networks, 5 (1975), pp. 357-370.

[36] J. M. W. RHYS, *A selection problem of shared fixed costs and network flows*, Management Sci., 17 (1970), pp. 200–207.

[37] K. RITTER, *Ein verfahren zur losung parameterabhangiger, nichtlineare maximum probleme*, Unternehmensforschung, 6 (1962), pp. 149–166.

[38] S. SCHAIBLE, *Fractional programming* II: *On Dinkelbach's algorithm*, Management Sci., 22 (1976), pp. 868–873.

[39] J. B. SIDNEY, *Decomposition algorithm for single-machine sequencing with precedence relations and deferral costs*, Oper. Res., 23 (1975), pp. 283–298.

[40] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 24 (1983), pp. 362–391.

[41] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[42] C. STEIN, *Efficient algorithms for bipartite network flow*, unpublished manuscript, Department of Computer Science, Princeton University, Princeton, NJ, 1986.

[43] H. S. STONE, *Critical load factors in two-processor distributed systems*, IEEE Trans. Software Engrg., 4 (1978), pp. 254–258.

[44] R. E. TARJAN, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[45] D. M. TOPKIS, *Activity selection games and the minimum-cut problem*, Networks, 13 (1983), pp. 93–105.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 56-67, February 1989

004

LOWER BOUNDS FOR ACCESSING BINARY SEARCH TREES WITH ROTATIONS*

ROBERT WILBER†

Abstract. Two methods are given for obtaining lower bounds on the cost of accessing a sequence of nodes in a symmetrically ordered binary search tree, in a model where rotations can be done on the tree and the entire sequence is known before accessing begins (but the accesses must be done in the order given). For example, it can be proven that the bit-reversal permutation requires $\Theta(n \log n)$ time to access in this model. It is also shown that the expected cost of accessing random sequences in this model is the same as it is for the case where the tree is static.

Key words. data structures, binary search tree, rotation, lower bound

AMS(MOS) subject classification. 68P

1. Introduction. A binary search tree is a binary tree whose nodes are distinct members of some totally ordered set and for which the nodes are in symmetric order, i.e., each node is greater than all nodes in its left subtree and less than all nodes in its right subtree. Binary search trees can efficiently support such operations as insert, find, delete, find minimum, find maximum, split, and join. A variety of algorithms for maintaining binary search trees have been proposed that provide some combination of these operations [1]-[3], [5]-[9]. Most of these algorithms use *rotations* to modify binary trees. If node u is the left child of node v, then u is rotated over v by making v the new right child of u; this makes the old right child of u (if there is one) the new left child of v, and the old parent of v (if there is one) the new parent of u. The mirror image operation applies if u is the right child of v. A rotation preserves the symmetric order of the tree.

Here we consider the following problem. Given an initial binary search tree T_0, and a sequence s of nodes in T_0, what is the minimum time required to access the nodes in s (in the order given) when we are allowed to do rotations on the tree? We assume that the entire sequence s is known before we start (i.e., the algorithm is "offline"). The cost of assessing a node at depth d (where the root is at depth zero) is $d+1$, and the cost of a rotation is 1. One easy observation is that by losing at most a factor of 2, we may assume that we always access a node by rotating it to the root in some way. For, given an arbitrary rotation algorithm, we may simulate as follows. Whenever the algorithm accesses a node at depth d (at cost $d+1$), we instead do d rotations to bring the node to the root, access it there (at cost 1), and then do the reverse sequence of d rotations to bring the node back to where it was, for a total cost of $2d+1$. So without loss of generality we will consider only those algorithms that access a node by rotating it to the root. Such an algorithm shall be called a *standard search algorithm.* We let $\chi(s, T_0)$ denote the minimum cost of accessing the nodes in sequence s, starting from tree T_0, by a standard search algorithm. We are concerned with finding lower bounds for $\chi(s, T_0)$.

This problem is motivated by a conjecture of Sleator and Tarjan concerning their *splay algorithm* [9]. The splay algorithm is an online algorithm for maintaining binary

* Received by the editors March 11, 1987; accepted for publication (in revised form) March 17, 1988. A preliminary version of this paper appeared in the Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 61-70.

† AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

search trees using rotations in a way that has provably good amortized behavior. Although any single operation on an n node tree can take $\Omega(n)$ time, Sleator and Tarjan were able to prove the following surprising fact.

THEOREM 0 (Sleator and Tarjan). *Let V be a set of n keys. Let s be a sequence of keys in V and let t be the minimum time required to access s on any static tree (i.e., a tree that is fixed for sequence s and cannot be modified as the sequence is accessed). Let T_0 be an arbitrary binary search tree whose nodes are the keys in V. Then the time required by the splay algorithm to access s starting from tree T_0 is $O(t+n)$.*

Thus the splay algorithm does essentially as well as the best static algorithm, even though the splay algorithm is an online algorithm. Sleator and Tarjan have conjectured that their splay algorithm does as well as the best offline dynamic binary tree algorithm (an algorithm that does rotations and is allowed to pick its initial tree after seeing the sequence). That is, for any sequence s and for any initial tree T_0 they conjecture that the time used by the splay algorithm to access sequence s is $O(\chi(s, T_0)+n)$.

This "dynamic optimality conjecture" implies that, for any initial n node tree, the splay algorithm is able to access the nodes of the tree in sorted order in $O(n)$ time. This corollary of the conjecture has been proved by Tarjan [11]. Even this very restricted case required a difficult proof.

In order to prove (or disprove) the dynamic optimality conjecture, it may be useful to have bounds on $\chi(s, T_0)$. Here two methods for getting lower bounds on $\chi(s, T_0)$ are described. The first method, described in the next section, requires the use of an auxiliary structure called a *lower bound tree.* The bound obtained is a function of the lower bound tree as well as of s and T_0. This method can achieve tight bounds for some particular sequences, such as a $\Theta(n \log n)$ bound for the bit reversal permutation, and can also be used to get tight expected time bounds for sequences generated at random. However, because of the somewhat artificial introduction of a lower bound tree, the method seems likely not to provide tight bounds in general.

Culik and Wood [4] showed that the number of rotations needed to convert one n node symmetrically ordered binary tree into another is at most $2n-2$. Sleator, Tarjan, and Thurston [10] proved that the bound is at most $2n-6$, and that this is tight for infinitely many n. Thus $\chi(s, T_0)$ does not have a strong dependence on T_0; the choice of initial tree cannot change the cost by more than $2n-6$. The second lower bound method, given in § 3, makes use of this—not only is there no dependence upon a lower bound tree, there is no dependence upon the initial tree T_0. The second lower bound method can also be used to get the $\Theta(n \log n)$ bound for accessing the bit reversal permutation (although with a worse constant factor than the first method yields). It seems more likely to give tight bounds for particular sequences than the first method, although it has not proven to be as suitable for analyzing the expected cost of accessing a random sequence.

2. The first lower bound. For a sequence of integers s and integer i we let $s+i$ denote the sequence obtained by adding i to each element of s. If s is an integer sequence and $x \leqq y$, then $s|_x^y$ denotes the subsequence of s consisting of the elements in the interval $[x, y]$.

We will assume without loss of generality that the nodes of search trees are consecutive integers. If u is a child of v the rotation of u over v is denoted by the pair (u, v). The action of a standard search algorithm on some particular access sequence and some particular initial search tree is completely described by the sequence of rotations generated by the algorithm. If R is a sequence of rotations and $x \leqq y$, then $R|_x^y$ denotes the subsequence of R consisting of the pairs in $[x, y]^2$.

Let T be a search tree whose nodes are the integers in $[j, k]$, for some $j \leqq k$. Then a *lower bound tree for* T is a symmetrically ordered binary tree with $2(k-j)+1$ nodes whose leaves are the integers $j, j+1, \cdots, k$ and whose internal nodes are the half-integers $j+\frac{1}{2}, j+\frac{3}{2}, \cdots, k-\frac{1}{2}$. We define an integer function $\Lambda_1(s, T_0, \Upsilon)$ such that for any search tree with consecutive integer nodes T_0, any sequence s of nodes in T_0, and any lower bound tree Υ for T_0, we have $\chi(s, T_0) \geqq \Lambda_1(s, T_0, \Upsilon)$. For fixed s and T_0 the function Λ_1 will have different values for different lower bound trees Υ; this lower bound method requires judgement in choosing a lower bound tree that will maximize the value of Λ_1.

We now describe how to compute $\Lambda_1(s, T_0, \Upsilon)$. Let m be the length of sequence s and let U be the set of internal nodes of Υ. For each node $u \in U$ we compute its *score*, $\lambda(u)$, as follows. Let l and r be the leftmost and rightmost leaf nodes, respectively, in the subtree of Υ with root u. Let $\sigma' = s|_l^r$ and let h be the length of σ'. Let v be the lowest common ancestor in T_0 of the nodes in $[l, r]$ (we always have $v \in [l, r]$). Let sequence $\sigma = \sigma_0, \sigma_1, \cdots, \sigma_h$, where $\sigma_0 = v$ and, for $i \in [1, h]$, $\sigma_i = \sigma'_i$. Say that an integer $i \in [1, h]$ is a *u-transition* if $\sigma_{i-1} < u$ and $\sigma_i > u$ or if $\sigma_{i-1} > u$ and $\sigma_i < u$. We define $\lambda(u) = |\{i \in [1, h]: i \text{ is a } u\text{-transition}\}|$. The function Λ_1 is computed as

$$\Lambda_1(s, T_0, \Upsilon) = m + \sum_{u \in U} \lambda(u).$$

THEOREM 1. *Let T_0 be a search tree with consecutive integer nodes, let Υ be a lower bound tree for T_0, and let s be a sequence of nodes in T_0. Then $\chi(s, T_0) \geqq \Lambda_1(s, T_0, \Upsilon)$.*

Before we can prove Theorem 1 we need to define a kind of generalized subtree operation. Let T be a search tree with integer nodes, and let V be the set of nodes in T. For any node $v \in V$, the *left inner path of* v is the path that starts at the left child of v and proceeds downward along the right child links until a node is reached that has no right child. If v has no left child its left inner path is the null path. Likewise, the *right inner path* of v is the path that starts at the right child of v and proceeds along the left links. Let $x \leqq y$. The *restriction of T to the interval* $[x, y]$, denoted $T|_x^y$, is defined as follows. The set of nodes of $T|_x^y$ is $V' = V \cap [x, y]$. Let $v \in V'$ and let P be v's left inner path in T. If P has no nodes in V' then v has no left child in $T|_x^y$. Otherwise the first node in P that is in V' is the left child of v in $T|_x^y$. The right child of v in $T|_x^y$ is defined similarly—it is the first node in V' along the right inner path of v in T, if such a node exists, and otherwise v has no right child in $T|_x^y$. Clearly $T|_x^y$ is a symmetrically ordered binary tree, and its root is the lowest common ancestor in T of the nodes in V'. Also, if u is a child of v in T and $u, v \in V'$, then u is a child of v in $T|_x^y$. More generally, if u is a descendent of v in T and $u, v \in V'$, then u is a descendent of v in $T|_x^y$.

LEMMA 2. *Let T_1 be a search tree with integer nodes, let node u be a child of node v in T, and let T_2 be the tree that results when u is rotated over v in T_1. Let $x \leqq y$. If either u or v is not in the interval $[x, y]$ then $T_1|_x^y = T_2|_x^y$. On the other hand, if u and v are both in $[x, y]$ then $T_2|_x^y$ is the tree obtained by rotating u over v in $T_1|_x^y$.*

Proof. Let $T'_1 = T_1|_x^y$ and $T'_2 = T_2|_x^y$. Consider the case where at least one of u or v is not in $[x, y]$. These are four subcases: u is the left child of v in T_1 and $u < x$, u is the left child of v and $v > y$, u is the right child of v and $u > y$, and u is the right child of v and $v < x$. The fourth case is the mirror image of the second case, the second case is the time reversal of the third case, and the third case is the mirror image of the first case, so we need consider only the first case, where u is the left child of v in T_1 and $u < x$. Let w be any node in $[x, y]$. We must show that the children of w in T'_1 are the same as in T'_2. Since $u < x$ and $w \geqq x$ node u cannot be in the right subtree of

w in T_1. Since v is the parent of u, v cannot be in the right subtree of w either. Thus the rotation of u over v has no effect on the right subtree of w, so the right child of w in T_2' is the same as the right child of w in T_1'. Let P be the left inner path of w. If $w \neq v$ and v is not in P then P is unaffected by the rotation, so the left child of w in T_1' is the same as it is in T_2'. If v is in P then the effect of the rotation is to insert u into P, and if $v = w$ the effect is to remove u from P, but since $u \notin [x, y]$ this does not change the left child of w in the restricted tree. Thus the rotation does not change the restricted tree.

The case where u and v are both in $[x, y]$ is just as easy and it is left to the reader. □

Proof of Theorem 1. Let m be the length of sequence s. After a node has been rotated to the root of the search tree the cost of accessing it is 1, so the access cost for sequence s, exclusive of the cost of all rotations, is m. Therefore it suffices to show that $\sum_{u \in U} \lambda(u)$, where U is the set of internal nodes of Υ, is a lower bound on the number of rotations that must be carried out by any standard search algorithm.

Let n be the number of nodes in the search tree. We use induction on n. If $n = 1$ then there is only one possible lower bound tree—the tree with a single leaf node and no internal nodes. In that case $\sum_{u \in U} \lambda(u) = 0$ and there is nothing to prove. So assume $n \geq 2$. Let R be the sequence of rotations generated by some standard search algorithm that accesses the sequence s starting from tree T_0. Let r be the length of R and, for an integer $t \in [1, r]$, let T_t be the tree that results from applying R_t to T_{t-1}. Let w be the root of Υ and let Υ^1 and Υ^2 be the left and right subtrees, respectively, of w. Let $R^1 = R|_{-\infty}^{w}$ and $R^2 = R|_{w}^{\infty}$. Let M be the subsequence of R obtained by deleting those rotations that are in R^1 or R^2. The sequences R^1, R^2, and M are disjoint, so we have $r = |R^1| + |R^2| + |M|$.

For $i = 1, 2$ let U^i be the set of internal nodes of Υ^i. Let $s^1 = s|_{-\infty}^{w}$ and let $s^2 = s|_{w}^{\infty}$. For $t \in [0, r]$, let $T_t^1 = T_t|_{-\infty}^{w}$ and let $T_t^2 = T_t|_{w}^{\infty}$. Let $T_0^{1\prime}, T_1^{1\prime}, \cdots, T_{r_1}^{1\prime}$ be the sequence of search trees obtained by deleting from the sequence $T_0^1, T_1^1, \cdots, T_r^1$ those trees T_t^1 such that $T_t^1 = T_{t-1}^1$. By Lemma 2, $T_t^{1\prime}$ can be derived from $T_{t-1}^{1\prime}$ by applying rotation R_t^1, for all $t \in [1, r_1]$. Also, for all $t \in [0, r]$ if the root v of T_t is less than w then v is also the root of T_t^1. Therefore R^1 is a sequence of rotations that, starting from T_0^1, brings to the root the successive nodes in s^1. Tree Υ^1 is a lower bound tree for T_0^1, and the scores assigned to its internal nodes are precisely the scores obtained in the computation of $\Lambda_1(s^1, T_0^1, \Upsilon^1)$. So by the induction hypothesis, $|R^1| \geq \sum_{u \in U^1} \lambda(u)$. Similarly, R^2 is a sequence of rotations that, starting from T_0^2, brings to the root the successive nodes in s^2, and Υ^2 is a lower bound tree for T_0^2. So $|R^2| \geq \sum_{u \in U^2} \lambda(u)$.

Let σ be the sequence obtained by concatenating the root of T_0 with s. Let integer i be a w-transition. Without loss of generality, we may assume that $\sigma_{i-1} < w$ and $\sigma_i > w$. After the $(i-1)$th access, σ_{i-1} is at the root of the search tree and after the ith access, σ_i is at the root. Thus between the $(i-1)$th and ith accesses there must be a time t such that the root of T_{t-1} is less than w and the root of T_t is greater than w. Let y be the root of T_{t-1} and z be the root of T_t. We must have $R_t = (z, y) \in M$. Thus there is at least one rotation in M for every w-transition so $|M| \geq \lambda(w)$.

Putting it all together, we have

$$r = |R^1| + |R^2| + |M| \geq \sum_{u \in U^1} \lambda(u) + \sum_{u \in U^2} \lambda(u) + \lambda(w) = \sum_{u \in U} \lambda(u). \qquad \square$$

As an application of Theorem 1, we show that the bit reversal permutation on n nodes requires $\Theta(n \log n)$ time to access. Let k be a nonnegative integer and let $i \in [0, 2^k - 1]$. If the k-binary representation of i is $b_{k-1} b_{k-2} \cdots b_1 b_0$ then the *k-bit reversal of i*, denoted by $\mathrm{br}_k(i)$, is the integer whose k-bit binary representation is

$b_0 b_1 \cdots b_{k-2} b_{k-1}$. The bit reversal permutation on $n = 2^k$ nodes is the sequence $B^k = \mathrm{br}_k(0), \mathrm{br}_k(1), \cdots, \mathrm{br}_k(n-1)$.

COROLLARY 3. *Let k be a nonnegative integer and let $n = 2^k$. Let T_0 be any search tree with nodes $0, 1, \cdots, n-1$. Then $\chi(B^k, T_0) \geq n \log n + 1$.*

(Note. All logarithms in this paper are base 2.)

PROOF. Let Υ be the balanced lower bound tree for T_0. That is, the root of Υ is $w = \frac{1}{2}n - \frac{1}{2}$ and in general at depth $d \in [0, k-1]$ the internal nodes are $(2i-1)2^{-(d+1)}n - \frac{1}{2}$, for $1 \leq i \leq 2^d$. Let U be the set of internal nodes of Υ and let $\lambda(u)$ be the score assigned to a node $u \in U$ for sequence B^k and initial tree T_0. By Theorem 1, we have $\chi(B^k, T_0) \geq \Lambda_1(B^k, T_0, \Upsilon) = n + \sum_{u \in U} \lambda(u)$. We show by induction that $\sum_{u \in U} \lambda(u) \geq n \log n - n + 1$.

If $k = 0$ then $n \log n - n + 1 = 0$ and there is nothing to prove. Suppose $k \geq 1$. Let Υ^1 and Υ^2 be the left and right subtree of Υ, respectively, and let U^1 and U^2 be the sets of their internal nodes. The elements of sequence B^k are alternatively less than w and greater than w. Thus, regardless of the choice of T_0, we have $\lambda(w) \geq n - 1$. We have $B^k|_{-\infty}^{w} = B^{k-1}$ and $B^k|_{w}^{\infty} = B^{k-1} + 2^{k-1}$. Also, Υ_1 and Υ_2 are balanced lower bound trees, so by induction $\sum_{u \in U^i} \lambda(u) \geq \frac{1}{2}n \log(\frac{1}{2}n) - \frac{1}{2}n + 1$, for $i = 1, 2$. Thus

$$\sum_{u \in U} \lambda(u) = \sum_{u \in U^1} \lambda(u) + \sum_{u \in U^2} \lambda(u) + \lambda(w) \geq n \log n - n + 1. \qquad \square$$

We now consider the expected cost of accessing a sequence generated at random. If x is a random variable we denote the expected value of x by $E(x)$. For $i \in [1, n]$ let $p_i \geq 0$ and let $\sum_{i=1}^{n} p_i = 1$. Suppose sequence s is generated by choosing m integers in $[1, n]$ independently and at random, where the probability that k is chosen is p_k. If we construct the optimum static tree for the given probabilities the expected cost of accessing s by an algorithm that does not change the tree is $\Theta(m(1 + \sum_{i=1}^{n} p_i \log(1/p_i)))$ [6]. We now show that the same expected cost applies to offline algorithms that do rotations.

THEOREM 4. *Let sequence s be generated in the manner described above. Then for any initial tree T_0 we have $E(\chi(s, T_0)) = \Theta(m(1 + \sum_{i=1}^{n} p \log(1/p_i)))$.*

Proof. The upper bound follows from the bound for the static case. For the lower bound it is convenient to assume that $p_i > 0$ for all i. If $p_k = 0$ for some k then that integer will never be selected and it does not contribute to the sum of the $p_i \log(1/p_i)$s, so it can be ignored. We use a lower bound tree Υ that is balanced with respect to probabilities. That is, for each internal node u of Υ we make the probability of accessing a leaf in the left subtree of u as close as possible to the probability of accessing a leaf in the right subtree of u. More precisely, a probability-balanced tree with leaves j through k is constructed recursively as follows (Υ is constructed by applying this procedure to leaves 1 through n). If $j = k$ then the tree has the single leaf node j and we are done. Otherwise, normalize the probabilities by setting $P = \sum_{i=j}^{k} p_i$ and, for $i \in [j, k]$, $p_i' = p_i / P$. Let c be the least integer such that $\sum_{i=j}^{c} p_i' \geq \frac{1}{2}$. If $\sum_{i=j}^{c-1} p_i' + \frac{1}{2}p_c' < \frac{1}{2}$, then set $b = c$, otherwise set $b = c - 1$. The root of the tree is $u = b + \frac{1}{2}$. The left subtree of u is the probability-balanced tree for leaves j through b, and the right subtree of u is the probability-balanced tree for leaves $b + 1$ through k. The assumption that $p_i > 0$ for all i ensures that the sets of leaves of the left and right subtrees of u are nonempty. The integer c is called the *center leaf* of u.

Let U be the set of internal nodes of Υ. We modify the accounting for the lower bound as follows. For $u \in U$, let $\alpha(u)$ be the number of times the center leaf of u occurs in sequence s. The *modified score* of u is $\lambda'(u) = \lambda(u) + \frac{1}{2}\alpha(u)$. Each leaf node is the center leaf of at most two internal nodes, so $\sum_{u \in U} \alpha(u) \leq 2m$. Thus $\Lambda_1(s, T_0, \Upsilon) = m + \sum_{u \in U} \lambda(u) \geq \sum_{u \in U} \lambda'(u)$.

Suppose we step through the successive elements of s, on the ith step computing the modified scores of the internal nodes on the basis of T_0 and the subsequence $s_1, s_2, \cdots, s_i$. For $u \in U$ let $\lambda_i'(u)$ be the increase in $\lambda'(u)$ due to step i. Let $\lambda_i(u)$ be the increase in $\lambda(u)$, and let $\alpha_i(u)$ be the increase in $\alpha(u)$. Let the leftmost leaf in the subtree with root u be l_u and let the rightmost leaf be r_u. Let $q_1 = \sum_{l_u \leqq j < u} p_j$ and let $q_2 = \sum_{u < j \leqq r_u} p_j$. Let $P = q_1 + q_2$ be the probability that s_i is a leaf of the subtree with root u. For $k = 1, 2$ let $q_k' = q_k / P$. We have $E(\lambda_i(u)) = P\tau$, where τ is the conditional probability that i is a u-transition, given that $s_i \in [l_u, r_u]$. If the last node accessed under u was in u's left subtree then $\tau = q_2'$, and if it was in u's right subtree then $\tau = q_1'$. (If there was no node previously accessed under u then τ is either q_2' or q_1', depending upon whether the lowest common ancestor of nodes l through r in T_0 is less than or greater than u.) In any case $\tau \geqq \min(q_1', q_2')$. Let c be the center leaf of u, and let $p_c' = p_c / P$ be the conditional probability that $s_i = c$, given that $s_i \in [l_u, r_u]$. We have $E(\alpha_i(u)) = Pp_c'$. Thus $E(\lambda_i'(u)) = E(\lambda_i(u)) + \frac{1}{2}E(\alpha_i(u)) \geqq P(\min(q_1', q_2') + \frac{1}{2}p_c')$.

By the construction of Υ, we always have $\min(q_1', q_2') + \frac{1}{2}p_c' \geqq \frac{1}{2}$. So $E(\lambda_i'(u)) \geqq \frac{1}{2}P$. Summing over all m elements of s, we have $E(\lambda'(u)) \geqq \frac{1}{2}mP$. Thus

$$E(\Lambda_1(s, T_0, \Upsilon)) \geqq \sum_{u \in U} E(\lambda'(u)) \geqq \frac{m}{2} \sum_{u \in U} \sum_{j=l_u}^{r_u} p_j = \frac{m}{2} \sum_{j=1}^{n} d_j p_j,$$

where d_j is the number of internal nodes along the path from the root of Υ to leaf j.

As is easily shown by induction, $\sum_{j=1}^{n} d_j p_j \geqq \sum_{j=1}^{n} p_j \log(1/p_j)$, for any binary tree with n leaves in which the internal path length of leaf j is d_j (see, for example, Knuth [6, p. 445]). So $E(\Lambda_1(s, T_0, \Upsilon)) \geqq \frac{1}{2}m \sum_{i=1}^{n} p_i \log(1/p_i)$. Also, we always have $\Lambda_1(s, T_0, \Upsilon) \geqq m$. So, by Theorem 1,

$$E(\chi(s, T_0)) \geqq E(\Lambda_1(s, T_0, \Upsilon)) \geqq \max\left(m, \frac{m}{2} \sum_{i=1}^{n} p_i \log(1/p_i)\right)$$

$$\geqq \frac{m}{3}\left(1 + \sum_{i=1}^{n} p_i \log(1/p_i)\right). \qquad \square$$

3. The second lower bound. One problem with the lower bound of the previous section is that it requires picking a good lower bound tree. For an initial tree with n nodes and an access sequence of length m the optimum lower bound tree (i.e., the one that gives the largest bound) can be found in time $O(mn^3)$ by dynamic programming, since if a lower bound tree with root w is optimal for sequence s its left subtree is optimal for sequence $s|_{-\infty}^{w}$ and its right subtree is optimal for sequence $s|_{w}^{\infty}$. However, if we hope to prove that an offline search algorithm is optimal by showing that the time it takes to access a sequence matches some known lower bound then the dynamic programming technique is not very helpful—it is very difficult to get a grip on how the lower bound that comes out relates to the access sequence. Also, there is good reason to believe that even the optimal lower bound tree does not in general give a good bound. For a long sequence may have an access pattern that varies widely from one section of the sequence to another, so that no single lower bound tree works very well.

In this section an alternative way of computing a lower bound for $\chi(s, T_0)$ is described. It does not depend upon a lower bound tree (or even upon T_0) and seems to be better able to handle shifting access patterns.

Let s be a sequence of length m. We describe below a procedure for computing, for each $i \in [1, m]$, a quantity $\kappa(i)$, called the *score* of access i. The new lower bound

is defined as

$$\Lambda_2(s) = m + \sum_{i=1}^{m} \kappa(i).$$

When $\kappa(i) > 0$ the procedure for computing $\kappa(i)$ also determines the *inside accesses of i*, $b_1, b_2, \cdots, b_{\kappa(i)}$, the *inside nodes of i*, $v_j = s_{b_j}$, for $j \in [1, \kappa(i)]$, the *crossing accesses of i*, $c_1, c_2, \cdots, c_{\kappa(i)+1}$, and the *crossing nodes of i*, $w_j = s_{c_j}$, for $j \in [1, \kappa(i)+1]$.

A formal procedure for computing the score of i, together with its inside and crossing accesses, is shown in Fig. 1. Here we give an informal description of the procedure. We find the inside and crossing accesses of i by working backward in time from access i. The first crossing access, c_1, is simply the access prior to access i, namely $i-1$. The corresponding crossing node is $w_1 = s_{c_1} = s_{i-1}$. The second crossing access, c_2, is found by going backward in time from c_1 until we reach an access to a node on the side of s_i opposite from w_1. That is, if $w_1 > s_i$ then c_2 is the latest access prior to c_1 to a node less than or equal to s_i, and if $w_1 < s_i$ it is the latest access prior to c_1 to a node greater than or equal to s_i. The corresponding crossing node is $w_2 = s_{c_2}$. Assume without loss of generality that $w_1 < s_i$ so that $w_2 \geqq s_i$. Once the second crossing access has been found we can determine the first inside node, v_1. It is the greatest node less than s_i accessed after c_2 but before (or at) access c_1. The first inside access, b_1, is the access to v_1 within this time interval (if v_1 has been accessed more than once within the interval b_1 is the latest such access). So we have at this point $c_2 < b_1 \leqq c_1 < i$ and $w_1 \leqq v_1 < s_i \leqq w_2$. If $w_2 = s_i$, we are done; otherwise we proceed with the computation of c_3. The third crossing access, c_3, is the latest access prior to c_2 to a node between

```
procedure compute-kappa(s, i)
    if i = 1 then begin κ(i) ← 0; quit end;
    c1 ← i − 1; w1 ← s_{i−1};
    if w1 < s_i then v0 ← +∞ else v0 ← −∞;
    l ← 1;
    loop
        case
        w_l = s_i: begin κ(i) ← l − 1; quit end;
        w_l < s_i:
            begin
            Q ← {j ∈ [1, c_l): s_j ∈ [s_i, v_{l−1})};
            if Q = ∅ then begin κ(i) ← l − 1; quit end;
            c_{l+1} ← max Q;
            w_{l+1} ← s_{c_{l+1}};
            v_l ← max {s_j : j ∈ (c_{l+1}, c_l] and s_j < s_i};
            b_l ← max {j ∈ (c_{l+1}, c_l] : s_j = v_l};
            end
        w_l > s_i:
            begin
            Q ← {j ∈ [1, c_l) : s_j ∈ (v_{l−1}, s_i]};
            if Q = ∅ then begin κ(i) ← l − 1; quit end;
            c_{l+1} ← max Q;
            w_{l+1} ← s_{c_{l+1}};
            v_l ← min {s_j : j ∈ (c_{l+1}, c_l] and s_j > s_i};
            b_l ← max {j ∈ (c_{l+1}, c_l] : s_j = v_l};
            end
        endcase;
    endloop
```

FIG. 1. *Procedure for computing the score of an access.*

v_1 (exclusive) and s_i (inclusive), and the corresponding crossing node is $w_3 = s_{c_3}$. Then the second inside node, v_2, is computed as the smallest node greater than s_i that is accessed after c_3 but before (or at) c_2. The second inside access, b_2, is the latest access to v_2 within this time interval. At this point we have $c_3 < b_2 \leqq c_2 < b_1 \leqq c_1 < i$, and $w_1 \leqq v_1 < w_3 \leqq s_i < v_2 \leqq w_2$. If $w_3 = s_i$ we are done; otherwise c_4 is computed as the latest access prior to c_3 to a node between s_i (inclusive) and v_2 (exclusive), and w_4 is set to s_{c_4}. Then v_3 is computed as the greatest node smaller than s_i that is accessed after c_4 but before (or at) c_3. We continue in this way, with the crossing and inside nodes at alternate sides of s_i, getting closer to s_i as we go back in time. Eventually we reach the previous access of s_i (if there is one) or the beginning of the sequence (if there is not) at which point we stop. The score, $\kappa(i)$, is the number of inside accesses found.

It will help to go through an example. We distinguish between accesses and nodes by using boldface letters (ordered alphabetically), rather than integers, to denote nodes. Suppose the sequence s is **a, i, h, j, g, f, c, l, k, e, n, d, b, p, m, o, i**. This sequence is illustrated in Fig. 2. The nodes are in sorted order and the order of the access sequence is given by directed edges above the nodes. Suppose we wish to compute $\kappa(17)$, the score for the second access of node **i**. We start at access 17 and follow the access sequence backward in time. The previous access (16) is c_1, the first crossing access, so $w_1 = \mathbf{o}$. To find the second crossing access we go backward until we find an access to a node on the side of **i** opposite from **o**. The two accesses immediately prior to access 16 are to nodes **m** and **p**, which are on the same side of **i** as **o**. Access 13 is to node **b**, so $c_2 = 13$ and $w_2 = \mathbf{b}$. The first inside node is the closest node to **i** accessed after c_2 that is on the right side of **i**. This node is **m**, reached at access 15, so $b_1 = 15$ and $v_1 = \mathbf{m}$. To get the third crossing access we go backward from access c_2 until we reach an access to a node greater than or equal to **i** and less than $v_1 = \mathbf{m}$. The access before 13 is to node **d**, on the wrong side of **i**, the access before that is to **n**, on the wrong side of **m**, and the access before that is to **e**, on the wrong side of **i**. It is not until access 9, to **k**, that we find c_3. The second inside node is the closest node to **i** on its left side that is accessed after c_3 and before or at access c_2. This is node **e**, so $b_2 = 10$ and $v_2 = \mathbf{e}$. The fourth crossing access is obtained by going backward from c_3 until an access to a node between $v_2 = \mathbf{e}$ (exclusive) and **i** (inclusive) is reached. This is access 6, to node **f**. The third inside node is the closest node to **i** on its right side that is accessed after c_4 and before or at c_3. This is node **k**, the same as the third crossing node, so $b_3 = c_3 = 9$. The fifth crossing access is the first access before c_4 to a node

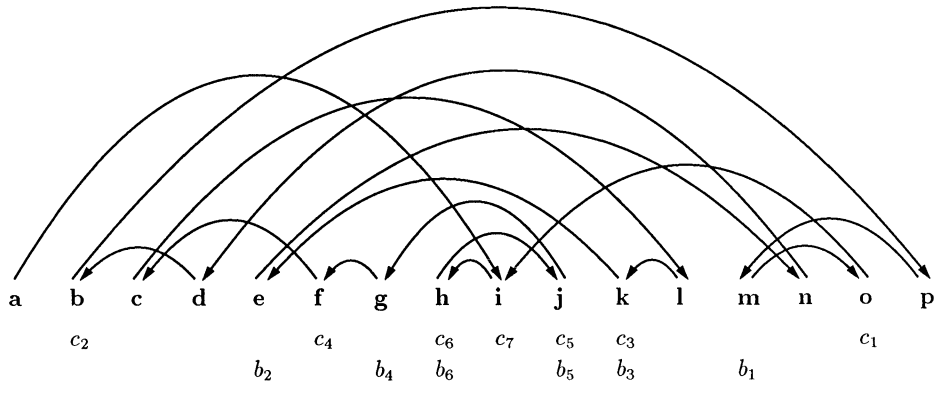


FIG. 2. *Computing the score for the second access of* **i** *in the sequence* **a, i, h, j, g, f, c, l, k, e, n, d, b, p, m, o, i.**

greater than or equal to **i** and less than $v_3 = \mathbf{k}$. This is access 4, to node $w_5 = \mathbf{j}$. The fourth inside node is the closest node to **i** on its left side that is accessed after c_5 and before or at c_4, and this is **g**, reached at access 5. Continuing in this way, we find $c_6 = 3$, $w_6 = \mathbf{h}$, and $b_5 = c_5 = 4$, $v_5 = w_5 = \mathbf{j}$. The seventh crossing access is 2, the previous access of **i**, and b_6 is determined to be equal to c_6. At this point the procedure terminates, with $\kappa(17) = 6$.

The crossing and inside accesses of i are all earlier than i and are equal to or later than the previous access of s_i (if there was one). Thus each of the crossing and inside nodes of i is above s_i at some time between the previous access of s_i and access i. Intuitively, after access $i-1$ we expect that the inside nodes of i will be on the path from the root to node s_i. Therefore for access i we expect that a standard search algorithm will have to rotate s_i over at least $\kappa(i)$ nodes, and then once s_i is at the root there will be a cost of 1 for accessing it. Of course, in general many other nodes will have been above s_i at some time since the previous access of s_i, but they can usually be kicked out of the way during the access of some inside node of i. Actually, you can always arrange to have as few inside nodes above s_i after access $i-1$ as you want, but for each inside node that is not above s_i either there will be some other node above s_i in its place or some extra cost will be imposed upon some access prior to the ith access.

The next two simple lemmas formally state properties that by now are probably quite clear.

LEMMA 5. *Let s be a sequence of integers of length m. Let $i \in [1, m]$, let $k = \kappa(i)$, and suppose $k \geqq 1$. Then the inside accesses of $i, b_1, \cdots, b_k$, the inside nodes of $i, v_1, \cdots, v_k$, the crossing accesses of $i, c_1, \cdots, c_{k+1}$, and the crossing nodes of $i, w_1, \cdots, w_{k+1}$, are all well defined. Also, the following relationships hold*:

$$c_{k+1} < b_k \leqq c_k < b_{k-1} \leqq c_{k-1} < \cdots < b_1 \leqq c_1 < i. \tag{1}$$

If $w_1 < s_i$ and k is odd then

$$w_1 \leqq v_1 < w_3 \leqq v_3 < \cdots < w_k \leqq v_k < s_i \leqq w_{k+1} < v_{k-1} \leqq w_{k-1} < \cdots < v_2 \leqq w_2, \tag{2a}$$

if $w_1 < s_i$ and k is even then

$$w_1 \leqq v_1 < w_3 \leqq v_3 < \cdots < w_{k-1} \leqq v_{k-1} < w_{k+1} \leqq s_i < v_k \leqq w_k < \cdots < v_2 \leqq w_2, \tag{2b}$$

if $w_1 > s_i$ and k is odd then

$$w_2 \leqq v_2 < w_4 \leqq v_4 < \cdots < w_{k-1} \leqq v_{k-1} < w_{k+1} \leqq s_i < v_k \leqq w_k < \cdots < v_1 \leqq w_1, \tag{2c}$$

and if $w_1 > s_i$ and k is even then

$$w_2 \leqq v_2 < w_4 \leqq v_4 < \cdots < w_k \leqq v_k < s_i \leqq w_{k+1} < v_{k-1} \leqq w_{k-1} < \cdots < v_1 \leqq w_1. \tag{2d}$$

Proof. Equations (2a)–(2d) are essentially the same, so assume without loss of generality that $w_1 < s_i$ and k is odd. Everything follows from an easy induction on l. Assume without loss of generality that $w_l < s_i$. If $l \leqq \kappa(i)$ then $\{j \in [1, c_l) : s_j \in [s_i, v_{l-1})\}$ is nonempty so c_{l+1} is well defined and is less than c_l. Also, $s_i \leqq w_{l+1} < v_{l-1}$, and the first branch of the **case** statement ensures that $w_{l+1} = s_i$ only if $l = \kappa(i)$. We have $w_l \in \{s_j : j \in (c_{l+1}, c_l]$ and $s_j < s_i\}$ so v_l is well defined and $v_l \geqq w_l$. The definition of v_l ensures that b_l is well defined and by definition of b_l we have $c_{l+1} < b_l \leqq c_l$. □

LEMMA 6. *Let s be a sequence of integers of length m and let $i \in [1, m]$ with $k = \kappa(i)$ positive. Let $c_1, \cdots, c_{k+1}$ be the crossing accesses of i, let $w_1, \cdots, w_{k+1}$ be the crossing nodes of i, and let $v_1, \cdots, v_k$ be the inside nodes of i. Let $l \in [1, k]$. Let interval I_l be*

either $(v_l, w_{l+1}]$ *or* $[w_{l+1}, v_l)$, *according to whether* $v_l < s_i$ *or* $v_l > s_i$. *Then* $i = \min\{j > c_{l+1} : s_j \in I_l\}$.

Proof. Assume without loss of generality that $v_l > s_i$. Let $Y = \{j > c_{l+1} : s_j \in (v_l, w_{l+1}]\}$. Let $h = \min Y$. Since $i \in Y$ we have $h \leqq i$. We show that the assumption that $h < i$ leads to a contradiction. Suppose $h < i$. Then $h \leqq i - 1 = c_1$. So, by (1), there is an $m \in [1, l]$ such that $h \in (c_{m+1}, c_m]$. We must have $w_m < s_i$ or $w_m > s_i$.

Suppose $w_m < s_i$. If $s_h < s_i$ then by definition of v_m we have $s_h \leqq v_m$. But $v_m \leqq v_l$ so $s_h \leqq v_l$, contradicting the definition of h. Thus the assumption that $w_m < s_i$ implies that $s_h \geqq s_i$. Since $s_{c_m} = w_m < s_i$ we must have $h < c_m$. Also, $w_{l+1} < v_{l-1} \leqq v_{m-1}$, so $s_h \in [s_i, v_{m-1})$. So by definition of c_{m+1} we have $c_{m+1} \geqq h$, a contradiction.

Therefore we must have $w_m > s_i$. Since $w_l < s_i$ we must have $m \neq l$, i.e., $m \leqq l - 1$. If $s_h > s_i$ then by definition of v_m we have $v_m \leqq s_h$. But $w_{l+1} \leqq w_{m+2} < v_m$, so $w_{l+1} < s_h$, contradicting the definition of h. So we must have $s_h \leqq s_i$ and since $v_{m-1} < v_l$ we have $s_h \in (v_{m-1}, s_i]$. Since $s_{c_m} = w_m > s_i$ we must have $h < c_m$. So by definition of c_{m+1} we have $c_{m+1} \geqq h$, a contradiction. □

THEOREM 7. *For any search tree* T_0 *and sequence* s *of nodes in* T_0 *we have* $\chi(s, T_0) \geqq \Lambda_2(s)$.

Proof. Let m be the length of s, and for each $i \in [1, m]$ define $\kappa(i)$ using the procedure `compute-kappa`(s, i). Once a node is at the root the cost of accessing it is 1, so it suffices to show that $\sum_{i=1}^{m} \kappa(i)$ is a lower bound on the number of rotations required by any standard search algorithm that accesses s.

We use an accounting argument. Let R be the sequence of rotations generated by some standard search algorithm acting on sequence s and initial tree T_0. Let r be the length of R. For $t \in [1, r]$, let T_t be the search tree obtained by applying rotation R_t to tree T_{t-1}. Let $\tau_0 = 0$ and, for $i \in [1, m]$, let τ_i be the smallest integer greater than or equal to τ_{i-1} such that s_i is at the root of T_{τ_i}.

For each time $t \in [1, r]$, we put a dollar on at most one node, determined as follows. Let $R_t = (u, v)$ (node u is rotated over node v). Let $I_{u,v} = (u, v]$ if $u < v$ and $I_{u,v} = [v, u)$ if $u > v$. Let j be the smallest integer such that $\tau_j \geqq t$ and $s_j \in I_{u,v}$, if such an integer exists. If j is undefined then no node gets a dollar at time t. Otherwise, a dollar is put on node s_j.

After rotation R_{τ_i} is carried out all the dollars on node s_i are thrown away. Since at most one dollar is put in the tree per rotation the number of dollars thrown away is a lower bound on the number of rotations used. Thus to get the desired bound it suffices to show that for each $i \in [1, m]$, after rotation R_{τ_i} has been carried out the number of dollars taken off node s_i is at least $\kappa(i)$.

Let $i \in [1, m]$ with $k = \kappa(i)$ positive. Let $b_1, \cdots, b_k$ be the inside accesses of i, let $v_1, \cdots, v_k$ be the inside nodes of i, let $c_1, \cdots, c_{k+1}$ be the crossing accesses of i, and let $w_1, \cdots, w_{k+1}$ be the crossing nodes of i. By setting $l = k$ in Lemma 6 we see that $i = \min\{j > c_{k+1} : s_j = s_i\}$, so no dollars are taken from s_i in the interval $(\tau_{c_{k+1}}, \tau_{c_1}]$. Therefore it suffices to show that for each $l \in [1, k]$ a dollar is placed on node s_i for some $t \in (\tau_{c_{l+1}}, \tau_{b_l}]$. (By Lemma 5 these k intervals are all disjoint.)

Let $l \in [1, k]$. Assume without loss of generality that $w_{l+1} \leqq s_i$. For all $t \in (\tau_{c_{l+1}}, \tau_{b_l}]$ let $a(t)$ be the lowest common ancestor of w_{l+1} and v_l in tree T_t. We have $a(\tau_{c_{l+1}}) = w_{l+1} \leqq s_i$ and $a(\tau_{b_l}) = v_l > s_i$. So there must be a $t \in (\tau_{c_{l+1}}, \tau_{b_l}]$ such that $a(t-1) \leqq s_i$ and $a(t) > s_i$. Let $R_t = (y, z)$. It is straightforward to verify that we must have $a(t-1) = z$ and $a(t) = y$, for otherwise the lowest common ancestor of w_{l+1} and v_l would not change with tth rotation. Also $w_{l+1} \leqq z$ and $y \leqq v_l$. Thus $s_i \in [z, y) \subseteq [w_{l+1}, v_l)$. Let h be the smallest integer such that $\tau_h \geqq t$. Since $t > \tau_{c_{l+1}}$ we have $h > c_{l+1}$. The node that gets the dollar for rotation t is s_j such that $j = \min\{g \geqq h : s_g \in [z, y)\}$ and by Lemma 6 that node is s_i. □

It is perhaps not immediately obvious that $\Lambda_2(s)$ ever gives a bound that is superlinear in the length of s. Here we show that $\Lambda_2(s)$ can be used to get the $\Theta(n \log n)$ bound for accessing the bit reversal permutation.

THEOREM 8. *Let B^k be the bit reversal permutation on $n = 2^k$ nodes. Then $\Lambda_2(B^k) \geqq \frac{1}{2} n \log n + 1$.*

Proof. It is convenient to index the sequence B^k from zero rather than 1, so that the ith access is to node $\mathrm{br}_k(i)$. By definition $\Lambda_2(s) = n + \sum_{i=0}^{n-1} \kappa(i)$, where $\kappa(i)$ is the number of inside accesses of i. We use induction on k to show that $\sum_{i=0}^{n-1} \kappa(i) \geqq \frac{1}{2} n \log n - n + 1$. When $k \leqq 1$ the claimed bound is zero so there is nothing to prove.

Suppose $k \geqq 2$. Clearly the sequence $\frac{1}{2}B_0^k, \frac{1}{2}B_1^k, \cdots, \frac{1}{2}B_{n/2-1}^k$ is precisely B^{k-1}. Let $i, j \in [0, n/2-1]$. Then j is an inside access of i for sequence B^k if and only if j is an inside access of i for sequence B^{k-1}. So by the induction hypothesis,

$$\sum_{i=0}^{n/2-1} \kappa(i) \geqq \frac{1}{2}\left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) - \frac{n}{2} + 1.$$

We also have $B^{k-1} = \frac{1}{2}(B_{n/2}^k - 1), \frac{1}{2}(B_{n/2+1}^k - 1), \cdots, \frac{1}{2}(B_{n-1}^k - 1)$. Let $i, j \in [n/2, n-1]$. Then j is a crossing access of i for sequence B^k if and only if $j - n/2$ is a crossing access of $i - n/2$ for sequence B^{k-1}. So each $i \in [n/2, n-1]$ has at least as many inside accesses in sequence B^k as $i - n/2$ has in sequence B^{k-1}. We will show that each $i \in [n/2, n-2]$ actually has at least one more inside access in B^k than $i - n/2$ has in B^{k-1}. When $i = n/2$ access $i - n/2$ has no crossing accesses and no inside accesses in B^{k-1}, but access i (to node 1) has two crossing accesses in B^k, namely, $i-1$ and 0, so it has one inside access. If $i \in [n/2+1, n-2]$ then $i - n/2$ has at least one crossing access in B^{k-1}, so it suffices to show that i has one more crossing access in B^k than $i - n/2$ has in B^{k-1}. Access i has a crossing access in $[n/2, n-1]$, namely $i-1$, so let c be the earliest crossing access of i that is greater than or equal to $n/2$. Let $u = B_i^k$. Suppose c is the lth crossing access of i. Assume without loss of generality that $B_c^k < u$. Let v be the $(l-1)$th inside node of i. (If $l = 1$ we may take v to be $+\infty$.) There is at least one even node in $[u, v)$ (namely $u+1$) and of these the one that is accessed latest is a crossing node of i, and its access is in $[0, n/2-1]$. So by the induction hypothesis,

$$\sum_{i=n/2}^{n-1} \kappa(i) \geqq \frac{1}{2}\left(\frac{n}{2}\right) \log\left(\frac{n}{2}\right) - \frac{n}{2} + 1 + \left(\frac{n}{2} - 1\right).$$

Therefore

$$\sum_{i=0}^{n-1} \kappa(i) \geqq \frac{1}{2} n \log n - n + 1. \qquad \square$$

Conclusions. Two methods for obtaining lower bounds on $\chi(s, T_0)$ have been described. The first is useful for getting bounds on the expected value of $\chi(s, T_0)$ when sequence s is chosen at random, whereas the second seems more likely to give tight bounds for specific sequences. The obvious unresolved problem is to obtain tight bounds for $\chi(s, T_0)$. That is, to find an efficient (polynomial-time) procedure for computing an upper bound on $\chi(s, T_0)$ that can be shown to match some known lower bound. Preferably the procedure would give an explicit method for optimally accessing sequence s offline. The function $\Lambda_2(s)$ may give a tight bound for the cost of accessing sequence s (up to a constant factor). However, I have been unable to find a binary search tree algorithm whose performance provably matches the $\Lambda_2(s)$ lower bound.

Of course, the ultimate goal is to find some simple online algorithm that always runs in time $O(\chi(s, T_0) + n)$. The splay algorithm may yet prove to be such an algorithm.

REFERENCES

[1] G. M. ADEL'SON-VEL'SKII AND E. M. LANDIS, *An algorithm for the organization of information*, Sov. Math. Dokl., 3 (1962), pp. 1259–1262.

[2] R. BAYER, *Symmetric binary* B*-trees: Data structure and maintenance algorithms*, Acta Informatica, 1 (1972), pp. 290–306.

[3] C. A. CRANE, *Linear lists and priority queues as balanced binary trees*, Tech. Report STAN-CS-72-259, Computer Science Department, Stanford University, Stanford, CA, 1972.

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

[5] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.

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

[7] J. NIEVERGELT AND E. M. REINGOLD, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33–43.

[8] H. OLIVIÉ, *A new class of balanced search trees: Half-balanced binary search trees*, Tech. Report 80-02, Interstedelijke Industriele Hogeschool Antwerpen-Mechelen, Antwerp, Belgium, 1980.

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

[10] D. D. SLEATOR, R. E. TARJAN, AND W. P. THURSTON, *Rotation distance, triangulations, and hyperbolic geometry*, Proc. 18th Annual ACM Symposium on Theory of Computing, (1986), pp. 122–135.

[11] R. E. TARJAN, *Sequential access in splay trees takes linear time*, Combinatorica, 5 (1985), pp. 367–378.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 68–81, February 1989

005

AN INCREMENTAL LINEAR-TIME ALGORITHM FOR RECOGNIZING INTERVAL GRAPHS*

NORBERT KORTE† AND ROLF H. MÖHRING‡

Abstract. The fastest-known algorithm for recognizing interval graphs [S. Booth and S. Lucker, *J. Comput. System Sci.*, 13 (1976), pp. 335–379] iteratively manipulates the system of all maximal cliques of the given graph in a rather complicated way in order to construct a consecutive arrangement (more precisely, a tree representation of all possible consecutive arrangements). This paper presents a much simpler algorithm using a related, but much more informative tree representation of interval graphs. This tree is constructed in an incremental fashion by adding vertices to the graph in a predefined order such that adding a vertex u takes $O(|\text{Adj}(u)|+1)$ amortized time.

Key words. interval graphs, incremental recognition, graph algorithm, perfect elimination scheme, modified *PQ*-tree, amortized time

AMS(MOS) subject classifications. 68C25, 68E10

1. Introduction. A graph (see [4] for all graph-theoretic notation not defined here) $G=(V, E)$ is called an *interval graph* if its vertices v can be represented by (not necessarily distinct) intervals I_v of the real line such that two vertices are adjacent if and only if the corresponding intervals intersect. Such a collection $(I_v)_{v\in V}$ is called an *interval representation* of G.

Interval graphs arise in many applications that deal with the arrangement of overlapping connected parts of a linear or sequential structure. Examples are seriation in archeology, consecutive retrieval in data base systems, scheduling problems, VLSI-layout problems, and many others (cf. [2], [4], [5], [8] for details and references). The main reason for these applications is the following characterization theorem due to [3], which relates the "sequential" structure of interval graphs to a consecutiveness property of their maximal cliques.

THEOREM 1.1 [3]. *A graph G is an interval graph if and only if the maximal cliques of G can be linearly ordered such that, for each vertex v, the maximal cliques containing v occur consecutively.*

We call such an arrangement of the maximal cliques a *consecutive arrangement.* Constructing such a consecutive arrangement is the basis of the fastest recognition algorithm for interval graphs presently available [1].

The **first phase** of this algorithm makes use of the fact that interval graphs are *triangulated* (cf. [4] for details) and thus have a *perfect vertex elimination scheme* (*perfect scheme* for short), i.e., an ordering $\delta=[v_1, v_2, \cdots, v_n]$ of the vertices of G such that each set $(\text{Adj}(v_i)\cup\{v_i\})\cap\{v_i, v_{i+1}, \cdots, v_n\}$ is a clique of G. (Here $\text{Adj}(v_i)$ denotes the set of vertices adjacent to v_i.) Testing whether such a scheme exists and constructing it can be done by lexicographic breadth-first search in $O(|V|+|E|)$ time (cf. [9], [11]). A perfect scheme $\delta=[v_1, \cdots, v_n]$ naturally induces a list of cliques $C(v_i)=\{v_j\in\text{Adj}(v_i)\,|\,j>i\}\cup\{v_i\}$ $(i=1, \cdots, n)$ which contains all maximal cliques.

* Received by the editors December 1, 1986; accepted for publication (in revised form) March 31, 1988. This work was done at the Institute of Operations Research, University of Bonn, Bonn, Federal Republic of Germany, and was supported by the Sonderforschungsbereich 303. A preliminary version of this paper was presented at the 12th Annual Workshop on Graphtheoretic Concepts in Computer Science, Munich, Federal Republic of Germany, 1986.

† Philips Design Zentrum, Vogt-Kölln-Straße 30, D-2000 Hamburg 54, Federal Republic of Germany.

‡ Fachbereich Mathematik (MA 6-1), Technische Universität Berlin, D-1000 Berlin 12, Federal Republic of Germany.

These maximal cliques are the input for the **second phase** of the algorithm, in which an attempt to construct a consecutive arrangement for the cliques is made. The data structure used in this phase is the *PQ-tree*. It was invented in [1] for the more general purpose of representing all permutations of a set M that are consistent with constraints of consecutiveness given by a system $\mathcal{M}$ of subsets of M with the convention that the elements of each $M' \in \mathcal{M}$ must occur consecutively in the permutation. In the interval graph case, M is the set of maximal cliques of G, and $\mathcal{M}$ consists of all $\mathcal{C}(v)$, $v \in V$, where $\mathcal{C}(v)$ denotes the set of all maximal cliques containing v.

A PQ-tree is a rooted tree T with two types of internal nodes: P and Q, which will be represented by circles and rectangles, respectively. The leaves of T are labeled 1-1 with the maximal cliques of the interval graph G (in general, with the elements of M).

The *frontier* of a PQ-tree T is the permutation of the maximal cliques obtained by the ordering of the leaves of T from left to right. PQ-trees T and T' are *equivalent*, if one can be obtained from the other by applying the following rules a finite number of times:

(1) Arbitrarily permute the successor nodes of a P-node.

(2) Reverse the order of the successor nodes of a Q-node.

For an example see Fig. 1, where the frontier of T is given by $[C_1, C_2, \cdots, C_6]$. An equivalent frontier is $[C_5, C_6, C_3, C_2, C_1, C_4]$.

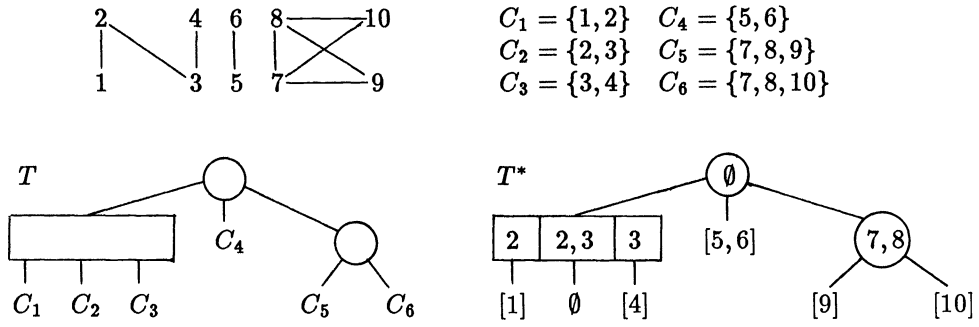


FIG. 1. *An interval graph G, its PQ-tree T, and its modified PQ-tree T*.*

In [1] it is shown that a graph G is an interval graph if and only if there is a PQ-tree T whose frontier represents a consecutive arrangement of the maximal cliques of G. They also develop an $O(|V|+|E|)$ algorithm that either constructs a PQ-tree for G (by successively making the maximal cliques from each $\mathcal{C}(v)$, $v \in V$, consecutive), or states that G is not an interval graph. If G is an interval graph, then all consecutive arrangements of the maximal cliques of G are obtained by taking equivalent PQ-trees.

The algorithm contains an update procedure that constructs, from a given PQ-tree T for a system $\mathcal{M} = \{\mathcal{C}(v_1), \cdots, \mathcal{C}(v_k)\}$, a PQ-tree T' representing $\mathcal{M}$ plus one additional constraint set $\mathcal{C}(v_{k+1})$. This is done in a bottom-up way along the tree T by comparing parts of the tree with a fixed number of *patterns* that induce certain local *replacements* in T. If none of the patterns applies, G is no interval graph. Otherwise, the algorithm returns T'. This update procedure is rather complicated, since many patterns must be considered, and large parts of the tree may be affected. Also the complexity analysis is rather intricate.

We develop a considerably simpler algorithm that follows a related approach (i.e., it uses a graph search method related to perfect schemes for constructing a consecutive arrangement), but differs essentially in the following points.

First, it uses a different data structure, the so-called *modified PQ-tree* introduced in [7]. Second, it is *incremental*, i.e., it grows the modified *PQ-tree* gradually by adding one vertex to the graph at a time (in a predefined order determined by lexicographic breadth-first search). The different data structure and the incremental growth result in a simple update scheme having three immediate advantages over that of Booth and Lueker: (a) fewer and simpler templates, (b) only a path of the tree must be modified when a vertex is added to the current graph (instead of a complete subtree), and (c) no need to be concerned with interior children of Q-nodes when exploring the path along which modifications occur (interior children of Q-nodes are a major headache for any PQ-tree implementation).

In addition, the modified PQ-tree contains much more information about the graph G than PQ-trees and is interesting in its own right. It is, for instance, used in [6], [7] to obtain fast algorithms for constructing an interval representation of G that respects given side constraints such as order relations among the intervals or their endpoints (*seriation with side constraints*; see also [8] for an overview).

Also, the incremental nature of the algorithm may be important for several applications. For instance, in gate matrix layout (a type of VLSI layout problem, see, e.g., [2]) we look for a large interval subgraph of an arbitrary graph representing conditions on the layout. One way to do this is to add as many vertices as we can while maintaining an interval graph. Hence data structures permitting incremental updating are very well suited to these applications.

The approach followed here has been extended in [6] to general *online algorithms* in which the graph is not known in advance, and vertices are added one at a time. This is, however, achieved at the cost of a higher worst-case complexity of $O(|V|^2)$ and a more complicated tree-updating scheme that must consider subtrees and interior sons of Q-nodes as does the algorithm of Booth and Lueker.

2. Modified *PQ*-trees. The *modified PQ-tree* (*MPQ*-tree) T^* assigns sets of vertices (possibly empty) to the nodes of a PQ-tree T representing an interval graph $G=(V, E)$. A P-node is assigned only one set, while a Q-node has a set for each of its sons (ordered from left to right according to the ordering of the sons).

For a P-node P, this set consists of those vertices of G contained in all maximal cliques represented by the subtree of P in T, but in no other cliques.

For a Q-node Q, the definition is more involved. Let $Q_1, \cdots, Q_m$ be the sons (in consecutive order) of Q, and let T_i be the subtree of T with root Q_i. (Note that $m \geqq 3$.) We then assign a set S_i (a so-called *section*) to Q for each Q_i. Section S_i contains all vertices that are contained in all maximal cliques of T_i *and* some other T_j, but *not* in any clique belonging to some other subtree of T that is not below Q. Since the maximal cliques containing a fixed vertex occur consecutively in T, it suffices to represent a vertex v belonging to several sections only in the leftmost and rightmost sections in which it appears. This convention assures that T^* represents G in $O(|V|)$ space, whereas PQ-trees require $O(|V|+|E|)$ space, since each maximal clique is stored separately. An example MPQ-tree is given in Fig. 1.

The following theorem from [7] summarizes some key properties of MPQ-trees:

THEOREM 2.1. [7]. *Let T be a PQ-tree for $G=(V, E)$ and let T^* be the associated MPQ-tree. Then we have the following*:

(a) *T^* can be obtained from T in $O(|V|+|E|)$ time and represents G in $O(|V|)$ space.*

(b) *Each maximal clique of G corresponds to a path in T^* from the root to a leaf, where each vertex $v \in V$ is as close as possible to the root.*

Property (b) states the essential property of *MPQ*-trees, which consists of distinguishing vertices of G with respect to their membership in maximal cliques while keeping the consecutiveness properties given by the underlying *PQ*-tree. In particular, the root of T^* contains all vertices belonging to all maximal cliques, while the leaves contain the *simplicial vertices* of G (vertices belonging only to one maximal clique).

As a consequence of this property, *MPQ*-trees may be viewed as *qbstract interval representations* of interval graphs that explicitly model the overlapping structure of the intervals while maintaining all degrees of freedom with respect to actual arrangements of the intervals by the *PQ*-tree operations. The higher a vertex u is located in the tree, the longer the respective interval I_u must be. *Standard* interval representations, in which the length of I_u is equal to the number of maximal cliques containing u, are given in Fig. 2 for the graphs and *MPQ*-trees from Figs. 1 and 3.

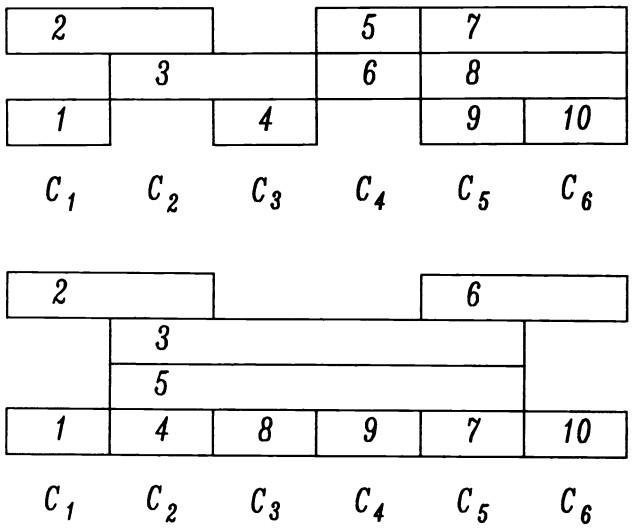


FIG. 2. *Standard interval representations for the graphs from Figs.* 1 *and* 3.

The following notation and statements concerning *MPQ*-trees will be used in the sequel. Given a node N of the tree T, we denote by V_N the associated vertex set if N is a P-node or a leaf. If N is a Q-node, we put $V_N := S_1 \cup \cdots \cup S_k$, i.e., the union of its sections. When no confusion is possible, we will sometimes identify a node N and its vertex set V_N.

LEMMA 2.2. *Let N be a Q-node. Let $S_1, \cdots, S_m$ (in this order) be the sections of N, and let V_i denote the set of vertices occurring below S_i in T $(i = 1, \cdots, m)$. Then we have the following*:

(a) $S_{i-1} \cap S_i \neq \emptyset$ *for* $i = 2, \cdots, m$.

(b) $S_1 \subseteq S_2$ *and* $S_{m-1} \supseteq S_m$.

(c) $V_1 \neq \emptyset$ *and* $V_m \neq \emptyset$.

(d) $(S_i \cap S_{i+1}) \setminus S_1 \neq \emptyset$ *and* $(S_{i-1} \cap S_i) \setminus S_m \neq \emptyset$ *for* $i = 2, \cdots, m-1$.

Proof. If $S_{i-1} \cap S_i = \emptyset$, then any two maximal cliques C_1, C_2 going through one of the sections $S_1, \cdots, S_{i-1}$ and $S_i, \cdots, S_m$, respectively, have only vertices in common that are above N in T. Hence, reversing the order of the maximal cliques going through

$S_i, \cdots, S_m$ and $S_1, \cdots, S_{i-1}$, respectively, gives another consecutive arrangement because of $m \geqq 3$. This contradicts the fact that only the reversal of the Q-node as a whole yields a consecutive arrangement. Hence (a) follows.

By definition of the sections, S_1 contains all the vertices that are in all the maximal cliques going through S_1 and some other S_j, but not those that occur in any other maximal clique. Since the S_j must also contain the respective vertices from S_1, this gives $S_1 = \bigcup_{j=2}^{m} (S_1 \cap S_j)$. The consecutiveness property implies $S_1 \cap S_j \subseteq S_1 \cap S_2$ for all $j \geqq 2$. Hence, $S_1 = S_1 \cap S_2$, which shows (b) for S_1. The proof for S_m is completely analogous.

Let C_1, C_2 be maximal cliques through S_1, S_2, respectively. Then $C_1 \backslash C_2 \neq \emptyset$ because of the maximality. All vertices above N in T belong to $C_1 \cap C_2$, and all vertices in $C_1 \cap V_N = S_1$ belong to $C_2 \cap V_N = S_2$ because of (b). So, $C_1 \backslash C_2 \subseteq V_1$, which gives $V_1 \neq \emptyset$, and also, by symmetry, $V_m \neq \emptyset$.

Assume that $(S_i \cap S_{i+1}) \backslash S_1 = \emptyset$, i.e., $S_i \cap S_{i+1} \subseteq S_1$. Then reversing the order of the maximal cliques going through $S_1, S_2, \cdots, S_i$ (there are at least two since $i \geqq 2$) gives another consecutive arrangement, since the overlapping condition with S_{i+1} is preserved. This contradicts the definition of a Q-node as in the proof of (a). □

3. Lexicographic breadth-first search. Lexicographic breadth-first search (LEXBFS for short) was introduced in [9] for the purpose of constructing perfect elimination schemes. LEXBFS is like the usual breadth-first search (BFS) with the additional rule that vertices with earlier visited neighbors are preferred.

More precisely, we say that $v, w \in V$ *disagree* on $u \in V$ is one of them is adjacent to u, but not both. Then LEXBFS produces an ordering $\lambda = [v_1, \cdots, v_n]$ of V with the property:

(3.1) If there is some v_i on which v_j, v_k with $i < j < k$ disagree, then the leftmost vertex on which they disagree is adjacent to v_j.

We will call this leftmost vertex that causes v_j to enter λ before v_k the *distinguisher* of v_j with respect to v_k.

THEOREM 3.1 [9]. *Let $G = (V, E)$ be a graph.*

(a) *A* LEXBFS*-ordering $\lambda = [v_1, \cdots, v_n]$ of G can be obtained in $O(|V|+|E|)$ time.*

(b) *G is triangulated if and only if $\delta = \lambda^R = [v_n, v_{n-1}, \cdots, v_1]$ is a perfect scheme. This can be tested for in $O(|V|+|E|)$ time.*

For implementation details (which are not too involved), we refer to [9], [4]. An example is given in Fig. 3.

We will show now that any LEXBFS traverses the *MPQ*-tree of a graph in a very special way. This feature will be exploited by our recognition algorithm.

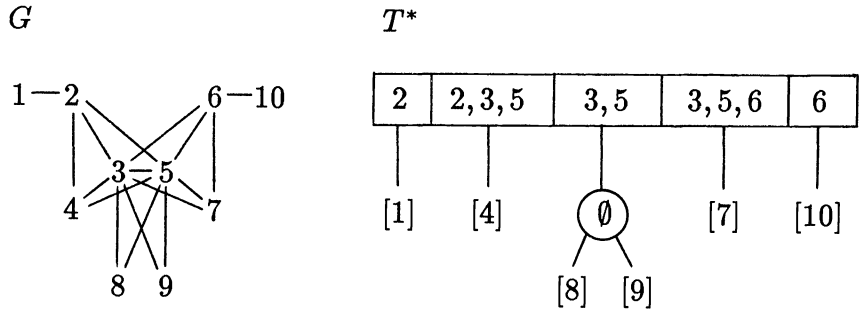


FIG 3. *A* LEXBFS*-ordering of an interval graph and the associated MPQ-tree.*

LEMMA 3.2. *Let $G=(V, E)$ be an interval graph, and let T be its associated MPQ-tree. Let N be a Q-node with sections $S_1, \cdots, S_m$ (in this order) and let V_i denote the set of vertices of G occurring below S_i in T.*

Then for any LEXBFS*-ordering λ, the rightmost vertex of $\bar{V} := S_1 \cup V_1 \cup \cdots \cup S_m \cup V_m$ with respect to λ is a member of V_1 or V_m.*

Proof. We will consider certain points or stages in the construction of λ. These stages occur when all vertices from $\bar{V} := \bigcup_{i=1}^{m} (S_i \cup V_i)$ that are forced by some earlier distinguisher have been visited, and the next distinguisher for vertices of $\bar{V}$ becomes active, i.e., determines the choice of the next vertices from $\bar{V}$ to be visited. We will show the following properties for each such stage:

(3.2) The already completely visited sets $S_i \cup V_i$ form a consecutive subsequence of $S_1 \cup V_1, \cdots, S_m \cup V_m$.

(3.3) If, at the current stage, $S_1 \cup V_1$ and $S_m \cup V_m$ are not completely visited, then at most one of them is completely visited at the next stage.

Obviously, (3.2) and (3.3) imply that the last vertex $v \in \bar{V}$ to be visited by λ belongs either to $S_1 \cup V_1$ or to $S_m \cup V_m$. Since $S_1 \subseteq S_2$ and $S_m \subseteq S_{m-1}$ (Lemma 2.2(b)), S_1 and S_m are then already completely visited. So $v \in V_1$ or $v \in V_m$, proving the lemma.

So it remains to show (3.2) and (3.3). We will do this by induction along the different stages. To start with, let u_1 be the first vertex in λ that disagrees on vertices from $\bar{V}$, i.e., u_1 is adjacent to some, but not to all vertices from $\bar{V}$. Since all vertices in an ancestor of node N are adjacent to all $v \in \bar{V}$, it follows that $u_1 \in \bar{V}$ and that u_1 is the first distinguisher for vertices from $\bar{V}$. So from $\bar{V}$, $\mathrm{Adj}(u_1)$ enters λ before any $v \in \bar{V} \backslash \mathrm{Adj}(u_1)$. If $u_1 \in \bar{S} := S_1 \cup \cdots \cup S_m$, then $\mathrm{Adj}(u_1) \cap \bar{V} = \bigcup_{u_1 \in S_i} (S_i \cup V_i)$, and these $S_i \cup V_i$ are consecutive. So (3.2) holds. Since u_1 is not adjacent to all $v \in \bar{V}$, we have $u_1 \notin S_1 \cap S_m$, which implies that V_1 or V_m still contains unvisited vertices after $\mathrm{Adj}(u_1)$ has been added to λ. This shows (3.3). If $u_1 \notin \bar{S}$, then u_1 is in some V_{i_0} (since all vertices above N are adjacent to every $v \in \bar{V}$). Then $\mathrm{Adj}(u_1) \cap \bar{V} \subseteq S_{i_0} \cup V_{i_0}$ and either no $S_i \cup V_i$ (if the inclusion is strict) or exactly $S_{i_0} \cup V_{i_0}$ has been completely visited after $\mathrm{Adj}(u_1)$ is added to λ. So (3.2) and (3.3) hold. Together, the cases $u_1 \in \bar{S}$ and $u_1 \notin \bar{S}$ yield the inductive base for the first stage.

So assume now that (3.2) and (3.3) were valid at the last stage, and that $u_1, \cdots, u_k$ are the previous distinguishers. Then, at the last stage, $\mathrm{Adj}(u_k) \cap \bar{V}$ was added to λ before any vertex $v \in \bar{V} \backslash \bigcup_{j=1}^{k} \mathrm{Adj}(u_j)$. Let $S_{i_0} \cup V_{i_0}, \cdots, S_{j_0} \cup V_{j_0}$ be the already completely visited sets after $\mathrm{Adj}(u_k)$ has been added to λ. We distinguish several cases.

Case 1. $i_0 = 1$, i.e., $S_1 \cup V_1$ is already completely visited. If $j_0 = m-1$, then only $S_m \cup V_m$ contains unvisited vertices and these are contained in V_m since $S_m \subseteq S_{m-1}$ and S_{m-1} is completely visited. So we may assume that $j_0 \leqq m-2$. Because of Lemma 2.2(a), $S_{j_0} \cap S_{j_0+1} \neq \emptyset$. For every $v \in S_{j_0} \cap S_{j_0+1}$, let $S_{r(v)}$ be the rightmost section containing v. We call $r(v)$ the *right range* of v. Let r be the minimum right range of a vertex $v \in S_{j_0} \cap S_{j_0+1}$. Any vertex $u \in S_{j_0}$ with $r(u) = r$ is said to have a *short right range.* By definition, $r > j_0$. Lemma 2.2(d) implies that $r < m$, i.e., every vertex u with short right range is not adjacent to any unvisited vertex from $(S_{r+1} \cup V_{r+1}) \cup \cdots \cup (S_m \cup V_m)$. Now consider two unvisited vertices $v_i \in S_i \cup V_i$ with $j_0 < i \leqq r$ and $v_j \in S_j \cup V_j$ with $j > r$. Then (3.2) and the definition of r imply that the only already visited vertices that disagree on v_i and v_j are vertices $u \in S_{j_0}$ with short range. So the first such u in λ is a distinguisher that forces all unvisited vertices from $\bigcup_{i=j_0+1}^{r} (S_i \cup V_i)$ to enter λ because of (3.1). This shows (3.2) and also (3.3) because of $r < m$.

Case 2. $j_0 = m$. This case is completely symmetric to the previous one.

Case 3. $i_0 = 2$ and $j_0 = m-1$. Then, only $S_1 \cup V_1$ and $S_m \cup V_m$ contain unvisited vertices from $\bar{V}$. Because of Lemma 2.2(a) these must belong to $V_1 \cup V_m$, which proves the lemma in this case.

Case 4. ($i_0 > 2$ and $j_0 \leqq m-1$) or ($i_0 \geqq 2$ and $i_0 < m-1$). Similar to Cases 1 and 2, we consider vertices $u \in S_{j_0} \cap S_{j_0+1}$ with short right range r and vertices $w \in S_{i_0-1} \cap S_{i_0}$ with short left range l. The assumptions on i_0 and j_0 guarantee that $l > 2$ or $r < m$. If $l > 2$ and $r < m-1$, then the first vertex in λ with short right or left range is the next distinguisher u_{k+1} that forces at least the sets $S_i \cup V_i$ in its short range to enter λ and possibly also on the other end if $l = l(u_{k+1})$ and $r = r(u_{k+1})$. So (3.2) and (3.3) hold. If $l > 2$ and $r = m-1$, then the first vertex with short left range or with neighbors in V_m but not in V_1 is the next distinguisher, and (3.2) and (3.3) follow. The final case $l = 2$ and $r < m-1$ is completely symmetric. □

In the example graph in Fig. 3, the stages are as follows. The first distinguisher is $u_1 = 1$, and it forces $S_1 \cup V_1 = \{1, 2\}$ into λ. Then $u_2 = 2$ is the first vertex with short right range $r = 2$. It forces $S_2 \cup V_2 = \{2, 3, 4, 5\}$ to enter λ. Then $u_3 = 3$, forcing $S_3 \cup V_3 \cup S_4 \cup V_4 = \{6, 7, 8, 9\}$ to enter λ. Finally, $u_4 = 6$, forcing $v = 10 \in V_m$ into λ.

Call a maximal clique C of G an *outer clique* if there is a consecutive arrangement of the maximal cliques of G with C as first (or last) clique. All maximal cliques that are not outer cliques are called *inner cliques*. If LEXBFS is viewed as traversing maximal cliques rather than vertices (a clique is traversed when its last vertex has been traversed), the maximal cliques for a vertex are not necessarily traversed consecutively (for example, vertex 6 of Fig. 3). But Lemma 3.2 implies that any such traversal ends with an outer clique. The following theorem gives a stronger version of this conclusion.

THEOREM 3.3. *Let $G = (V, E)$ be an arbitrary graph with associated* LEXBFS*-ordering $\lambda = [v_1, \cdots, v_n]$, and let G_i denote the subgraph of G induced by $V_i = \{v_1, \cdots, v_i\}$.*

Then G is an interval graph if and only if, for each $i = 1, \cdots, n-1$, G_i is an interval graph and $\mathrm{Adj}(v_{i+1}) \cap V_i$ *is contained in an outer clique of G_i. In this case, v_{i+1} is a simplicial vertex of G_{i+1}.*

Proof. If G is an interval graph then, by heredity, each G_i is an interval graph, and thus triangulated. Furthermore, $\lambda_{i+1} := [v_1, \cdots, v_{i+1}]$ is a LEXBFS-ordering for G_{i+1} and thus, by the results of [9], v_{i+1} is a simplicial vertex of G_{i+1}, i.e., it is contained in a unique maximal clique C of G_{i+1}. By Lemma 3.2, C is an outer clique of G_{i+1}. Since v_{i+1} is simplicial, $\mathrm{Adj}(v_{i+1}) \cap V_i \subseteq C$.

In the converse direction, suppose that G_i is an interval graph (which is trivial for $i = 1$) and $\mathrm{Adj}(v_{i+1}) \cap V_i$ is contained in an outer clique of G_i. Then there is a consecutive arrangement $C_1, \cdots, C_m$ of the maximal cliques of G_i with $C_0 := \mathrm{Adj}(v_{i+1}) \cap V_i \subseteq C_1$. Then either $C_1 \cup \{v_{i+1}\}, C_2, \cdots, C_m$ (if $C_0 = C_1$) or $C_0 \cup \{v_{i+1}\}, C_1, C_2, \cdots, C_m$ (if $C_0 \subset C_1$) is a consecutive arrangement of the maximal cliques of G_{i+1}. Thus, G_{i+1} is an interval graph with simplicial vertex v_{i+1}. The iterative application of this argument then yields that $G_n = G$ is an interval graph. □

Note that it is not necessary to know whether G is triangulated or not. Furthermore, Lemma 3.2 and the theorem do not hold for other graph-search orderings λ yielding perfect schemes (such as the simple maximal cardinality search considered in [11]). BFS together with lexicographical tie breaking are essential for these results.

4. The algorithm for growing the *MPQ*-tree. Theorem 3.3 is the basis for an incremental algorithm that traverses G in a LEXBFS-ordering and iteratively builds the *MPQ*-tree T_{i+1} representing G_{i+1} from the *MPQ*-tree T_i for G_i if G_{i+1} is an interval

graph (i.e., if v_{i+1} has only neighbors in some outer clique of G_i), or states that G_{i+1} (and thus $G = G_n$) is not an interval graph.

Each such *building step* consists of two phases, a *labeling phase*, and a *tree-updating phase.*

In the labeling phase, each node N (including Q-nodes) of the current tree T and each section S of a Q-node is labeled according to how the new vertex u is related to the vertices of N or S. The label is ∞, 1, or 0 if u is adjacent to all, some, or no vertex from N or S, respectively. Empty sets obtain the label 0. Labels 1 and ∞ are called *positive* labels. Note that a Q-node has a label for itself and for each of its sections. These labels will be used for testing the condition that u is adjacent only to vertices from an outer clique of the current graph.

LEMMA 4.1. *Let G be an interval graph, let T be its associated MPQ-tree, and let u be a vertex to be added to G. Then $G+u$ is an interval graph if and only if* (a) *and* (b) *hold*:

(a) *All nodes with a positive label are contained in a unique path of T.*

(b) *For each Q-node N with positive label, $\mathrm{Adj}(u) \cap V_N$ is contained in an outer section of Q.*

The proof follows immediately from Theorem 3.3. Note that (b) does not imply that only the outer section S has a positive label. There may be others, but all vertices in them that are adjacent to u are also contained in S. So, in particular, the sections with a positive label are consecutive and have label 1, except the outer section, which has label 1 or ∞.

If the labeling phase is affirmative (i.e., $G+u$ is an interval graph), then the new *MPQ*-tree $T(G+u)$ of $G+u$ is obtained by modifying $T(G)$ in the updating phase. This modification is *local* with respect to $T(G)$, as is shown by Lemma 4.2 below.

We use the following notation in the updating phase. G is the given graph, and $T(G)$ is its *MPQ*-tree. The vertex to be added is u, and $\mathrm{Adj}(u)$ is its set of adjacent vertices in G. We will omit the trivial case $\mathrm{Adj}(u) = \emptyset$, in which the tree updating can obviously be carried out in constant time (even the labeling phase can be omitted). Let P' denote the unique path of $T(G)$ containing all nodes with positive label, and let P be a path from the root to a leaf containing P'. There are several possible choices for P if P' does not contain a leaf of $T(G)$, but because of Lemma 4.1, P can be chosen such that it corresponds to a maximal outer clique C of G. Let N_* be the lowest node in P with label 1 or ∞. If P contains nonempty P-nodes or sections above N_* with label 0 or 1, let N^* be the highest such P-node or Q-node containing the section. Otherwise put $N_* = N^*$.

LEMMA 4.2. *In the above situation, exactly one of the two following cases applies*:

Case (a) *All nonempty P-nodes and sections on P have label ∞. Then in $G+u$, the maximal clique C of G is replaced by $C+u$. $T(G+u)$ is obtained from $T(G)$ by adding u to the leaf of P.*

Case (b) *There are nonempty P-nodes or sections or a leaf on P with label* 0 *or* 1. *Then in $G+u$, the maximal clique $\mathrm{Adj}(u) \cup \{u\}$ is added, and $T(G+u)$ is obtained from $T(G)$ by modifying only nodes between N_* and N^* on P (including N_*, N^*).*

Proof. Case (a) is obvious. Case (b) follows by observing that the set of cliques of $G+u$ differs from that of G only by adding the new maximal clique $C_0 := \mathrm{Adj}(u) \cup \{u\}$. This affects C and all maximal cliques of G that have a nonempty intersection with $\mathrm{Adj}(u) \subseteq C$. However, vertices belonging to nodes N not on P or below N_* or above N^* are either adjacent to no $v \in \mathrm{Adj}(u)$, or to all such v. So only vertices belonging to nodes between N_* and N^* must be reorganized. □

The construction of $T(G+u)$ in Case (a) is obvious. In Case (b), the modification is done by traversing the path P bottom-up from N_* to N^*.

In each step, the current node N (together with its associated sets and its label) is compared with a small number of patterns that trigger the appropriate modification of N. The combination of pattern recognition and modification is called a *template*. There are three groups of templates, depending on whether the current node N is a leaf, a P-node, or a Q-node. Each group has a template for up to three subcases, depending on whether or not $N = N_*$, $N_* \neq N^*$, etc. These groups of templates are displayed in Figs. 4–6. In all cases, $V_N = A \cup B$ denotes the partition of the vertex set V_N of the current node or section into the set A of vertices adjacent to u and the set B of vertices not adjacent to u. Furthermore, T_i and T_i' always denote subtrees of

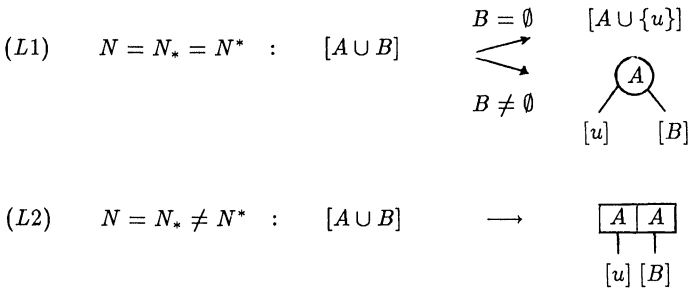
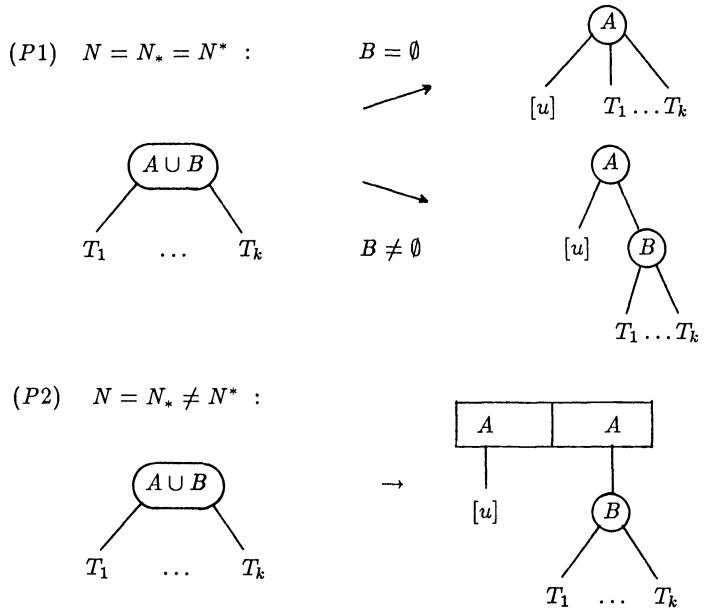


FIG. 4. *Templates for a leaf.*



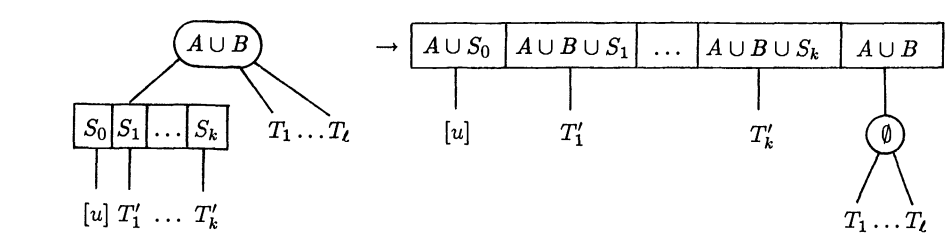


FIG. 5. *Templates for a P-node.*

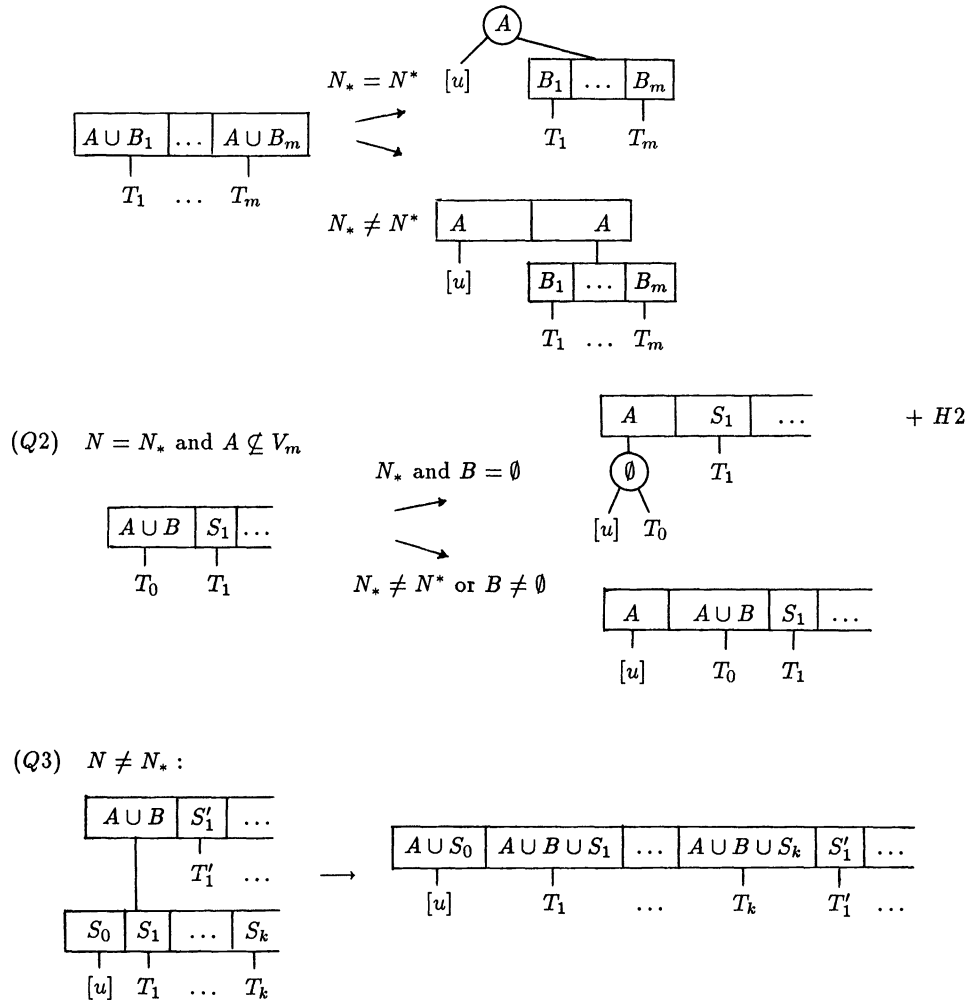


FIG 6. *Templates for a Q-node.*

$T(G)$ that contain only nodes with label 0. (This is an immediate consequence of Lemma 4.1.) $H1$ and $H2$ are help-templates that deal with special cases such as only one son, etc. They are displayed in Fig. 7. Note that template $(L1)$ is designed so as to also cover Case (a) in Lemma 4.2.

THEOREM 4.3. *The application of the templates along the path P from* N_* *to* N^* *correctly produces* $T(G+u)$.

Proof. It is easily seen that the templates cover all formally distinct cases. Also, all templates for the case $N_* = N^*$ and the help-templates $H1$ and $H2$ are easily shown to be correct.

Now consider the case $N_* \neq N^*$. Since N_* has the label 1 or ∞, the associated set A_* is nonempty. Similarly, the set B^* associated with N^* is nonempty. Let B^0 denote the set of vertices associated with nodes that are below N^* but not on P (which exist since N^* is not a leaf).

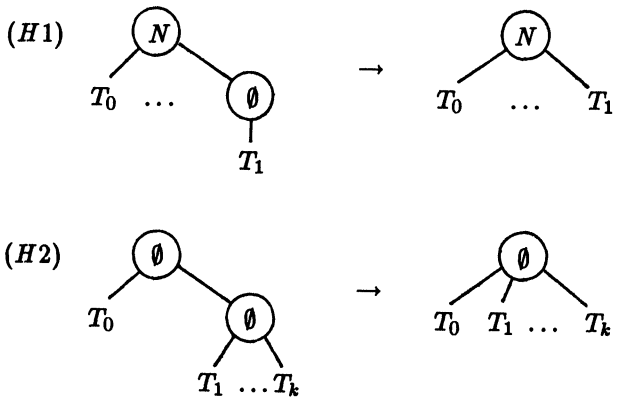


FIG. 7. *Help-templates* (H1) *and* (H2).

Let $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ denote the collections of maximal cliques of $G+u$ containing $A_* \cup \{u\}$, $A_* \cup B^*$, and $B^* \cup B^0$, respectively. Obviously, they must be represented by a Q-node in $T(G+u)$. Let $\bar{Q}$ be this Q-node. The first section of $\bar{Q}$ is the union of all sets A associated with the nodes or sections between N_* and N^*. Furthermore, $\bar{Q}$ contains an inner section S_N for each node N between N_* and N^* (including N_*, N^*), where S_N is the union of all sets A associated with nodes or sections between N and N^* on the path P. Other inner sections may occur by "gathering" sections from Q-nodes N between N_* and N^*. Finally, the last section has the associated vertex set $A^* \cup B^*$ of N^*.

The templates $L2$, $P2$, $Q1$ a) and $Q2$ a) (Figs. 4–6) construct the first part of this Q-node and return it as input for the next template. Inductively, the remaining templates on the path from N_* to N^* construct the complete Q-node. Although straightforward, the details are rather tedious, and therefore omitted. □

COROLLARY 4.4. *If $N_* \neq N^*$, then all nodes properly between N_* and N^* on the path P in $T(G)$ will become inner sections of a Q-node in $T(G+u)$.*

As an example, consider the interval graph in Fig. 3. Using the given LEXBFS, the following templates are applied (the notation is "added vertex: templates-added vertex," etc., the starting graph is the graph with vertex 1) 2: $L1$-3: $L1$-4: $L1$-5: $L1$-6: $L2$, $P3$, $H1$-7: $L1$-8: $Q2$-9: $P1$-10: $L2$, $P3$, $Q3$. This example also shows that while the Q-node $\bar{Q}$ of $T(G+u)$ is constructed, several sections may be equal or empty. This happens, e.g., when 10 is added and $P3$ is applied.

5. Implementation and complexity. In this section, we show that the above methods lead to an $O(|V|+|E|)$ algorithm for recognizing whether a given graph $G=(V, E)$ is an interval graph and for producing its associated MPQ-tree $T(G)$ if it is.

To achieve this complexity, we must use a good data structure for MPQ-trees. We assume that the children $N_1, \cdots, N_k$ of a P-node N are kept in a doubly linked circular list. Each N_i has a parent pointer to N, and N has a child pointer to one N_i. The sections $S_1, \cdots, S_m$ of a Q-node N have a pointer to their neighbor sections (so inner sections have two, while outer sections have only one). N has child pointers only to the outer sections S_1 and S_m, and only these have a parent pointer to N. Each section S_i has a pointer to its son. Each node has a parent pointer to its father (which is a P-node or a section of a Q-node). With each node or section, the associated set of vertices is given by a doubly linked circular list. Vertices contained in several sections

of a Q-node are included only in the lists belonging to the rightmost and the leftmost of these sections. Each vertex in a list has a pointer to the associated node or section. Vertex positions are given by an array with, for each vertex v, two vertex pointers pointing to the one or two list locations in which v occurs. Using this array, we have access to the node and/or sections containing a vertex v, the position of v in the associated list(s), and the type of the node in $O(1)$ time.

As was mentioned in § 3, a LEXBFS-ordering of G can be produced in $O(|V|+|E|)$ time. Consider now a labeling phase. For each $v \in \text{Adj}(u)$, we do the following:

A1. Find the associated node N (leaf, P-node or Q-node).
A2. If N is a Q-node, check whether one of the two sections pointed at is an outer section by looking at the neighbor pointers.
A3. Remove v from the element list associated with N or the outer section and insert it in a new list (that represents the set A).
A4. Put the node N or the outer section on a queue QU.

Altogether, this takes $O(|\text{Adj}(u)|)$ time. In particular, we have (in Step A2) already checked condition (b) of Lemma 4.1. If condition (b) is fulfilled we have two lists for each node or outer section with positive label: the new list representing the set A, and the modified old list representing the set B.

In order to check condition (a) of Lemma 4.1, we do the following until QU is empty:

B1. Delete N (a leaf, P-node or outer section) from the front of QU.
B2. If N is unmarked (initially all N on QU are unmarked), mark it and add its father (unmarked) at the rear of QU. Keep a pointer from the father to N.

When QU is empty, all nodes initially on QU and their ancestors have been marked and are kept (by the introduced pointers) in a rooted tree whose root is equal to the root of the MPQ-tree. Obviously, condition (a) of Lemma 4.1 is satisfied if and only if this rooted tree is a path, i.e., if and only if each node has at most one son. This can be checked by traversing it downward from the root in $O(1)$ time per node. Also in $O(1)$ time, we can check whether the associated set B is empty (i.e., the node/section has label ∞), and thus find N^*. Finally, N_* is the leaf of the rooted tree (if it is a path). Concerning the complexity, we have the following lemma.

LEMMA 5.1. *Testing whether condition* (b) *of Lemma* 4.1 *holds takes* $O(|\text{Adj}(u)|)$ *time. Testing whether condition* (a) *holds takes* $O(\#\ \text{nodes of}\ T) = O(|V|)$ *time if* (a) *is false, and* $O(n_0 + |\text{Adj}(u)|)$ *time if* (a) *is true, where* n_0 *denotes the number of nodes/sections with label* 0 *between* N_* *and* N^*.

Since the algorithm stops if (a) is false, the "bad" complexity of $O(|V|)$ occurs only once and thus does not destroy the overall complexity of $O(|V|+|E|)$ at which we aim. So in the sequel, we will assume that (a) and (b) always hold.

Our complexity analysis uses the notion of *amortized time* introduced in [10]. Its definition involves the assignment of a potential to the states that occur after certain steps of the algorithm. Here, a *step* is a joint labeling and updating phase. The *potential* after a step depends on the current MPQ-tree T and is defined as

$$\text{(5.1)}\quad \text{Pot}(T) := \#\ \text{nodes} + \#\ \text{outer sections} + \sum_{N \text{ is a leaf or a } P\text{ node}} 2 \cdot |V_N| + \sum_{S \text{ is an outer section}} |S|.$$

The *potential difference* $\Delta(s)$ of a step s is the difference of the potentials after the previous step and after s. Then the *amortized time* $AT(s)$ of step s is defined as $AT(s) = RT(s) + c \cdot \Delta(s)$, where $RT(s)$ is the *real time* required by step s and c is a

suitable positive constant (which will be specified later). In the beginning of the algorithm, the potential is 0; in the end it is nonnegative and equal to the sum of the $\Delta(s)$ over all steps. So summing up over all steps s, we obtain

$$\begin{aligned} \sum_s AT(s) &= \sum_s RT(s) + \sum_s \Delta(s) \\ &\geqq \sum_s RT(s) = \text{total real time.} \end{aligned} \tag{5.2}$$

Thus it suffices to consider only the amortized time per step s. Divide s into the labeling part s_1 and the updating part s_2. Then $RT(s_1) \leqq c_1 \cdot (n_0 + |\text{Adj}(u)|)$ with some constant c_1 because of Lemma 5.1, and $\Delta(s_1) = 0$ since the tree does not change in s_1. Now consider the updating part s_2 of step s.

The application of a template involves *recognition* of which template applies, *modification of the tree structure*, and an *update of the vertex sets* associated with tree nodes and sections.

The tree modification can obviously be carried out in $O(1)$ time per template.

Recognition requires knowing whether $N_* = N^*$, $N = N_*$, and, for $(Q1)$-$(Q3)$ whether or not $A \subseteq S_m$ and $B = \emptyset$. The first two tests can be done in constant real time per node N along the path, since N_* and N^* are known from the labeling phase. Testing whether $A \subseteq S_m$ can be done in $O(|A|)$ time by traversing the list representing A (which is available from the labeling phase) and using the array of vertices to check whether $u \in S_m$ for each $u \in A$. So for all sets A along the path in T, this takes real time proportional to $|\text{Adj}(u)|$. Finally, testing whether $B = \emptyset$ takes constant time per node, since B is available as list from the labeling phase.

Updating the vertex sets requires adjusting the vertex pointers from the vertices in $A \cup B$, and in cases in which a P-node is changed into a Q-node, making a copy of the sets A and B.

If possible, *adjustment of vertex pointers* is done in such a way that only vertices from A need to be considered (this is the case for $(L1)$, $(L2)$, $(P1)$, $(P2)$, and $(Q1)$). So in these cases, the real time for pointer adjustments is proportional to $|A|$, and thus to $|\text{Adj}(u)|$ along the path from N_* to N^*. In all other cases, the vertices from B are assigned to an inner section of the new Q-node. This reduces Pot (T) by $|B|$, since B was either contained in a leaf or P-node or in an outer section.

Similarly, copying all lists A along the path from N_* to N^* takes real time proportional to $|\text{Adj}(u)|$, while copying lists B also reduces the potential by $|B|$, since these vertices belong to inner sections of Q-nodes.

Altogether this shows that the real time per template can be subdivided into a constant amount c_1 plus an amount $c_2 \cdot |A|$ plus $c_3 \cdot |B|$. So along the path from N_* to N^*, this takes real time $c_2(n_0 + |\text{Adj}(u)|) + c_3 \cdot |\text{Adj}(u)| + c_4 \cdot |\bar{B}|$, where $\bar{B}$ is the union of all sets B for which vertex pointers must be adjusted or that must be copied. At the same time, the potential is reduced by $n_0 - 1 + |B|$, since all nodes properly between N_* and N^* become inner sections because of Corollary 4.4, and since the elements from B are moved from a P-node, leaf, or outer section into an inner section.

So choosing $c \geqq \max\{c_1 + c_2, c_3, c_4\}$, we obtain

$$\begin{aligned} AT(s) &= RT(s_1) + \Delta(s_1) + RT(s_2) + \Delta(s_2) \\ &\leqq (c_1 + c_2) \cdot (n_0 + |\text{Adj}(u)|) + c_3 \cdot |\text{Adj}(u)| + c_4 \cdot |\bar{B}| \\ &\quad - c \cdot (n_0 - 1) - c \cdot |\bar{B}| \\ &\leqq c \cdot (|\text{Adj}(u)| + 1). \end{aligned} \tag{5.3}$$

So altogether, we have the following.

THEOREM 5.2. *If $G+u$ is an interval graph, then constructing its MPQ-tree $T(G+u)$ from $T(G)$ takes $O(|\mathrm{Adj}\,(u)|+1)$ amortized time.*

COROLLARY 5.3. *Testing whether a given graph $G=(V, E)$ is an interval graph and constructing its MPQ-tree is possible in $O(|V|+|E|)$ time.*

Proof. The above complexity analysis gives Theorem 5.2. The total real time in the corollary is bounded by t_1+t_2, where t_1 is the time to recognize at some step that G is not an interval graph and where t_2 is the time required in the other steps. Lemma 5.1 yields $t_1=O(|V|)$, and (5.1), together with Theorem 5.2, yields

$$t_2 \leqq \sum_s AT(s) \leqq \sum_{u\in V} c\cdot(|\mathrm{Adj}\,(u)|+1)=O(|V|+|E|). \qquad \square$$

Concerning space requirement, the algorithm requires only $O(n)$ space if the current next vertex u in the LEXBFS-scheme and its adjacency set are given. We also note that LEXBFS requires only partial knowledge of the graph in the sense that unvisited parts may still change as the search continues. This implies that it is not necessary to know the whole graph in advance, but only portions "compatible" with LEXBFS.

Note that we can obtain an on-line, $O(n)$ space algorithm by allowing for more complicated templates. This is achieved in [6] at the cost of a higher time complexity of $O(n^2)$.

Acknowledgment. We thank the referees for pointing out several weaknesses in an earlier proof of Lemma 3.2 and for bringing the application to gate matrix layout problems to our attention.

REFERENCES

[1] S. BOOTH AND S. LUEKER, *Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms*, J. Comput. System Sci., 13 (1976), pp. 335–379.

[2] N. DEO, M. S. KRISHNAMOORTY, AND M. A. LANGSTON, *Exact and approximate solutions for the gate matrix layout problem*, IEEE Trans. Computer Aided Design, 6 (1987), pp. 79–84.

[3] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[4] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[5] ———, *Interval graphs and related topics*, Discrete Math., 55 (1985), pp. 113–121.

[6] N. KORTE, *Intervallgraphen und* MPQ-*Bäume: Algorithmen und Datenstrukturen zur Manipulation von Intervallgraphen und der Lösung von Seriationsproblemen*, Ph.D thesis, Report 87461, Institute for Operations Research, Universität Bonn, Federal Republic of Germany, 1987.

[7] N. KORTE AND R. H. MÖHRING, *Transitive orientation of graphs with side constraints*, in Proc. Workshop on Graphtheoretic Concepts in Computer Science 1985, H. Noltemeier, ed., Trauner, Linz, 1985, pp. 143–160.

[8] R. H. MÖHRING, *Algorithmic aspects of comparability graphs and interval graphs*, in Graphs and Order, I. Rival, ed., Reidel, Dordrecht, the Netherlands, 1985, pp. 41–101.

[9] D. J. ROSE, R. E. TARJAN, AND G. S. LUEKER, *Algorithmic aspects of vertex elimination of graphs*, SIAM J. Comput., 5 (1976), pp. 266–283.

[10] R. E. TARJAN, *Amortized computational complexity.* SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 306–318.

[11] R. E. TARJAN AND M. YANNAKAKIS, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs.* SIAM J. Comput., 13 (1984), pp. 566–579.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 82-94, February 1989

006

MINIMUM DELAY CODES*

LAWRENCE L. LARMORE†

Abstract. Huffman's algorithm finds a prefix-free binary code on a weighted alphabet which minimizes the expected length of the code string for a single symbol. A definition is given for the *expected delay* for a prefix-free binary code on a weighted alphabet: it is the expected time between a request to transmit the symbol and the completion of that transmission, assuming a channel with fixed capacity, where requests are queued. An $O(n^5)$-time $O(n^3)$-space *convex hull* algorithm is given that finds a code of minimal expected delay, where n is the size of the alphabet. It is conjectured that the algorithm has substantially lower time and space complexities in the worst case, and still lower in the average case.

The heart of the proof of polynomial time complexity is the *convex hull theorem*, which states that under certain conditions a binary tree that minimizes one penalty measure can be changed to a binary tree that minimizes a second penalty measure in a carefully controlled sequence of changes called *elementary shifts*.

Key words. coding, binary tree, convex hull, dynamic programming

AMS(MOS) subject classifications. 68P20, 68P25

1. Introduction. *Motivation.* This paper is the response to a question originated by Gopinath [3], and treated by Flores [1] in his dissertation. The classic Huffman coding problem is to find a prefix-free binary code[1] for a weighted alphabet, which minimizes the expected time to transmit a symbol. If the goal is to maximize information transmitted over a channel, the expected length of one code string is the correct penalty measure that needs to be minimized. But suppose that what is desired is to minimize the expected waiting time between the request to transmit a symbol and the completion of that transmission. We call that quantity *delay*, and it is the sum of two components: the time the symbol waits on a queue if the channel is busy, and the actual time to transmit the code. The expected delay $\bar{d}$ is thus the sum of two quantities: one is the expected length of a code string, and the other depends on the distribution of arrival times of requests. (We assume that the channel transmits bits at the rate of one per unit time.) If arrivals are independent, the arrival times have a Poisson distribution, and standard queueing theory techniques show that

$$\bar{d} = \frac{\lambda \bar{l}^2}{2(1-\lambda \bar{l})} + \bar{l}$$

where λ is the expected number of requests per unit time, $\bar{l}$ is the expected length of a code string, and $\bar{l}^2$ is the expected value of the square of the length of a code string [1]. Huffman's classic algorithm [6] finds a code that minimizes $\bar{l}$ in $O(n \log n)$ time, actually linear time if the frequencies are presorted [8]. Finding a code that minimizes expected delay is much harder. In fact, up until now, no polynomial-time algorithm was known.

Preliminaries. We let n be the size of the source alphabet, and ϕ_i the frequency of its ith symbol. The alphabet can first be sorted by frequency; thus we can assume

* Received by the editors October 7, 1987; accepted for publication (in revised form) April 7, 1988.

† California State University, Dominguez Hills, California. Present address, Department of Information and Computer Science, University of California, Irvine, California 92717.

[1] A code is *prefix-free* if no code string is a proper prefix of any other. The advantage of a code being prefix-free is that a coded message can be uniquely decoded. For example, if $a \to 0$, $b \to 11$, and $c \to 10$, the string 0100011100 can be uniquely decoded as *acaabca*.

$\phi_i \geqq \phi_{i-1}$. A prefix-free binary code corresponds to a binary tree[2] of size n. If $T = (l_1, \cdots, l_n)$ is such a tree, define

$$\bar{l} = \bar{l}(T) = \sum_{i=1}^{n} \phi_i l_i, \qquad \bar{l}^2 = \bar{l}^2(T) = \sum_{i=1}^{n} \phi_i l_i^2,$$

$$\bar{d} = \bar{d}(T) = \frac{\lambda \bar{l}^2(T)}{2(1-\lambda \bar{l}(T))} + \bar{l}(T).$$

The *minimum delay problem* is then to find a binary tree T for which $\bar{d}(T)$ is minimized, subject to the condition that $\lambda \bar{l}(T) < 1$.

Plotting trees in a plane. Only trees for which $l_i \geqq l_{i-1}$ need be considered, since a tree that does not satisfy that monotonicity condition can always be replaced by a tree at least as good which does (see, for example, [1]). Abstractly, the minimum delay problem can be solved geometrically. Plot each T in the plane as the point $(\bar{l}(T), \bar{l}^2(T))$, and let $\mathcal{P}$ be the set of all those plotted points. It is possible for two distinct trees T and T' to be plotted to the same point: if so, we say that T and T' are *equivalent.* Let

$$f(x, y) = \frac{\lambda y}{2(1-\lambda x)} + x.$$

The minimum delay tree T^{opt} then is plotted as a point $P^{\text{opt}} \in \mathcal{P}$ in the region $0 < x < 1/\lambda$, $y > 0$, for which f is minimized. The level curves of f are parabolic, with negative first derivative and positive second derivative in that region. If a line is drawn through P^{opt} tangent to the level curve of f through that point, all other points of $\mathcal{P}$ will lie on or above the level curve, and hence above the tangent. (See Fig. 1.)

The convex hull. A set $C \subseteq \mathbf{R}^2$ is said to be *convex* if the line segment joining any two points of C lies entirely within C. If $S \subseteq \mathbf{R}^2$ is any set, $\mathrm{CH}(S)$, the *convex hull*

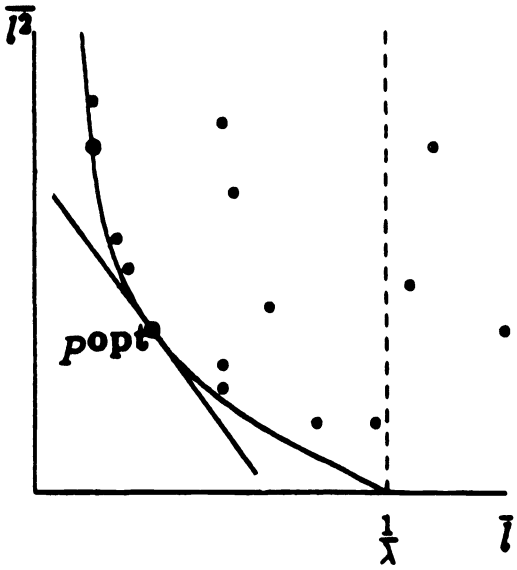


FIG. 1

[2] In this paper, a *binary tree* will be a full binary tree, i.e., each nonleaf node has exactly two children. Furthermore, a tree will be identified with the list consisting of the depths of its leaves, in left-to-right order.

of S, is defined to be the smallest convex set containing S. If S is finite, $\mathrm{CH}(S)$ is a polygon together with its interior.

Let $H = \mathrm{CH}(\mathcal{P})$. The boundary of H is a closed polygonal curve. One of the vertices of this boundary will be $P_{x\text{-min}}$, which we define to be the point of $\mathcal{P}$ for which x is minimized, and another is $P_{y\text{-min}}$, which we define to be the point of $\mathcal{P}$ for which y is minimized. (If there are two or more points of $\mathcal{P}$ for which x is minimized, the one with minimum y is taken to be $P_{x\text{-min}}$; this is similar for $P_{y\text{-min}}$.) Let B be the portion of the boundary of H that lies to the lower left of H, a polygonal arc joining $P_{x\text{-min}}$ and $P_{y\text{-min}}$. We call *extremal points* of $\mathcal{P}$ vertices of B, and one of them must be P^{opt}. (See Fig. 2.)

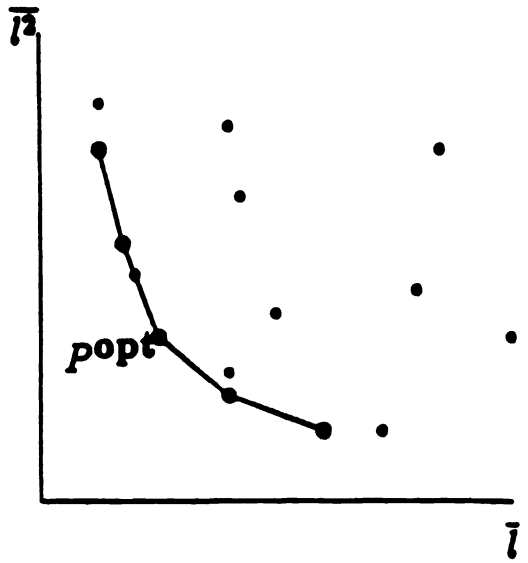


FIG. 2 *The polygonal path B.*

The convex hull algorithm finds all vertices of B and then simply determines which of them minimizes f by linear search. The time to find one vertex of B is $O(n^3)$, using a bottom-up approach similar to Garey's algorithm [2], and the number of vertices of B is $O(n^2)$ by the convex hull theorem given in § 5.

2. The convex hull algorithm. In this section, we give the details of the algorithm and the proof of correctness. Throughout, assume that a $\phi_1, \cdots, \phi_n$ are given, where ϕ_i is the frequency of the ith symbol of the source alphabet.

Outline of the method. The algorithm uses a subroutine we call *Find_Best*(α, β), which, for given $\alpha, \beta \geqq 0$, returns a tree T for which the quantity

$$\alpha \bar{l}(T) + \beta \bar{l}^2(T)$$

is minimized. If B has m vertices, all of those vertices may be found by executing *Find_Best* at most $4m+3$ times. Each call of *Find_Best* takes $O(n^3)$ time and uses $O(n^3)$ space, but the space can be reused for the next call.

The first step of the convex hull algorithm is to find a tree T_1 for which $\bar{l}(T)$ is minimized; either *Find_Best*$(1, 0)$ or Huffman's original algorithm can be used. The second step is to find a tree T_2 for which $\bar{l}^2(T)$ is minimized by calling *Find_Best*$(0, 1)$. The third step is a call to a recursive function *Span*, which produces a list of trees

whose plotted points in the graph include all vertices between two given points on the path B. Finally, linear search through those trees will locate T^{opt}. ("&" denotes concatenation of lists.)

THE CONVEX HULL ALGORITHM.
$T_1 \leftarrow$ *Find_Best*$(1, 0)$
$T_2 \leftarrow$ *Find_Best*$(0, 1)$
if T_1 and T_2 are equivalent **then**
    return T_1
else
    Candidate_List $\leftarrow (T_1)$ & *Span*(T_1, T_2) & (T_2)
    return some $T \in$ *Candidate_List* for which $\bar{d}(T)$ is minimized
end if

SPAN(T, T')
$\alpha = \bar{l}^2(T) - \bar{l}^2(T')$
$\beta = \bar{l}(T') - \bar{l}(T)$
$T'' \leftarrow$ *Find_Best*(α, β)
if $\alpha\bar{l}(T'') + \beta\bar{l}(T'') = \alpha\bar{l}(T) + \beta\bar{l}(T)$ **then**
    return *empty list*
else
    return *Span*(T, T'') & (T'') & *Span*(T, T'')
end if

Intuitively, *Span* works as follows. T is plotted above and to the left of T' in $\mathbf{R}^2$. Connect those two points by a line l, and find the lowest line l' parallel to l that meets $\mathcal{P}$. If $l' = l$, then T and T' both lie on B, and no vertices of B lie between them. Otherwise, some portion of B must meet l', and *Find_Best*(α, β) returns one point of B on l', although that point is not necessarily a vertex of B. (See Fig. 3.)

Correctness of the convex hull algorithm is guaranteed by the following lemma, whose proof we postpone to § 3:

LEMMA 2.1. (i) *Candidate_List contains all vertices of B.*

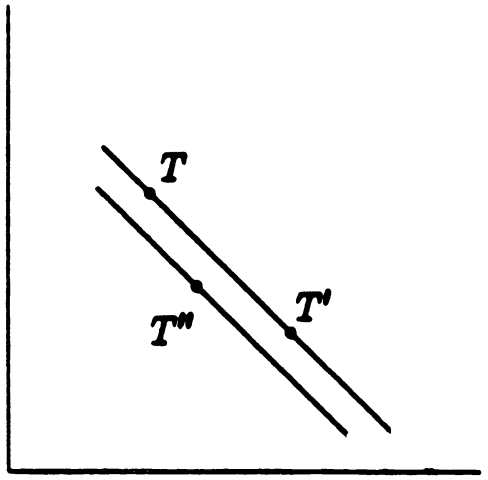


Fig. 3

(ii) *If m is the number of vertices of B, then the number of calls to Find_Best does not exceed $4m+3$.*

The following lemma is proved in § 5.

LEMMA 2.2. *If $\phi_i \leqq \phi_{i-1}$ for all i, the number of vertices of B does not exceed n^2.*

The time complexity of the convex hull algorithm is the time complexity of the subroutine *Find_Best*, multiplied by the number of calls of that subroutine. By Lemmas 2.1 and 2.2, the number of those calls is $O(n^2)$, if $\phi_i \leqq \phi_{i-1}$.

We now concentrate on the subroutine *Find_Best*(α, β). Let T be the tree that will be found by that subroutine, i.e., a tree for which α-β-*penalty*$(T) = \alpha \bar{l}(T) + \beta \bar{l}^2(T)$ is minimized.

The function g. For any pair i, j such that $1 \leqq i \leqq j \leqq n$, and for any $d \geqq 0$, we define $g(i, j, d)$ recursively, as follows:

$$g(i, i, d) = \alpha\phi_i d + \beta\phi_i d^2$$

$$\text{if } i < j, \text{ then } g(i, j, d) = \min\{g(i, k-1, d+1) + g(k, j, d+1) \mid i < k \leqq j\}.$$

If T has a subtree whose root is at depth d, and whose leaves are the ith through the jth leaves of T, then the contribution of that subtree to the α-β-penalty of T must be minimized, since T has minimum overall α-β-penalty. The recursive definition of g guarantees that $g(i, j, d)$ is that contribution. The α-β-penalty of T itself is $g(1, n, 0)$.

The values of $g(i, j, d)$ can then be computed dynamically, in order of increasing $j - i$. Since a subtree at depth d can have size at most $n - d$, g need only be computed for triples satisfying the condition $d \leqq n - j + i - 1$. There are $O(n^3)$ such triples, and (using linear search among all candidates for k) the time to compute $g(1, n, 0)$ is $O(n^4)$, and the space is $O(n^3)$.

Recovery of T. A system of pointers can be retained to recover the optimal tree T. If $i < j$, define $cut(i, j, d)$ to be that value of k for which the minimum is achieved in the recurrence portion of the definition of g. If there are multiple k's for which the minimum is achieved, $cut(i, j, d)$ is defined to be the smallest one. The values of *cut* are naturally computed during the dynamic algorithm, and if they are retained, the tree T can be recovered in $O(n)$ time.

The monotonicity lemma. Find_Best can actually be executed in $O(n^3)$ time by making use of the following lemma (similar to monotonicity lemmas used in [2] and [7], for example).

LEMMA 2.3 (Monotonicity Lemma). *If $j > i+1$, then*

$$cut(i, j-1, d) \leqq cut(i, j, d) \leqq cut(i+1, j, d).$$

We postpone the proof of the monotonicity lemma to § 4. The fact that the lemma reduces the computation time by an entire order of magnitude can be shown by an argument similar to that used by Knuth [7]. Let $N(i, j, d)$ be the number of candidates that need to be examined to determine $cut(i, j, d)$. The total time to execute *Find_Best* is dominated by the number of times such a candidate must be examined, which is

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sum_{d=0}^{n+j-i-1} N(i, j, d).$$

We have

$$N(i, i+1, d) = 1;$$

$$\text{if } j > i+1, \text{ then } N(i, j, d) = cut(i+1, j, d) - cut(i, j-1, d) + 1$$

by the monotonicity lemma. Substituting the values for $N(i, j, d)$ and canceling terms, we can see that the summation is $O(n^3)$.

Combining this result with Lemmas 2.1 and 2.2, we see that the convex hull algorithm has time complexity $O(n^5)$ if $\phi_i \leqq \phi_{i-1}$, since the search set for $cut(i, j, d)$ can be restricted to the range indicated by the monotonicity lemma.

3. Proof of Lemma 2.1. *Proof of Lemma* 2.1. We begin by making a claim, which we prove by induction.

CLAIM. *Suppose Span*(T, T') *is called and there are exactly* r *vertices of* B *that lie strictly between* T *and* T'. *Then Span*(T, T') *will find all those vertices, using at most* $4r+1$ *calls to Find_Best. Furthermore, at most* $4r-1$ *calls to Find_Best will be used if* $r>0$.

Proof of Claim. If $r=0$, one call to *Find_Best* will detect that there are no vertices of B strictly between T and T'. Now suppose $r>0$. Let T'' be the point returned by the top level call of *Find_Best* in the execution of *Span*(T, T'). Either T'' is itself a vertex of B, or lies in the interior of a line segment on B. Let r_0 be the number of vertices strictly between T and T'', and let r_1 be the number of vertices strictly between T'' and T'. We now consider the two cases. If T'' is a vertex of B, $r_0+r_1=r-1$. The number of calls already made is 1, the number of calls needed to find all vertices between T and T'' is at most $4r_0+1$, and the number of calls needed to find all vertices between T'' and T' is at most $4r_1+1$. Thus, the total number of calls does not exceed $4r-1$. On the other hand, suppose T'' lies in the interior of a line segment which is an edge of B. The endpoints of that edge must be vertices, both of which must be distinct from $\{T, T'\}$. It follows that $r_0>0$, $r_1>1$, and that $r=r_0+r_1$. The number of calls already made is 1, the number of calls needed to find all vertices between T and T'' is at most $4r_0-1$, and the number of calls needed to find all vertices between T'' and T' is at most $4r_1-1$. Thus, the total number of calls does not exceed $4r-1$.

The convex hull algorithm makes two initial calls to *Find_Best*. By the claim, no more than $4m+1$ additional calls are needed. This concludes the proof of Lemma 2.1.

4. The monotonicity lemma. Let g be the function introduced in § 2, with fixed choice of α and β. We first need a technical lemma.

LEMMA 4.1.

(i) $g(i, i, d+1)-g(i, i, d)=\phi_i(\alpha+(2d+1)\beta)$,

(ii) $g(i, j, d+1)-g(i, j, d)\geqq\phi_i(\alpha+(2d+1)\beta)$.

Proof. Equation (i) follows immediately from the basis of the definition of g. Equation (ii) is proved by induction on $j-i$, and the basis follows from (i). Now suppose that $j>i$. Let $k=cut(i, j, d+1)$. From the definitions of g and *cut*, we have

$$g(i, j, d+1)=g(i, k-1, d+2)+g(i, k-1, d+2), \tag{1}$$

$$g(i, j, d)\leqq g(i, k-1, d+1)+g(i, k-1, d+1), \tag{2}$$

while from the inductive hypothesis we have

$$g(i, k-1, d+2)-g(i, k-1, d+1)\geqq\phi_i(\alpha+(2d+3)\beta)\geqq\phi_i(\alpha+(2d+1)\beta), \tag{3}$$

$$g(k, j, d+2)-g(k, j, d+1)\geqq\phi_k(\alpha+(2d+3)\beta)\geqq 0. \tag{4}$$

Subtracting (2) from (1), then substituting (3) and (4), we conclude the proof of Lemma 4.1.

Next we need to show that the function g satisfies a "quadrangle condition," as defined by Yao [11].

LEMMA 4.2. *If* $i_0\leqq i_1\leqq j_0\leqq j_1$, *then*

$$g(i_0, j_0, d)+g(i_1, j_1, d)\leqq g(i_1, j_0, d)+g(i_0, j_1, d).$$

Proof. The proof is by induction on $j_0 - i_1$. We will consider the basis step last, since it is more complicated.

Suppose $i_1 < j_0$. Let $k_0 = cut(i_1, j_0, d)$ and let $k_1 = cut(i_0, j_1, d)$. By the symmetry of the problem, we can assume that $k_0 \leqq k_1$ without loss of generality. By the definitions of g and *cut*:

$$g(i_0, j_0, d) \leqq g(i_0, k_0 - 1, d+1) + g(k_0, j_0, d+1), \tag{1}$$

$$g(i_1, j_1, d) \leqq g(i_1, k_1 - 1, d+1) + g(k_1, j_1, d+1), \tag{2}$$

$$g(i_1, j_0, d) = g(i_1, k_0 - 1, d+1) + g(k_0, j_0, d+1), \tag{3}$$

$$g(i_0, j_1, d) = g(i_0, k_1 - 1, d+1) + g(k_1, j_1, d+1), \tag{4}$$

while by the inductive hypothesis:

$$g(i_0, k_0 - 1, d+1) + g(i_1, k_1 - 1, d+1) \leqq g(i_1, k_0 - 1, d+1) + g(i_0, k_1 - 1, d+1). \tag{5}$$

Add (1) and (2), add (3) and (4), and add the quantity $g(k_0, j_0, d+1) + g(k_1, j_1, d+1)$ to both sides of (5). The result follows by transitivity.

We now consider the basis case, namely $i_1 = j_0 = c$, which itself needs to be proved by induction on $j_1 - i_0$. The basis here, namely $i_0 = j_1$, is trivial. Otherwise, let $k = cut(i_0, j_1, d)$. By symmetry of the problem, we may assume, without loss of generality, that $k \leqq c$. By the definitions of g and of *cut*, we have

$$g(i_0, c, d) \leqq g(i_0, k - 1, d+1) + g(k, c, d+1), \tag{6}$$

$$g(i_0, j_1, d) = g(i_0, k - 1, d+1) + g(k, j_1, d+1), \tag{7}$$

while by Lemma 4.1

$$g(c, j_1, d+1) - g(c, j_1, d) \geqq g(c, c, d+1) - g(c, c, d). \tag{8}$$

By the inductive hypothesis

$$g(k, c, d+1) + g(c, j_1, d+1) \leqq g(c, c, d+1) + g(k, j_1, d+1). \tag{9}$$

Add $g(c, j_1, d)$ to both sides of (6). Subtract (8) from (9), add $g(i_0, k-1, d+1)$ to both sides, then substitute on the right side using (7). The result follows from transitivity.

5. The convex hull theorem. This section is devoted primarily to the statement and proof of the convex hull theorem, which is used to prove Lemma 2.2.

If f and g are weight functions that satisfy the hypotheses of the convex hull theorem, each tree can be plotted in a two-dimensional graph, using those two weight functions. The trees that lie along the lower left boundary of the convex hull of the graph of all trees are called (f, g)-*minimal.* The convex hull theorem states that consecutive trees along that boundary are obtained, one from the other, by a specific kind of change called an *elementary shift.* Under certain conditions that hold for the minimum delay problem, the theorem also limits the number of those trees. When we let the two weight functions be $\bar{l}^2$ and $\bar{l}$, the convex hull theorem yields an upper bound on the running time of the convex hull algorithm.

We begin by introducing a new representation of binary trees. Fix $n \geqq 1$ and $L \geqq 0$. We only consider trees $(l_1, \cdots, l_n)$ such that $l_i \leqq L$. We say that T is *monotone* if $l_i \geqq l_{i-1}$. If no depth restriction is desired, let $L = n - 1$.

Nodeset representation. Define a *node* to be an ordered pair (i, l) such that $i \in [1, n]$, which is called the *index* of the node, and $l \in [1, L]$, which is called the *level* of the node. Any set of nodes we call a *nodeset.* If T is a tree, define

$$nodeset(T) = \{(i, l) \mid 1 \leqq l \leqq l_i\}$$

where l_i is the depth of the ith leaf of T. For example, if T is the tree shown in Fig. 4(a), *nodeset*(T) is the set shown in Fig. 4(b).

Width and rank. If (i, l) is any node, define

$$width(i, l) = 2^{-l}, \qquad rank(i, l) = iL - l + 1.$$

If A is a nodeset, *width*(A) will be the sum of the widths of its constituent nodes.

Properties of nodesets. If A is any nodeset, we have the following:

(a) A is *monotone* if $(i, l) \in A$ implies $(i+1, l) \in A$;

(b) A is *proper* if $(i, l) \in A$ implies $(i, l-1) \in A$;

(c) A is a *virtual treeset* if $width(A) = n - 1$;

(d) A is a *treeset* if $A = nodeset(T)$ for some tree T.

The Kraft equality[3] implies that every treeset is a virtual treeset, and it can be shown (see, for example, [4]) that if A is a monotone proper virtual treeset, it is the treeset of some monotone tree.

Weight functions. We define a *weight function* to be any positive function

$$f: [1, n] \times [1, L] \to \Lambda$$

where Λ is any ordered integral domain.[4]

If A is any nodeset, define $f(A)$ (which we call the *f-weight* of A) to be the sum of the weights of the members of A, and if T is a tree, define $f(T) = f(nodeset(T))$. We have the following:

(a) f is *monotone* if $f(i+1, l) \leqq f(i, l)$;

(b) f is *proper* if $f(i, l-1) \leqq f(i, l)$;

(c) A is *f-minimal* if there is no nodeset of width equal to A of smaller f-weight.

To emphasize the fact that minimality is a relative term, we shall sometimes say, for clarity, that A is "f-minimal among nodesets of width w."

Example. Let $\Phi = (\phi_1, \cdots, \phi_n)$ be a frequency list. Define

$$\bar{l}_\Phi(i, l) = \phi_i, \qquad \bar{l}^2_\Phi(i, l) = \phi_i(2l - 1).$$

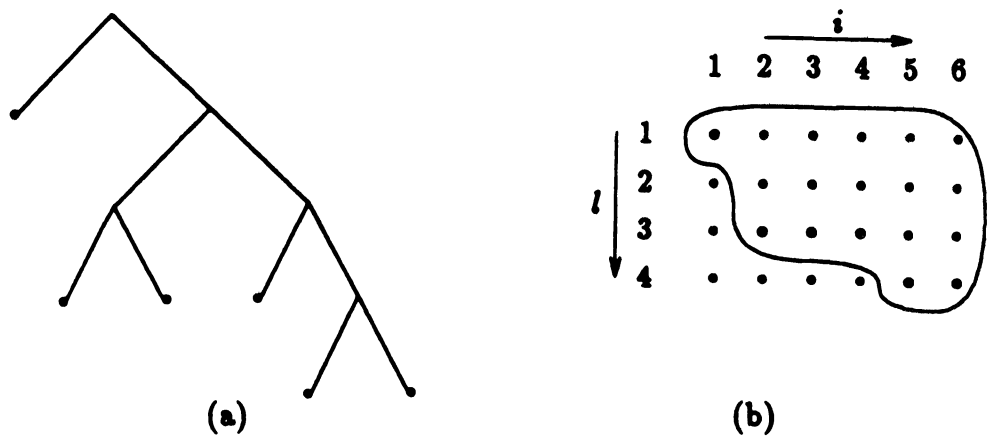


FIG. 4. (a) T. (b) *nodeset*(T).

[3] That is, $\sum_{i=1}^{n} 2^{-l_i} = 1$.

[4] That is, Λ has addition, subtraction, and multiplication; multiplicative identity; and satisfies the usual commutative, associative, and distributive laws. Furthermore, Λ is ordered, and $a < b$ implies $a + c < b + c$, also $ac < bc$ if $c > 0$. Two well-known examples are $\mathbf{R}$ = *reals* and $\mathbf{Z}$ = *integers*. See [10], for example, for a discussion of integral domains.

These weight functions are simply the expected path length and expected square path length defined in the Introduction. Furthermore, $\bar{l}_\Phi$ and $\bar{L}^2_\Phi$ are proper, and they are monotone if Φ is monotone (i.e., $\phi_i \leqq \phi_{i-1}$).

Pairs of weight functions. Suppose f and g are weight functions, taking values in the same ordered integral domain Λ. We say that a weight function h is a *positive linear combination* of f and g if there exist nonnegative $\alpha, \beta \in \Lambda$, not both zero, such that $h(i, l) = \alpha f(i, l) + \beta g(i, l)$ for all (i, l). We say that a nodeset is *(f, g)-minimal* if it is h-minimal for some h, which is a positive linear combination of f and g. For clarity, we shall sometimes say "A is (f, g)-minimal among nodesets of width w."

Flatness. We say that a weight function g is *flatter* than another weight function f provided $l' > l$ if the following inequality holds:

$$g(i, l)f(i', l') \leqq f(i, l)g(i', l').$$

For example, for any frequency list Φ, $\bar{l}^2_\Phi$ is flatter than $\bar{l}_\Phi$. (The word "flatter" is used because typically a tree that is g-minimal will have more equally deep leaves than a tree that is f-minimal.)

Elementary shifts. If A and A' are nodesets, we say that *A' is obtained from A by an elementary shift*, and write $A \Rightarrow A'$, if the following two conditions hold:

$$\mathit{width}(A') = \mathit{width}(A), \qquad A' - A \text{ consists of exactly one node.}$$

THEOREM 5.1 (convex hull theorem). *Let f, g be weight functions, taking values in a multiplicative ordered domain Λ, and let $0 \leqq w \leqq n(1-2^{-L})$ be an integer. Then there exists a sequence of nodesets $A_0, \cdots, A_K$, each of width w, such that we have the following*:

(1) $f(A_{k-1}) \leqq f(A_k)$;

(2) $g(A_{k-1}) \geqq g(A_k)$;

(3) *A_0 is f-minimal among nodesets of width w*;

(4) *A_K is g-minimal among nodesets of width w*;

(5) *A_k is (f, g)-minimal among nodesets of width w*;

(6) *If h is any positive linear combination of f and g, then there is some A_k that is h-minimal among nodesets of width w*;

(7) *Either $A_{k-1} \Rightarrow A_k$ or $A_k \Rightarrow A_{k-1}$.*

Furthermore, if f and g are both monotone and proper, then

(8) *A_k is monotone and proper.*

Furthermore, if g is flatter than f, then

(9) $A_{k-1} \Rightarrow A_k$;

(10) $K \leqq nL$.

We now prove Lemma 2.2. Let $\Lambda = \mathbf{R}$, $L = n-1$, $w = n-1$, $f = \bar{l}_\Phi$, and $g = \bar{l}^2_\Phi$. By (6), each tree whose plot is a vertex of B is equivalent to some tree whose nodeset is some A_k. Since $K \leqq n^2 - n$, we are done.

We will first prove the convex hull theorem under the additional hypothesis that the two weight functions satisfy a general position condition. Then we will reduce the general convex hull theorem to this special case by making use of infinitesimals.

General position. We say that a weight function f is in *general position* if no two nodesets have the same f-weight, and we say that f is in *almost general position* if no three nodesets have the same f-weight. If f and g are weight functions, we say that f and g are in *jointly general position* if f and g are each in general position and if each positive linear combination of f and g is in almost general position.

The concept of almost general position has a simple geometric interpretation. Graph each nodeset A to the point $(f(A), g(A))$. Then f and g are in jointly general position if no two nodesets are graphed to points in the same vertical or the same horizontal line, and no three nodesets are graphed to collinear points.

LEMMA 5.2. *If all the hypotheses of Theorem* 5.1 *are satisfied, and if f and g are in jointly general position, then the conclusions of Theorem* 5.1 *hold.*

We now begin the proof of Lemma 5.2. Let $A_0, \cdots, A_K$ be the set of all (f, g)-minimal nodesets, sorted in order of increasing f-weight. Then (1), (5), and (6) automatically hold. If $g(A_{k-1}) < g(A_k)$, then A_k could not be (f, g)-minimal, so (2) holds. Letting $\alpha = 1$ and $\beta = 0$, we obtain (3). Letting $\alpha = 0$ and $\beta = 1$, we obtain (4).

To prove (7), we need two lemmas.

LEMMA 5.3. *Let* $\alpha = g(A_{k-1}) - g(A_k)$, let $\beta = f(A_k) - f(A_{k-1})$, *and let* $h = \alpha f + \beta g$. *Then* A_{k-1} *and* A_k *are both h-minimal, and are the only two h-minimal nodesets.*

Proof. Since $h(A_{k-1}) = h(A_k)$, either both are h-minimal or neither is. Suppose neither is. Then there must exist an h-minimal nodeset. By (6), that h-minimal nodeset must be A_j for some j. By (5), A_k is h'-minimal for some positive linear combination $h' = \alpha' f + \beta' g$. We have three inequalities:

Inequality 1: $h(A_k) > h(A_j)$, since A_j is h-minimal and A_k is not;
Inequality 2: $h'(A_{k-1}) \geqq h'(A_k)$, since A_k is h'-minimal;
Inequality 3: $h'(A_j) \geqq h'(A_k)$, since A_k is h'-minimal.

The remainder of the proof is tedious but straightforward. First, multiply both sides of Inequality 1 by β', both sides of Inequality 2 by $f(A_j) - f(A_k)$, and both sides of Inequality 3 by β. Add the left sides and the right sides of the three resulting inequalities to obtain a single inequality, which will be strict, since Inequality 1 is strict. Replace each instance of α with $g(A_{k-1}) - g(A_k)$, and each instance of β with $f(A_k) - f(A_{k-1})$. There will be 12 terms on each side, all of which cancel. Thus we obtain $0 < 0$, a contradiction. The assumption that $j < k - 1$ leads to a similar contradiction. Finally, note that h must be in almost general position; thus A_{k-1} and A_k are the only h-minimal nodesets. This concludes the proof of Lemma 5.3.

LEMMA 5.4. *If A is any nodeset of width at least* 2^{-l}, *and if the level of every element of A is at least l, then there exists a nodeset* $B \subseteq A$ *such that* $width(B) = 2^{-1}$.

Proof. The proof is by backward induction on l. If $l = L$, every element of A has level L, and we can let B consist of just one of those elements. Otherwise, we consider two cases. The first case is that A contains a node of level l. Let B consist of just that node. In the second case, A has no node of level l, i.e., all of its elements have level at least $l+1$. By the inductive hypothesis, we can pick $B_1 \subseteq A$ and $B_2 \subseteq A - B_1$, each of width $2^{-(l+1)}$. Let $B = B_1 \cup B_2$. This completes the proof of Lemma 5.4.

We now prove (7). Let h be as in the statement of Lemma 5.3. Let $B = A_k - A_{k-1}$, and $B' = A_{k-1} - A_k$. If either B or B' consists of a single node, we are done by definition of elementary shift. Otherwise, let l be the smallest level of any node in either B or B', i.e., 2^{-l} is the greatest width of any node in B or B'. Note that $width(B) = width(B') > 2^{-l}$. By Lemma 5.4, we can pick nodesets $C \subseteq B$ and $C' \subseteq B'$ each of width 2^{-l}. Let $A = A_{k-1} \cup C - C'$, and let $A' = A_k \cup C' - C$. The combined h-weight of A and A' must be equal to the combined h-weight of A_{k-1} and A_k, because exactly the same nodes appear in the summations. On the other hand, it must be greater by Lemma 5.3, a contradiction. Thus (7) is established.

For the remainder of the proof of Lemma 5.2, assume that f and g are monotone and proper.

LEMMA 5.5. *If A is any nodeset of width* $q2^{-l} + r$, *where q is an integer and* $0 \leqq r < 2^{-l}$, *then there exists a nodeset* $B \subseteq A$ *such that* $width(B) = q2^{-l}$.

Proof. The proof is by induction on r, which is necessarily an integral multiple of 2^{-L}. If $r = 0$, let $B = A$. Otherwise, let $(i, l') \in A$ have the greatest level, i.e., smallest width. Then apply the inductive hypothesis to $A - \{(i, l')\}$, and Lemma 5.5 is proved.

Suppose that A_k is not monotone. Then there exists $(i, l) \in A_k$ such that $(i+1, l) \notin A_k$. Let $A = A_k \cup \{(i+1, l)\} - \{(i, l)\}$. By monotonicity and general position of f and g, A has smaller f-weight and also smaller g-weight than A_k. Thus, A_k cannot be (f, g)-minimal, which contradicts (5). Suppose that A_k is not proper. Then there exists $(i, l) \in A_k$ such that $(i, l-1) \notin A_k$. Let $A = A_k \cup \{(i, l-1)\} - \{(i, l)\}$. By Lemma 5.5, there exists a nodeset $B \subseteq A$ of width w. Since f and g are proper and in general position, B has smaller f-weight and smaller g-weight than A_k, so A_k cannot be (f, g)-minimal, which contradicts (5). Thus (8) holds.

For the remainder of the proof of Lemma 5.2, we assume that g is flatter than f.

By (7), either $A_{k-1} \Rightarrow A_k$ or $A_k \Rightarrow A_{k-1}$. If the former holds, we are done. If the latter holds, we will obtain a contradiction. Let (i, l) be the unique member of $A_{k-1} - A_k$. Let $B = A_k - A_{k-1}$. We first note that, for any $(i', l') \in B$, $l' \geqq l$, since B must have width 2^{-l}. Since A_{k-1} and A_k are monotone, $l' > l$.

By (1), we have

Inequality 1: $f(i, l) < \sum_{(i', l') \in B} f(i', l')$

while by (2), we have

Inequality 2: $g(i, l) > \sum_{(i', l') \in B} g(i', l')$.

Since g is flatter than f, $g(i, l) f(i', l') \leqq f(i, l) g(i', l')$ for all $(i', l') \in B$. Multiplying Inequality 1 by $g(i, l)$ and Inequality 2 by $f(i, l)$, we obtain a contradiction. Thus (9) is established.

Now, by (9), $A_k - A_{k-1}$ always consists of a single node, say (i, l). If $A_{k-1} - A_k$ consists of a single node, that would imply that $A_k \Rightarrow A_{k-1}$, which was shown to be impossible in the previous paragraph. Thus, A_{k-1} has greater cardinality than A_k, and $|A_0| \geqq |A_K| + K$. Since there are only nL nodes altogether, $|A_0| \leqq nL$. Inequality (10) follows immediately.

This concludes the proof of Lemma 5.2.

Proof in the general case. We now reduce the convex hull theorem to Lemma 5.2, i.e., to the special case where the weight functions are in relatively general position.

The polynomial domain. Let x and y be formal symbols, and let $\Lambda[x, y]$ be the polynomials with coefficients in Λ over x and y. We make $\Lambda[x, y]$ an ordered integral domain by the rules that y is infinitesimal, and x is infinitesimally smaller than y. That is, if p, q, r, s are nonnegative integers, and if λ, μ are positive elements of Λ, then $0 < \lambda x^p y^q < \mu x^r y^s$ if and only if one of the following holds: either $q > s$, or $q = s$ and $p > r$, or $q = s$ and $p = r$ and $\lambda < \mu$.

Degrees. If $\lambda \in \Lambda$, we say that the term $\lambda x^p y^q$ has *x_degree p* and *y_degree q*. Any nonzero $\alpha \in \Lambda[x, y]$ consists of the sum of unlike nonzero terms. We define the *x_y_degree* of α to be the maximum *x_degree* of a term of α of maximal *y_degree*. For example, *x_y_degree*$(x^5 + x^3 y^2 + x^4 y^2) = 4$. We note that, for nonzero $\alpha, \beta \in \Lambda[x, y]$:

$$x\_y\_degree(\alpha\beta) = x\_y\_degree(\alpha) + x\_y\_degree(\beta).$$

Proof of Theorem 5.1. We define weight functions $f^{[x]}$ and $g^{[y]}$, by taking values in $\Lambda[x, y]$, by

$$f^{[x]}(i, l) = f(i, l) + x^{n(L-l)+i}, \qquad g^{[y]}(i, l) = g(i, l) + y^{n(L-l)+i}.$$

Intuitively, $f^{[x]}$ and $g^{[y]}$ differ from f and g "infinitesimally." The differences are not enough to disturb relations among nodesets using f and g, but are used only to break enough ties to ensure general position.

We refer to the quantity $n(L-l)+i$ as the *degree* of the node (i, l). Each node has a distinct degree in the range $[1, nL]$.

LEMMA 5.6. *The weight functions $f^{[x]}$, $g^{[y]}$ are jointly in general position.*

Proof. Since no two nodes have the same degree both $f^{[x]}$ and $g^{[y]}$ are (individually) in general position. Now suppose that there are three distinct nodesets, A, B, and C of the same h-weight, where $h = \alpha f^{[x]} + \beta g^{[y]}$ and $\alpha, \beta \in \Lambda[x, y]$ are not both zero. Now let r be the highest degree that serves to distinguish between some two of those three nodesets, i.e., any node of degree higher than r lies in either all or none of the three sets A, B, and C, while the node of degree r lies in either one or two, but not in three, of those same sets.

Without loss of generality, A and B both contain the node of degree r while C does not, or C contains the node of degree r while A and B do not. Let $s < r$ be the highest degree of a node that lies in A but not in B, or in B but not in A. Since A and B have the same h-weight, we have

Equation 1: $\alpha(f^{[x]}(A) - f^{[x]}(B)) = \beta(g^{[y]}(B) - g^{[y]}(A))$,

while, since A and C have the same h-weight, we have

Equation 2: $\alpha(f^{[x]}(A) - f^{[x]}(C)) = \beta(g^{[y]}(C) - g^{[y]}(A))$.

Let $a = x\_y\_degree(\alpha)$ and $b = x\_y\_degree(\beta)$. From Equation 1, we have $a + s = b + 0$, while from Equation 2, we have $a + r = b + 0$, a contradiction. Thus, Lemma 5.6 is proved.

The difference between the f-weight of a node and the $f^{[x]}$-weight of that same node is infinitesimal in Λ, i.e., it cannot be made larger than any positive element of Λ by multiplication by any element of Λ. Thus, if a nodeset A is $f^{[x]}$-minimal it is also f-minimal. Similarly, if A is $g^{[y]}$-minimal it is also g-minimal, and if it is $(f^{[x]}, g^{[y]})$-minimal, it is also (f, g)-minimal. We also observe the following:

(a) If f is monotone, then $f^{[x]}$ is montone;
(b) If g is monotone, then $g^{[y]}$ is monotone;
(c) If f is proper, then $f^{[x]}$ is proper;
(d) If g is proper, then $g^{[y]}$ is proper;
(e) If g is flatter than f, then $g^{[y]}$ is flatter than $f^{[x]}$.

Thus, $f^{[x]}$ and $g^{[y]}$ satisfy the hypotheses of Lemma 5.2, and if $A_0, \cdots, A_K$ is a list of nodes which satisfies the conclusion of Lemma 5.2 for the inputs $f^{[x]}$ and $g^{[y]}$, it also satisfies the conclusion of Theorem 5.1 for the inputs f and g.

This concludes the proof of Theorem 5.1, the convex hull theorem.

6. Open questions. *Improvements.* When we use the methods of [8], the time to execute *Find_Best* should be reduced from $O(n^3)$ to roughly $O(n^{5/2} \log n)$. Such a result will be unimportant, though, if the conjectures given below hold.

CONJECTURE 1. The time for the convex hull algorithm can be reduced to $O(n^4 \log n)$ by dynamically computing, for all pairs (i, j), in order of increasing $j - i$, $g(i, j, 0)$ for all possible choices of α and β, and by making use of the monotonicity lemma, Lemma 2.3.

CONJECTURE 2. The number of vertices of B is $O(n^{1/2})$ in the average case.

Conjecture 2 is suggested by a statistical result by Renyi and Sulanke [9]. The Renyi–Sulanke result states that the number of points in the boundary of the convex hull of a randomly selected set of N points in the plane is expected to be $\Theta(\log^{1/2} N)$, if the points are placed according to a normal distribution. In our application, if N is the number of trees of size n, then $\log N = \Theta(n)$. We expect that it will be quite

difficult to show that Conjecture 2 holds, but simulations indicate just such a bound with random data.

The most optimistic conjecture, which combines the best features of Conjectures 1 and 2, follows.

CONJECTURE 3. The convex hull algorithm can be modified to run in $O(n^{5/2} \log n)$ time in the average case.

Again, simulations support Conjecture 3.

Alphabetic codes. We say that a prefix-free code is *alphabetic* if the source alphabet has a given order, and the lexical order of the code strings must correspond to that order. The example code given in the Introduction is not alphabetic, but the following code is:

$$a \to 0$$
$$b \to 10$$
$$c \to 11.$$

Hu and Tucker [5] present an algorithm that minimizes $\bar{l}$ for a prefix-free alphabetic code. The convex hull algorithm presented in this paper correctly finds an alphabetic code of minimal expected delay, since at no time was it necessary to assume that the ϕ_i are monotone during the proof of correctness. However, there is no guarantee that the algorithm will take polynomial time, since Lemma 2.2 strongly depends on monotonicity of the weights. The question of whether a polynomial-time algorithm exists for finding an alphabetic prefix-free code of minimal delay remains open.

Dynamic and parallel computation. The algorithm presented in this paper is sequential, and finds a minimum delay code for given (static) frequencies. Instead we could ask for a dynamic algorithm that changes the code as frequencies are updated during the transmission of a message. Either the static or the dynamic code could be computed using parallel processors; the question then would be to find an efficient tradeoff between the number of processors and time. We could also ask whether the problem of finding a minimum delay code is in class *NC*, the class of problems that can be solved in polylogarithmic time using polynomially many processors.

Acknowledgment. I wish to thank George Lueker for calling my attention to this problem.

REFERENCES

[1] C. FLORES, *Encoding of bursty sources under a delay criterion*, Ph.D. thesis, University of California, Berkeley, CA, 1983.

[2] M. R. GAREY, *Optimal binary search trees with restricted maximal depth*, SIAM J. Comput., 3 (1974), pp. 101-110.

[3] B. GOPINATH, private communication.

[4] T. C. HU AND K. C. TAN, *Path length of binary search trees*, SIAM J. Appl. Math., 22 (1972), pp. 225-234.

[5] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable length alphabetic codes*, SIAM J. Appl. Math., 21 (1971), pp. 514-532.

[6] D. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. Inst. Radio Engineers, 40 (1952), pp. 1098-1101.

[7] D. E. KNUTH, *Optimum binary search trees*, Acta Informatica, 1 (1971), pp. 14-25.

[8] L. L. LARMORE, *Height-restricted optimal binary trees*, SIAM J. Comput., 16 (1987), pp. 1115-1123.

[9] A. RENYI AND R. SULANKE, *Uber die konvexe Hulle von n zufallig gewahlten Punkten*, Z. Wahrsch. Verw. Gebiete., 2 (1963), pp. 75-84.

[10] B. L. VAN DER WAERDEN, *Modern Algebra*, Frederick Ungar, New York, 1950.

[11] F. F. YAO, *Efficient dynamic programming using quadrangle inequalities*, in Proc. 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 429-435, 1362-1371.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 95-111, February 1989

007

THE BOOLEAN HIERARCHY II: APPLICATIONS*

JIN-YI CAI†, THOMAS GUNDERMANN‡, JURIS HARTMANIS§, LANE A. HEMACHANDRA¶, VIVIAN SEWELSON[1], KLAUS WAGNER[2], AND GERD WECHSUNG‡

Abstract. *The Boolean Hierarchy* I: *Structural Properties* [J. Cai et al., *SIAM J. Comput.*, 17 (1988), pp. 1232-1252] explores the structure of the boolean hierarchy, the closure of NP with respect to boolean operations. This paper uses the boolean hierarchy as a tool with which to extend and explain three important results in structural complexity theory.

(1) Hartmanis, Immerman, and Sewelson [*Proc. 15th Annual Symposium on the Theory of Computation*, 1983, pp. 382-391] showed that E = NE if and only if NP − P contains no sparse sets. In this paper it is shown that this reflects a behavior of the boolean hierarchy. When E = NE, sparse sets fall from alternate levels of the boolean hierarchy (Fig. 2(a)). Furthermore, it is shown that capturable sets (i.e., subsets of sparse NP sets) are banished from the boolean hierarchy when E = NE: E = NE implies that BH − P has no capturable sets.

(2) Counting classes are natural candidates as complete languages for the levels of the boolean hierarchy. The authors show that in relativized worlds counting classes are *not* complete for the levels of the boolean hierarchy. Relatedly, the work of Blass and Gurevich [*Inform. and Control*, 55 (1982), pp. 80-88] is extended and it is concluded that counting classes are weak in some relativized worlds.

(3) Karp and Lipton [*Proc. 12th Annual Symposium on the Theory of Computation*, 1980, pp. 302-309] showed that if NP has a sparse oracle (i.e., if there is a sparse set S so $NP \subseteq P^S$; equivalently, if NP has small circuits), then the polynomial hierarchy collapses to NP^{NP}. The authors demonstrate that this cannot be much improved. There is a relativized world in which NP has a sparse oracle, yet the boolean hierarchy is infinite. Thus no proof that relativizes can show: NP has a sparse oracle implies that the polynomial hierarchy equals the boolean hierarchy.

The results of this paper present new ideas and techniques, and put previous results about NP and D^P in a richer perspective. Throughout, the emphasis is on the structure of the boolean hierarchy and its relations with more common classes.

Key words. sparse sets, counting classes, circuits, boolean hierarchy, structural complexity theory, polynomial-time hierarchy, relativized complexity classes

AMS(MOS) subject classifications. 68Q15, 03D15

1. Introduction and overview of results.

1.1. Introduction and background.

The boolean hierarchy I: *Structural properties* [CGI] explored the structural properties of the boolean hierarchy [W85a]. The hierarchy defined in that paper is used here to extend three central results of structural complexity theory.

* Received by the editors October 29, 1986; accepted for publication (in revised form) March 4, 1988. Preliminary versions of portions of this paper were presented at the 1985 International Conference on Fundamentals of Computation Theory [W85a] and the 1986 Structure in Complexity Theory Conference [CH86], [CH85].

† Computer Science Department, Cornell University, Ithaca, New York 14853. Present address, Department of Computer Science, Yale University, New Haven, Connecticut 06520. The research of this author was supported by a Sage Fellowship and National Science Foundation grants DCR-8301766, DCR-850597, and CCR-8709818.

‡ Mathematics Section, Friedrich-Schiller University, Jena, German Democratic Republic.

§ Computer Science Department, Cornell University, Ithaca, New York 14853. The research of this author was supported by National Science Foundation grants DCR-8301766 and DCR-852097.

¶ Computer Science Department, Cornell University, Ithaca, New York 14853. Present address, Department of Computer Science, Columbia University, New York, New York 10027. The research of this author was supported by a Fannie and John Hertz Fellowship, National Science Foundation grants DCR-8301766, DCR-8520597, and CCR-8809174, and a Hewlett-Packard Corporation equipment grant.

[1] Department of Mathematics and Computer Science, Dartmouth College, Hanover, New Hampshire 03755.

[2] Institute for Mathematics, University of Augsburg, Augsburg, Federal Republic of Germany.

D^P, the closure of NP∪coNP under intersections, has recently been carefully studied [PY82], [PW85], [CM85]. A natural completion of such extensions of NP is the boolean hierarchy—the closure of NP under boolean operations. The boolean hierarchy (BH) has a clear "physical" interpretation. The sets in the boolean hierarchy are exactly those representable by *hardware over* NP. Each boolean hierarchy language is accepted by a hardware tree connecting NP machines (Fig. 1).

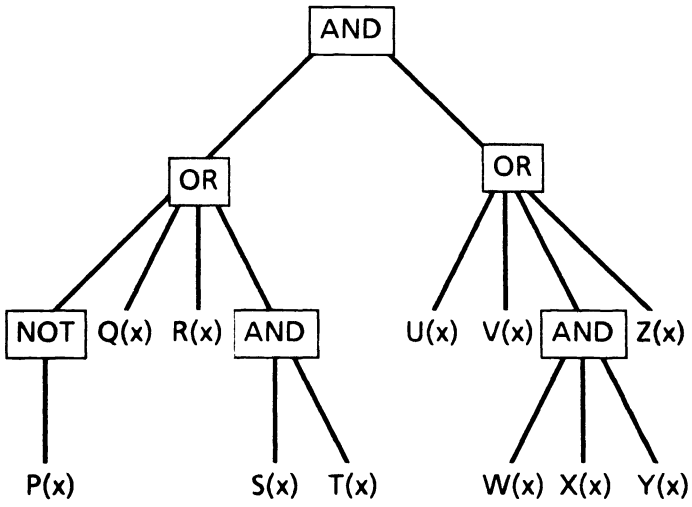


FIG. 1. *Hardware over* NP.

Each level of the boolean hierarchy consists of sets represented by a certain fixed structure of boolean operators on NP sets. Sets in the boolean hierarchy have normal forms. Each can be written as a finite union of D^P sets:

$$\mathrm{NP}(0) = \mathrm{P},$$
$$\mathrm{NP}(1) = \mathrm{NP},$$
$$\mathrm{NP}(2) = \mathrm{D}^{\mathrm{P}} = \{L_1 \cap \bar{L}_2 \mid L_1, L_2 \in \mathrm{NP}\},$$
$$\mathrm{NP}(3) = \{(L_1 \cap \bar{L}_2) \cup L_3 \mid L_1, L_2, L_3 \in \mathrm{NP}\},$$
$$\mathrm{NP}(4) = \{((L_1 \cap \bar{L}_2) \cup L_3) \cap \bar{L}_4 \mid L_1, L_2, L_3, L_4 \in \mathrm{NP}\},$$
$$\mathrm{BH} = \bigcup_{i>0} \mathrm{NP}(i),$$
$$\mathrm{coNP}(i) = \{S \mid \bar{S} \in \mathrm{NP}(i)\}.$$

LEMMA [CGI]. *The following are equivalent*:

(1) $T \in \mathrm{BH}$.

(2) *T is a finite union of* D^{P} *sets.*

(3) *T is a finite intersection of* $\mathrm{coD}^{\mathrm{P}}$ *sets.*

(4) *T is in the boolean closure of* NP.

(5) $T \leqq^{\mathrm{P}}_{btt} \mathrm{SAT}$.

1.2. Sparse sets in the boolean hierarchy and E = ?NE. Hartmanis, Immerman, and Sewelson [HIS83] linked the behavior of sparse sets in NP to the E = NE question. We show that E = NE forces sparse sets in odd levels of the boolean hierarchy down one level. This result is the best possible among results holding in all relativizations;

our "candy cane" relativization alternates stripes with and without sparse sets throughout the boolean hierarchy (Fig. 2b). Indeed, since this oracle leaves sparse sets in $\mathrm{coNP}(2k+1)-\mathrm{NP}(2k+1)$ but not in $\mathrm{NP}(2k+1)-\mathrm{coNP}(2k+1)$, we have an infinite set of structurally asymmetric complementary classes—the levels of the boolean hierarchy.

THEOREM. $\mathrm{E}=\mathrm{NE}$ *if and only if for all* k, $\mathrm{NP}(2k+1)-\mathrm{NP}(2k)$ *has no sparse sets.*

THEOREM (Candy Cane Relativization). *There exists a set* A *such that for all* $k \geq 0$, $\mathrm{NP}^A(2k+1)-\mathrm{NP}^A(2k)$ *has no sparse sets, yet* $\mathrm{NP}^A(2k+2)-\mathrm{NP}^A(2k+1)$ *has sparse sets.*

It follows easily that $\mathrm{E}=\mathrm{NE}$ forces all tally sets out of $\mathrm{BH}-\mathrm{P}$, but may leave sparse sets. Is there some natural class of sparse sets that is richer than the class of tally sets and can be forced from the boolean hierarchy? To answer this question, we define *capturability* of sets, a notion we feel to be of general interest.

DEFINITION. A set is **capturable** if it is a subset of some sparse set in NP.

THEOREM. *If* $\mathrm{E}=\mathrm{NE}$, *then all capturable sets in the boolean hierarchy are in* P.

1.3. Counting classes. Counting classes, such as the class US defined by Blass and Gurevich [BG82], seem promising sources of natural complete sets for the boolean hierarchy. Let S be a finite or cofinite set of natural numbers. We define its associated counting class, CP_S, as the set of languages accepted by a nondeterministic polynomial-time machine that has the following acceptance mechanism: an input x is accepted if and only if the number of accepting computation paths on input x is a member of S. These classes are all contained in Δ_2^{P}. In § 3 we discuss natural complete languages, containment relationships, and relativized separations among counting classes. Our relativizations, which are obtained by combinatorially exploiting the limited control of counting machines, reveal the weakness of counting classes; in appropriately relativized worlds, counting classes fail to even contain NP.

These classes have canonical complete languages of the form $\mathrm{SAT}_S=\{f \mid \text{the number of solutions to } f \text{ is a member of } S\}$. Further, many basic containments can be derived from the following theorem.

CONTAINMENT THEOREM. *For* $z \geq 1$, $c_l \geq 1$ $(1 \leq l \leq z)$, $k \geq 0$, *and* $j > i \geq \min_{1 \leq l \leq z}(k_l) \geq 1$:

$$\mathrm{CP}_{\{i,j\}} \subseteq \mathrm{CP}\left\{c_1\binom{i}{k_1}+c_2\binom{i}{k_2}+\cdots+c_z\binom{i}{k_z}+k,\ c_1\binom{j}{k_1}+c_2\binom{j}{k_2}+\cdots+c_z\binom{j}{k_z}+k\right\}.$$

We display a world where more counting is strictly more powerful. Nonetheless, the most interesting aspect of counting is that it can be made *weak*. Counting can be made too weak to capture NP or coNP. As a consequence, no amount of counting is hard for even low levels of the boolean hierarchy.

THEOREM. *There is a recursive set* A *so that* $\bigcup_{S \text{ finite}} \mathrm{CP}_S^A \not\supseteq \mathrm{NP}^A$ *and* $\bigcup_{S \text{ cofinite}} \mathrm{CP}_S^A \not\supseteq \mathrm{coNP}^A$.

COROLLARY. *There is a relativized world where no amount of counting is hard for* $\mathrm{D}^{\mathrm{P}}(A)$ *or* $\mathrm{coD}^{\mathrm{P}}(A)$ *(i.e., where no counting class contains either* $\mathrm{D}^{\mathrm{P}}(A)$ *or* $\mathrm{coD}^{\mathrm{P}}(A)$*).*

1.4. The boolean hierarchy and sparse oracles for NP. We study the relations between sparse oracles and the boolean hierarchy. We say NP has a *sparse oracle* if there is a sparse set S so that $\mathrm{NP} \subseteq \mathrm{P}^S$. Mahaney [M82] showed that if NP has a sparse many-one complete set then $\mathrm{P}=\mathrm{NP}$. Karp and Lipton [KL80] showed that if NP has a sparse Turing hard set or a sparse oracle, then $\Sigma_2^{\mathrm{P}}=\Pi_2^{\mathrm{P}}$. Does the existence of a sparse oracle for NP imply something stronger than $\Sigma_2^{\mathrm{P}}=\Pi_2^{\mathrm{P}}$?

By displaying a relativized world where NP has a sparse oracle but in which the boolean hierarchy is infinite, we show that the conclusion in [KL80] is nearly the

strongest possible among results that relativize. In terms of circuits, this result shows that (in relativized worlds) the existence of small circuits for NP does not imply that $P = NP$ or even that the boolean hierarchy is finite. Our result improves the work of Kurtz, Immerman, and Mahaney reported in [HIS83]. Similarly, by displaying a relativized world where PSPACE has a sparse oracle but where the boolean hierarchy is infinite, we show that the Karp-Lipton result on sparse oracles for PSPACE (i.e., if PSPACE has a sparse oracle then $\text{PSPACE} = \Sigma_2^P$) is also nearly as strong as possible. Indeed, we show the stronger result that $P^S = \text{PSPACE}^S$, S sparse, can be consistent with an infinite boolean hierarchy.

THEOREM. *There is a relativized world where the boolean hierarchy is infinite and* PSPACE *has sparse oracles (and, indeed, in which there is a sparse S so $P^S = \text{PSPACE}^S$).*

COROLLARY. *No proof that relativizes can show the following: if* NP *has a sparse oracle then the polynomial hierarchy equals the boolean hierarchy.*

2. Sparse sets in the boolean hierarchy and E = ?NE.

2.1. Basic results. In 1983, Hartmanis, Immerman, and Sewelson [HIS83] extending work of Book [B74], noted a connection between the density of sets in NP and the structure of exponential time classes. They proved the following theorem, in which $E = \bigcup_{c>0} \text{TIME}[2^{cn}]$ and $NE = \bigcup_{c>0} \text{NTIME}[2^{cn}]$.

THEOREM 2.1.1 [HIS83]. *The following are equivalent:*

(1) *There is a sparse set in* $NP - P$.

(2) *There is a tally set in* $NP - P$.

(3) $E \neq NE$.

COROLLARY 2.1.1 [HIS83]. *There is a recursive set A for which $E^A = NE^A$ and $\text{coNP}^A - P^A$ has sparse sets.*

This section considers the implications of the existence of sparse sets in the boolean hierarchy. We start by noting that there are tally sets in $BH - P$ if and only if $E \neq NE$. It follows that forcing tally sets from $NP - P$ banishes them from the entire boolean hierarchy; that is, $BH - P$ has tally sets if and only if $NP - P$ does.

THEOREM 2.1.2. *There are tally sets in* $BH - P$ if and only if $E \neq NE$.

Proof of Theorem 2.1.2. $\Rightarrow$: Assume $E = NE$. Let L be a tally set from the boolean hierarchy. In normal form, as the finite union of D^P sets, $L = (N_1 \cap A_1) \cup \cdots \cup (N_z \cap A_z)$, where the $N_i \in NP$ and $A_i \in \text{coNP}$. Since $L \subseteq 1^*$, $L = ((N_1 \cap 1^*) \cap (A_1 \cap 1^*)) \cup \cdots \cup ((N_z \cap 1^*) \cap (A_z \cap 1^*))$. When $E = NE$, all tally sets in $NP \cup \text{coNP}$ fall to P [HIS83], so L is the boolean combination of P sets. Thus L is in P.

$\Leftarrow$: If $E \neq NE$, there are even tally sets in $NP - P$, by an easy padding procedure. □

COROLLARY 2.1.2. *There are tally sets in* $NP - P$ *if and only if there are tally sets in* $BH - P$.

The behavior of sparse sets in the boolean hierarchy when $E = NE$ is unusual: sparse sets fall out of alternate levels. That is, $NP - P$, $NP(3) - DP$, $NP(5) - NP(4)$, $\cdots$ have no sparse sets when $E = NE$ (Fig. 2a). Also, $\text{coD}^P - \text{coNP}$, $\cdots$, $\text{coNP}(2k+2) - \text{coNP}(2k+1)$ have no sparse sets.

There is a relativized world where sparse sets do fall from alternate levels, yet they find sanctuary in the gaps, i.e., P, $D^P - NP$, $NP(4) - NP(3)$, $\cdots$. Since in this world the boolean hierarchy is striped with regions of sparseness and nonsparseness, we call this the candy cane relativization (Fig. 2(b)). Corollary 2.1.1 reflects the base case of this result.

THEOREM 2.1.3. (a) $E = NE \Leftrightarrow (\forall k)$ [$NP(2k+1) - NP(2k)$ *has no sparse sets*].[1]

[1] Recall that $NP(0) = P$.

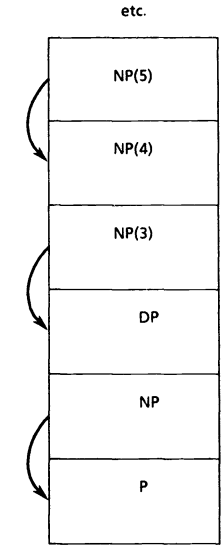


FIG. 2(a). *The arrows indicate the movement of sparse sets when* E = NE.

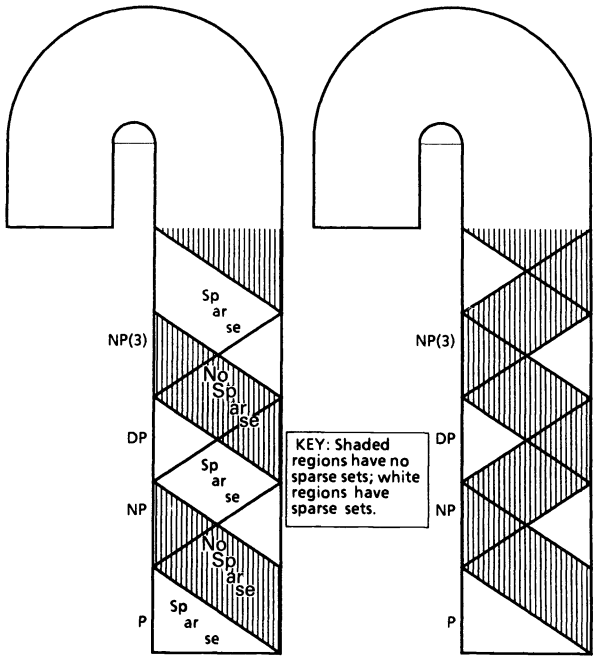


FIG. 2(b). *In our candy cane relativization, regions with and without sparse sets alternate. The left cane displays the world of Theorem* 2.1.4. *An easy modification of the oracle (noting Theorem* 2.1.3(b)) *gives the right cane.*

(b) $E = NE \Rightarrow (\forall k)\,[coNP(2k+2) - coNP(2k+1)$ *has no sparse sets*].

Proof of Theorem 2.1.3. (a) $\Leftarrow$: The proof is by Theorem 2.1.1.

(a) $\Rightarrow$: Let $E = NE$ and T be a sparse set in $NP(2k+1)$. T can, by the definition of $NP(2k+1)$, be written $N \cup V$, where N is in NP and V is in $NP(2k)$. Since T is sparse, N must be sparse. So N is a sparse set in NP and thus is in P, since $E = NE$. $NP(2k)$ is closed under intersection with P sets, so T is in $NP(2k)$.

(b) The proof of (b) is similar. □

THEOREM 2.1.4. *There is a recursive set A for which we have the following*:

(1) $E^A = NE^A$ (*thus* $NP^A - P^A$, $NP^A(3) - DP^A, \cdots, NP^A(2k+1) - NP^A(2k), \cdots$ *have no sparse sets*), *and*

(2) $(\forall k)\,[coNP^A(2k+1) - NP^A(2k+1)$ *has sparse sets*].

Proof. The proof is deferred to § 2.3.

Theorem 2.1.3(a) is a complete characterization, yet Theorem 2.1.3(b) is only an implication. Can Theorem 2.1.3(b) be made into an "if and only if" statement? In certain relativized worlds, the answer is *no*, as it is possible to construct an oracle for which $NP - P$ contains sparse sets (thus by [HIS83] $E \neq NE$) yet for which $coNP = NP$ (so $BH - coNP = \emptyset$).

2.2. Capturable sets in the boolean hierarchy. So far, we have noted that $E = NE$ forces all tally sets out of $BH - P$, but may leave sparse sets. Is there some natural class of sparse sets that is richer than the class of tally sets and can be forced from the boolean hierarchy?[2] To study this question, we define *capturability* of sets.

DEFINITION 2.2.1. A set is **capturable** if it is a subset of some sparse set in NP.

Immunity of sets has been intensely studied (see Russo [R85] and Schöning and Book [SB84]). A set is NP-immune if it contains no infinite NP subset. Our definition of uncapturability provides a complement to this notion; immunity means no NP set is inside a set and uncapturability means the set is inside no sparse NP set. A set T is uncapturable if every NP superset of T is nonsparse.

Capturable sets are a rich class. Nonetheless, capturable sets are sufficiently simple that we can answer the question that motivated this section. When $E = NE$, all capturable sets in the boolean hierarchy fall into P.

We first prove the following weaker result.

LEMMA 2.2.1. *If* $E = NE$, *then* $BH - coNP$ *has no capturable sets.*

Proof of Lemma 2.2.1. Assume $E = NE$ and let L be a capturable set from the boolean hierarchy. Write L in normal form as the finite union of D^P sets: $L = \bigcup_{i \leq k_L} (N_i \cap A_i)$, where $N_i \in NP$ and $A_i \in coNP$. Let S be the sparse NP set capturing L. Then $L = \bigcup_{i \leq k_L} ((N_i \cap S) \cap A_i)$. But since $E = NE$ each $N_i \cap S$ is in P, so L is the finite union of coNP sets and thus is itself in coNP. □

THEOREM 2.2.1. *If* $E = NE$, *then all capturable sets in the boolean hierarchy are in* P.

Proof of Theorem 2.2.1. Assume $E = NE$ and let S be a capturable set in the boolean hierarchy, captured by S': $S \subseteq S' \in NP$, S' and S are sparse. Let

$$L = \{n \# k \mid z_{n,k} \in S\}$$

where $z_{n,k}$ is the kth string (lexicographically) in $W_n = S' \cap (\Sigma + \varepsilon)^n$. Since $E = NE$, S' is P-printable by the results of Hartmanis and Yesha [HY84], and by the above lemma, $S \in coNP$. Thus $L \in coNE$, so $L \in E$.

---

[2] Certainly for the class "subsets of P-printable sets [HY84], [HH86a]" this property holds. We seek a far richer class.

Since $L \in \mathrm{E}$, the following program accepting S is in P:

{Let $n := |x|$;
P-print W_n;
If $x \notin W_n$ then {reject x}
else {suppose x is the kth string (lexicographically) in W_n;
accept x if and only if $n \# k \in L$.
}
}

□

2.3. The candy cane construction. Recall (§ 2.1) that the candy cane construction leaves sparse sets in patches throughout the boolean hierarchy (Fig. 2(b)). Though the construction is somewhat involved, the strategy is straightforward. While coding an NE complete set into E, we simultaneously perform diagonalizations in the boolean hierarchy to insure an infinite class of sets sanctuary at various levels of the hierarchy. The oracle places the "sawing" technique introduced in [CGI] within the framework developed in [HIS83, Thm. 12]. The "sawing" technique is described in detail in [CGI].

We achieve $\mathrm{E}^A = \mathrm{NE}^A$ by coding into our oracle an NE-complete language. Let N be a machine so that for all oracles A, $L(N^A)$ is complete for NE^A. Without loss of generality (i.e., by padding), N runs in time 2^{2^n}. We will code our oracle so $1 \# x \# 1^{2^{2^{|x|}}} \in A \Leftrightarrow x \in L(N^A)$.

Simultaneously, we wish to offer sanctuary to sparse sets in coNP, coNP(3), coNP(5), $\cdots$, coNP$(2k+1)$, $\cdots$. We define a family of languages $\{S_{2k+1}\}$, so that each S_{2k+1} is obviously in coNP$(2k+1)$. By careful diagonalization we show that, for each k, no NP$(2k+1)$ machine accepts S_{2k+1}. In addition, we poison the oracle to insure that each S_{2k+1} remains sparse.

To space out our diagonalizations so no two interfere with each other, we define widely spaced columns in which the diagonalizations are based. Let $C_k = \{m \mid (\exists i)[m = 2^{2^{\langle k,i \rangle}}]\}$, where $\langle \cdot, \cdot \rangle$ is a standard pairing function. Each diagonalization involving S_{2k+1} will use a diagonalizing string whose length is in C_{2k+1}.

Taking advantage of these columns as spacing mechanisms, define our sanctuary-finding sets as follows. Note that each S_{2k+1} is in coNP$(2k+1)$. The notation $(\forall w : C : S)$ means "for all w satisfying C, S holds."

$$S_1 = \{x \mid |x| \in C_1 \wedge (\forall y : |y| = |x| + 1 : 0xy \notin A)\},$$

$$S_3 = \{x \mid |x| \in C_3 \wedge [(\forall y : |y| = |x| + 1 : 0xy \notin A) \wedge [(\exists y : |y| = |x| + 2 : 0xy \in A) \vee (\forall y : |y| = |x| + 3 : 0xy \notin A)]]\}$$

$$S_{2k+1} = \{x \mid |x| \in C_{2k+1} \wedge [(\forall y : |y| = |x| + 1 : 0xy \notin A) \wedge [(\exists y : |y| = |x| + 2 : 0xy \in A) \vee [(\forall y : |y| = |x| + 3 : 0xy \notin A) \wedge [\cdots[(\exists y : |y| = |x| + 2k : 0xy \in A) \vee (\forall y : |y| = |x| + 2k + 1 : 0xy \notin A)]\cdots]]]]\}.$$

Now, for each k, we must assure that S_{2k+1} is accepted by no NP$(2k+1)$ machine. We model an NP$(2k+1)$ language H as a collection of $k+1$ NP machines N_i and k coNP machines A_i, so $H = N_1 \cup [A_2 \cap [N_2 \cup [\cdots \cup [A_{k+1} \cap N_{k+1}] \cdots]]]$. We use the machines and their languages synonymously. Without loss of generality, let the ith coNP$(2k+1)$ machine be composed of machines running each in time $n^i + i$.

Using the method of successive restrictions (the "sawing" technique) described in [CGI], we easily perform the needed diagonalizations.

CANDY CANE CONSTRUCTION.

Stage n, Part A. If $n \notin \bigcup_{l \geq 0} C_{2l+1}$ do nothing. Otherwise, suppose $n \in C_{2k+1}$. Let H be the first NP$(2k+1)$ machine that might still accept S_{2k+1}; let us say that H is the

lth NP $(2k+1)$ machine. If $l > \frac{1}{2}\log n$ then skip Part A, otherwise go on. We show that $L(H^A) \neq S^A_{2k+1}$.

Find an x_0, $|x_0| = n$, such that for all y with $n \leqq |y| \leqq n+2k+1$, $0x_0y$ is *unspecified* (i.e., a string that has been neither placed in nor barred from A).

Recall that

$$L(H) = L(N_1) \cup [L(A_2) \cap [L(N_2) \cup [\cdots \cup [L(A_{k+1}) \cap L(N_{k+1})]\cdots]]].$$

Case 1. There is some extension of A so $N_1(x_0)$ accepts. Thus $x_0 \in L(H^A)$ on this extension. Fix the oracle strings queried along some accepting path of $N_1^A(x_0)$ on this extension. To assure that $x_0 \notin S_{2k+1}$, find a y, $|y| = n+1$, so $0x_0y$ is unspecified and add it to the oracle.

Case 2. For every extension of A, $N_1^A(x_0)$ rejects and for some extension of A so that $(\forall y\colon |y| = n+1\colon 0x_0y \notin A)$, $A_2^A(x_0)$ rejects. Thus $x_0 \notin L(H^A)$ on this extension. Fix in A the elements along some rejecting path. Now assure $x_0 \in S_{2k+1}$ by adding an unspecified string $0x_0y$, $|y| = n+2$, to A.

Case $2j$ $(1 \leqq j \leqq k)$. Cases $1, 2, \cdots, 2j-1$ fail to hold and for some extension of A so that $(\forall y\colon n+1 \leqq |y| \leqq n+2j-1\colon 0x_0y \notin A)$, $A_{j+1}^A(x_0)$ rejects. Fixing a rejecting path assures $x_0 \notin L(H^A)$. Insure that $x_0 \in S_{2k+1}$ by choosing a $|y| = n+2j$ so $0x_0y$ is unspecified and add it to our oracle.

Case $2j+1$ $(0 \leqq j \leqq k)$. Cases $1, 2, \cdots, 2j$ fail to hold and for some extension of A so that $(\forall y\colon n+1 \leqq |y| \leqq n+2j\colon 0x_0y \notin A)$, $N_{j+1}^A(x_0)$ accepts. Fixing an accepting path puts x_0 in $L(H^A)$. Assure $x_0 \notin S_{2k+1}$ by choosing a y, $|y| = n+2j+1$, so $0x_0y$ is unspecified and add $0x_0y$ to our oracle.

Case $2k+2$. Cases $1, 2, \cdots, 2k+1$ fail to hold. So H rejects x_0 on any extension where $(\forall y\colon n+1 \leqq |y| \leqq n+2k+1\colon 0x_0y \notin A)$. Freeze all these strings out of A. Now $x_0 \in S_{2k+1}$, but $x_0 \notin L(H^A)$.

All cases. If any of the cases $1, 2, \cdots, 2k+2$ has occurred, we must act to maintain the sparseness of S_{2k+1}. For each $x \neq x_0$, find a y_x, $|y_x| = |x|+1$ so $0xy_x$ has not yet been specified and add that string to A. Thus S_{2k+1} has at most one string of length n.

Stage n, Part B. Recall that N is a machine so $L(N^A)$ is NE^A complete for all A. We will code so that $1\#x\#1^{2^{2|x|}} \in A \Leftrightarrow x \in L(N^A)$.

For each string x so that $\log^2 n \leqq |x| \leqq \log^2 (n+1)$, in order from the smallest such x to the largest, do the following. If there is an extension of A so that $N^A(x)$ accepts, fix the elements along that path and add $1\#x\#^{2^{2|x|}}$ to A. Otherwise, fix $1\#x\#1^{2^{2|x|}}$ as out of A.

End Construction

Notes on the construction.

(1) We can find an x_0 in Part A, as there are 2^n candidates and at most $2^{1+\log^2 n} 2^{2\log^2 n} = 2n^{3\log n}$ strings of length at least n have been fixed. All these have been fixed in Part B. Our Part A actions are spaced double exponentially far apart in length. Each Part A action cannot get near the strings involved in the next Part A diagonalization.

(2) Similarly, we can find the strings y_x with which we poison strings to maintain sparseness in Part A.

(3) In Part B, each coding string $1\#x\#1^{2^{2|x|}}$ will certainly be unspecified when we assign it. No Part B computation will have touched this string. Any Part A machine run at stage n cannot even reach things of length $n^{\log n}$, by our assumption that the ith NP $(2k+1)$ machine is composed of machines running in time at most $n^i + i$ and our choice of H. So no Part A machine has touched our coding strings.

3. Counting classes. Counting classes at first seems tempting as natural complete sets for the boolean hierarchy. In this section, we define the notion of a counting class, and display relativized worlds where counting classes are *not* complete for the levels of the boolean hierarchy.

NP is the class of sets accepted by nondeterministic polynomial-time Turing machines (NPTMs) that (by definition) accept when *at least one* computation path accepts. US is the class of sets accepted by NPTMs that accept when *exactly one* computation path accepts. Blass and Gurevich [BG82] showed that there are relativized worlds where UNIQUE-SAT, a natural complete set for US, is and is not complete for D^P.

We are interested in more generalized notions of machines that count accepting paths (see also [GW87]). Our machines accept an input if the number of accepting paths generated by that input is in a prescribed set. These counting classes have natural complete sets, i.e., formulas with specified numbers of satisfying assignments. By way of contrast, counting classes in which certain numbers of accepting paths are forbidden have no complete sets in appropriately relativized worlds (Hartmanis and Hemachandra [HH86b]).

Please note that throughout § 3, even when not explicitly mentioned, in every use of CP_S and SAT_S we tacitly assume that S is a subset of $\{1, 2, 3, \cdots\}$ and is either finite or cofinite.

DEFINITION 3.1. (a) SAT_S is the set of all formulas whose number of satisfying assignments is in S. For example, SAT is $SAT_{\{1,2,3,\cdots\}}$ and UNIQUE-SAT is $SAT_{\{1\}}$.

(b) CP_S is the class of sets accepted by NPTMs that accept, by definition, when they have a number of accepting computations in S. Mnemonically, CP stands for "counting polynomial time." Two examples are: NP is $CP_{\{1,2,3,\cdots\}}$ and US is $CP_{\{1\}}$.

LEMMA 3.1. (a) SAT_S *is many-one-complete for* CP_S.

(*b*) *For any S and any complexity class C closed downward under many-one polynomial-time reductions,*

$$SAT_S \textit{ is } A\textit{-hard}^3 \Leftrightarrow CP_S \supseteq A.$$

For example, SAT_S *is* NP-*hard* $\Leftrightarrow CP_S \supseteq NP$.

(c) *For S finite,* $SAT_S \supseteq coNP$. *For S cofinite,* $SAT_S \supseteq NP$.

Proof of Lemma 3.1. (a) Cook's reduction [C71], [GJ79] is parsimonious.

(b) The proof is immediate.

(c) Let S be finite. Let a_z be the largest element of S. Given a formula f, we can quickly either find a solution (so f is not in UNSAT) or find a_z nonsatisfying assignments. Thus f with these assignments ORed on has a_z more solutions than f, and is thus in SAT_S if and only if f is in UNSAT; thus SAT_S is coNP-hard. Thus we are done by part (b).

Similarly, for S cofinite, SAT_S is NP-hard because we can add j true assignments to a formula, $j = \max\{i \mid i \geq 0 \wedge i \notin S\}$. □

Clearly, counting of this sort can be done in hardware. For example, a set in CP_S, $|S| = z$, is the union of z D^P sets, and so is in $NP(2z)$. Similarly, for cofinite S, $CP_S \subseteq coNP(2y)$, when $|\bar{S}| = y$. The containment can easily be made an equality in relativized worlds. Is the containment always an equality?

We show that in appropriately relativized worlds counting is weak. For example, we display a world where for *no* choice of S is SAT_S complete for D^P or coD^P. Furthermore, in this world for no choice of finite S can SAT_S even be NP-hard and

[3] We always use "hard" in the many-one sense.

for no choice of cofinite S can SAT_S be coNP-hard (where we relativize satisfiability in the standard way—adding membership testing gates). Indeed, each counting class with S, S finite (cofinite), fails to contain NP (coNP) even when its run time is boosted from polynomial to certain *exponential* times.

Similarly, we separate many counting classes so strongly that large boosts in run time fail to give containment. All these relativized containments are obtained by combinatorially exploiting the limited control of counting machines. On the other hand, we note that a clear containment structure exists among counting classes.

3.1. A counting hierarchy. Suppose you have a party at which each guest may know the names of some of the other guests. The following combinatorial lemma states conditions under which your party will contain a large set of "strangers," i.e., a large set A such that each person in A knows the name of no person in A (except perhaps his own name). This lemma will be useful in proving our separations. S_i contains the names of party-goers known by party-goer i. In the proofs of the following relativizations (e.g., Theorem 3.1.1(b)), the S_w correspond to the strings queried by a certain counting machine when string w is added to the oracle. The combinatorial lemma lets us find strings w' and w'' so that adding both simultaneously causes the number of accepting paths to add.

COMBINATORIAL LEMMA.

$$(\forall S_1, \cdots, S_l \subseteq N) \quad \left[\sum |S_i| < \left\lceil \frac{l(l-1)}{k(k-1)} \right\rceil \Rightarrow (\exists T \subseteq \{1, 2, \cdots, l\}, |T| \geqq k)(\forall e \in T) \right.$$
$$\left. [S_e \cap (T - e) = \varnothing] \right].$$

Proof of Combinatorial Lemma. Indeed, we show there is a T of size exactly k. There are $\binom{l}{k}$ candidate k-tuples for T. Each element added to some set destroys the candidacy of at most $\binom{l-2}{k-2}$ new tuples. For example, adding $x \leqq l$ to set S_y eliminates tuples containing both x and y. Thus we need at least

$$\left\lceil \binom{l}{k} \Big/ \binom{l-2}{k-2} \right\rceil = \left\lceil \frac{l(l-1)}{k(k-1)} \right\rceil$$

elements to eliminate all tuples. □

Our first theorem shows that "the number of values accepted on" yields a hierarchy: *more counting is more powerful,* and strictly so in some worlds.

THEOREM 3.1.1.

(a) $\mathrm{CP}_{\{1\}} = \bigcup_{|S|=1} \mathrm{CP}_S \subseteq \bigcup_{|S|=2} \mathrm{CP}_S \subseteq \bigcup_{|S|=3} \mathrm{CP}_S \subseteq \cdots \subseteq \mathrm{BH} \subseteq \Delta_2^{\mathrm{P}}$.

(b) *There is a recursive oracle* A *so* $\bigcup_{|S|=1} \mathrm{CP}_S^A \subset \bigcup_{|S|=2} \mathrm{CP}_S^A \subset \bigcup_{|S|=3} \mathrm{CP}_S^A \subset \cdots$.

Proof of Theorem 3.1.1(a). Clearly counting is in BH. Indeed, CP_S sets can be represented as the union of $|S|$ D^{P} sets, and thus by the lemma cited in § 1.1 are in the boolean hierarchy. If $L \in \mathrm{CP}_{\{a_1, \cdots, a_z\}}$, there is a $\mathrm{CP}_{\{1, 2+a_1, 2+a_2, \cdots, 2+a_z\}}$ machine that simulates the $\mathrm{CP}_{\{a_1, \cdots, a_z\}}$ machine for L, and mindlessly adds two accepting paths. This new machine accepts L. Since this holds in general, $\bigcup_{|S|=1} \mathrm{CP}_S \subseteq \bigcup_{|S|=2} \mathrm{CP}_S \subseteq \bigcup_{|S|=3} \mathrm{CP}_S \subseteq \cdots$. Finally, $\mathrm{CP}_{\{j\}} = \mathrm{CP}_{\{1\}}$ from Theorem 3.2.1.1. □

Proof of Theorem 3.1.1(b). Because of space limitations, we just show a world in which $\bigcup_{|S|=1} \mathrm{CP}_S^A \subset \bigcup_{|S|=2} \mathrm{CP}_S^A$. By Theorem 3.1.1(a), it suffices to show a language L in $\mathrm{CP}_{\{1,2\}}^A$ that is not in $\mathrm{CP}_{\{1\}}^A$. We diagonalize over all $\mathrm{CP}_{\{1\}}$ machines to show that $L = \{0^i \mid 1 \leqq |\Sigma^i \cap A| \leqq 2\}$ is not in $\mathrm{CP}_{\{1\}}$. Since L is clearly in $\mathrm{CP}_{\{1,2\}}$ this suffices. Initially, set $A := \varnothing$.

Stage s. We will establish that the sth $\mathrm{CP}_{\{1\}}$ machine, call it C_i, does not accept L. Let $p(\cdot)$ be the polynomial bound on the length of each of C_i's computation paths

(and thus on the length of the strings it queries). Choose j so large that no string of length j or larger has yet been touched, and so large that $p(j) < (2^j - 1)/2$.

Case 1. $C_i^A(0^j)$ has accepting paths.

If it has one accepting path, freeze A for all strings of length up to $p(j)$; now $0^j \notin L$ but $0^j \in L(C_i^A)$.

If it has more than one accepting path, fix the elements along two paths, put some unfrozen length j string into A, and then freeze A for all strings of length up to $p(j)$; now $0^j \in L$ but $0^j \notin L(C_i^A)$.

Case 2. For some length j string y, $C_i^{A \cup y}(0^j)$ rejects.

Set $A := A \cup \{y\}$ and freeze A for all strings of length up to $p(j)$. Now $0^j \in L$, $0^j \notin L(C_i^A)$.

Case 3. For all length j strings, y: $C_i^{A \cup y}(0^j)$ accepts (i.e., it has a unique accepting path).

For each length j string w define a set

$$S_w = \{z \mid |z| = j \text{ and } C_i^{A \cup w}(0^j) \text{ queries } z \text{ along its accepting path}\}.$$

We use the Combinatorial Lemma from the start of this section with $k = 2$, $l = 2^j$, and using the 2^j sets S_w for $|w| = j$. The precondition of the lemma is met since by our conditions on the choice of j we have $2^j p(j) < 2^j((2^j - 1)/2)$. Thus there exist strings y', y'' so y' is not in $S_{y''}$ and y'' is not in $S_{y'}$. Thus if we add y' and y'' simultaneously to A we will get at least two accepting paths: the ones corresponding to $S_{y'}$ and $S_{y''}$. (Why are these two paths distinct? If $y'' \notin S_{y''}$ then we have an accepting path in $C_i^A(0^j)$, and this was treated in Case 1. Hence $S_{y''}$ and $S_{y'}$ differ on y''.) So $C_i^{A \cup y' \cup y''}(0^j)$ has at least two accepting paths and rejects. Setting $A := A \cup \{y', y''\}$ and fixing A up to length $p(j)$ we have insured that $0^j \in L$ but $0^j \notin L(C_i^A)$. □

More counting gives more power, as these theorems show. Indeed, we have an infinite hierarchy of counting power. Attempts to make the polynomial hierarchy infinite in some relativization took many years to succeed [Y85], [BS79].

Counting can also be made *weak*. Below, we show that counting may be too weak to even capture NP or coNP. As a consequence, we show that in certain relativized worlds no amount of counting is hard for even low levels of the boolean hierarchy.

THEOREM 3.1.2. *There is a recursive set A so that* $\bigcup_{S \text{ finite}} \mathrm{CP}_S^A \not\supseteq \mathrm{NP}^A$ *and* $\bigcup_{S \text{ cofinite}} \mathrm{CP}_S^A \not\supseteq \mathrm{coNP}^A$.

COROLLARY 3.1.1. *There is a relativized world where no amount of counting is hard for* $\mathrm{NP}^A(k)$ *or* $\mathrm{coNP}^A(k)$, $k > 1$ *(i.e., where no counting class contains either* $\mathrm{NP}^A(k)$ *or* $\mathrm{coNP}^A(k)$, $k > 1$*). In particular, no counting class contains* $\mathrm{D}^{\mathrm{P}}(A)$ *or* $\mathrm{coD}^{\mathrm{P}}(A)$.

Proof of Corollary 3.1.1. $\mathrm{NP}^A(k)$ and $\mathrm{coNP}^A(k)$ are each NP^A-hard and coNP^A-hard for every $k > 1$, yet the theorem above shows that the artificial and natural complete languages for counting classes fail to be hard for at least one of these. □

Proof of Theorem 3.1.2. Let $L_A = \{0^n \mid (\exists y)[|y| = n \wedge y \in A]\}$. Let $LL_A = \{0^n \mid (\forall y)[|y| = n \Rightarrow y \notin A]\}$. Clearly $L_A \in \mathrm{NP}^A$ and $LL_A \in \mathrm{coNP}^A$.

Stage $\langle\langle i, z, a_1, \cdots, a_z\rangle, 0\rangle$, $0 < a_1 < \cdots < a_z$. We show that the ith NP machine, viewed as a $\mathrm{CP}_{\{a_1, \cdots, a_z\}}^A$ machine, does not accept L_A. Without loss of generality NP_i runs in $\mathrm{NTIME}[n^i + i]$.

Choose m so large that nothing of length greater than m has yet been fixed, and so large that $2^m(m^i + i) < \lceil 2^m(2^m - 1)/a_z(a_z + 1) \rceil$. Let B be the current version of A. Run $N_i^B(0^m)$.

Case 1. $N_i^B(0^m)$ accepts. $L(N_i^B) \neq L_A$ and we are done.

Case 2a. $N_i^B(0^m)$ rejects and for one of the 2^m ways we can add a length m string, y, to B, $N_i^{B \cup y}(0^m)$ rejects. Set our current A to $B \cup y$ and again $L(N_i^A) \neq L_A$.

Case 2b. $N_i^B(0^m)$ rejects and for all the 2^m ways we can add a length m string, y, to B, $N_i^{B\cup y}(0^m)$ accepts. We apply the combinatorial lemma (from the start of this section) to get a contradiction.

That is, for each of the 2^m extensions, $B \cup y$, mark one accepting path of $N_i^{B\cup y}(0^m)$ and say the path is *named* by y. Let S_y denote the strings queried on the path named y. By our assumption that N_i runs in NTIME $[n^i+i]$, $\sum |S_y| \leqq 2^m(m^i+i)$. By our choice of m, the combinatorial lemma applies (with $l=2^m$, $k=a_z+1$) and chooses for us a set T of a_z+1 special marked paths, none of which queries the name of any of its a_z brethren. Furthermore, the a_z+1 paths are all distinct, since each one contains its own name and none of its brethren does. Each special path does indeed contain its name, otherwise it would have caused $N_i^B(0^m)$ to accept, but we supposed it did not. Thus $N_i^{B\cup T}(0^m)$ has at least a_z+1 accepting paths and, viewed as (i.e., with the acceptance mechanism of) a $\mathrm{CP}^A_{\{a_1,\cdots,a_z\}}$ machine, rejects. Yet $0^m \in L_A$. So setting our current A to $B \cup T$, $L(H_i^A) \neq L_A$ and we have diagonalized away another machine.

Stage $\langle\langle i, z, a_1, \cdots, a_z\rangle, b\rangle$, $0<a_1<\cdots<a_z<b$. We show that the ith NP machine, viewed as a $\mathrm{CP}^A_{\{a_1,\cdots,a_z,b,b+1,b+2,\cdots\}}$ machine, does not accept LL_A.

Choose m large. Denote by B the current version of A. Run $N_i^B(0^m)$. If it rejects, we are done. If it accepts, for each of the 2^m ways we can add a length m string, y, to B, run $N_i^{B\cup y}(0^m)$. If any of these 2^m ways accepts, we are done. Now if extension $B \cup y$ has no accepting paths or has some accepting paths but queries y on none of them, y must appear on some accepting path of $N_i^B(0^m)$. Thus all but at most $(b-1)m^i+i$ of our 2^m extensions cause "nice" rejections: rejections (with oracle $B \cup y$) where there is some accepting path that queries y.

From here, we are done by the argument of the previous part. In brief, for each "nice" extension mark a path that queries the extension's name; appeal to the combinatorial lemma to get a set T of b paths that do not contain each others' names; thus, $N_i^{B\cup T}(0^m)$ has b accepting paths and accepts, but $0^m \notin LL_A$, thus $L(N_i^A) \neq LL_A$. □

Indeed, counting is even weaker than these results suggest. By making full use of our combinatorial control arguments, we can show that even giving counting machines *exponential* run times does not allow them to subsume NP.

THEOREM 3.1.3. *For each k, there is a recursive oracle A for which* $\mathrm{NP}^A \not\subseteq \bigcup_{S\,\text{finite}} \mathrm{CTIME}_S^A[2^{kn}]$, *where* $\mathrm{CTIME}_S^A[f(n)]$ *is just like* CP_S^A, *except our time bound has been boosted from* P *to* $f(n)$.

Proof of Theorem 3.1.3. This is an improvement on our proof of Theorem 3.1.2, whose notation we adopt. We must make new versions of L_A and LL_A that depend on many strings. We let $L_A=\{0^n \mid (\exists y)[|y|=(k+1)n \wedge y \in A]\}$ and $LL_A=\{0^n \mid (\forall y)[|y|=(k+1)n \Rightarrow y \notin A]\}$. There are $2^{(k+1)^n}$ ways (the l for the Combinatorial Lemma) of adding strings. We choose m so large that

$$2^{(k+1)m}\left(c+\frac{2^{km}}{(a_z+1)^2}\right)<\frac{2^{(k+1)m}(2^{(k+1)m}-1)}{a_z(a_z+1)}. \qquad \square$$

Indeed, the same tricks apply within the counting hierarchy. For example, there is a world where

$$\bigcup_{c,|S|=z} \mathrm{CTIME}_S^A[2^{cn/\log n}] \not\supseteq \bigcup_{|S|=z+1} \mathrm{CP}_S^A.$$

3.2. Relations and natural containments among the CP classes.

3.2.1. Relations based on values. The previous section shows that "the number of values accepted on" gives a hierarchy. In this section, we fix the *number* of values, k,

our classes accept on and see what effects the actual *values* have. For $k = 1$, the number of paths makes no difference. We can multiply paths (to show $\mathrm{CP}_{\{1\}} \supseteq \mathrm{CP}_{\{k\}}$) and simulate k-tuples of paths (to show $\mathrm{CP}_{\{k\}} \supseteq \mathrm{CP}_{\{1\}}$) [see, for example, Theorem 3.2.2.1]. Hence we have the following theorem.

THEOREM 3.2.1.1. *For every j, k and A,* $\mathrm{CP}^A_{\{j\}} = \mathrm{CP}^A_{\{k\}}$.

The situation differs markedly when $k = 2$. Our classes here are $\mathrm{CP}_{\{a,b\}}$, $a < b$, and we will call a the *minor* subscript and b the *major* subscript. Here (with proper relativizations) a class never contains any class with a larger major subscript (Theorem 3.2.1.2). When the major subscripts are the same, we can usually relativize so the class with the larger minor subscript is not contained by the other. On the other hand, there is more than lexicographical ordering at work here. We can also relativize away from lexicographical ordering. Indeed, we can combine the relativizations to obtain a world A where, for example, $\mathrm{CP}^A_{\{1,73\}} \not\subseteq \mathrm{CP}^A_{\{40,73\}}$ and $\mathrm{CP}^A_{\{1,73\}} \not\supseteq \mathrm{CP}^A_{\{40,73\}}$.

THEOREM 3.2.1.2. (a) *There is a recursive oracle A so that for* $i < j < j'$, $\mathrm{CP}^A_{\{i,j\}} \subset \mathrm{CP}^A_{\{i,j'\}}$.

(b) *There is a recursive oracle A so that for all* $1 < j < k$, $\mathrm{CP}^A_{\{1,k\}} \not\subseteq \mathrm{CP}^A_{\{j,k\}}$.

(c) *There is a recursive set A so that for all i, j, k, l with* $i < k$, $j < k$, $jl \leqq k$, *and* $i + l > k$, $\mathrm{CP}^A_{\{i,k\}} \not\supseteq \mathrm{CP}^A_{\{j,k\}}$.

COROLLARY 3.2.1.1. *There is a world where no* $\mathrm{CP}^A_{\{i',j'\}}$ *contains* $\bigcup_{i,j} \mathrm{CP}^A_{\{i,j\}}$.

Our results extend naturally to $k = 3, 4, \cdots$, and by interlacing the relativizations, we can simultaneously obtain these results for all k.

3.2.2. Containments. As we have just seen, combinatorial manipulations allow us to construct many relativized separations. Nonetheless, there is a clear and ordered set of containments among counting classes. Figure 3 shows the containments implied by Corollary 3.2.2.1. Figure 4 shows further containments from Theorem 3.2.2.1.

THEOREM 3.2.2.1 (Containment Theorem). *For* $z \geqq 1$, $c_l \geqq 1$ $(1 \leqq l \leqq z)$, $k \geqq 0$, *and* $j > i \geqq \min_{1 \leqq l \leqq z}(k_l) \geqq 1$,

$$\mathrm{CP}_{\{i,j\}} \subseteq \mathrm{CP}\left\{ c_1\binom{i}{k_1} + c_2\binom{i}{k_2} + \cdots + c_z\binom{i}{k_z} + k,\ c_1\binom{j}{k_1} + c_2\binom{j}{k_2} + \cdots + c_z\binom{j}{k_z} + k \right\}.$$

COROLLARY 3.2.2.1. (a) $\mathrm{CP}_{\{i,j\}} \subseteq \mathrm{CP}_{\{i,j'\}}$ *when* $i < j < j'$.

(b) $\mathrm{CP}_{\{i,j\}} \subseteq \mathrm{CP}_{\{ki+l,kj+l\}}$ *for* $k \geqq 1$, $l \geqq 0$.

Proof of Theorem 3.2.2.1. Simulate the $\mathrm{CP}_{\{i,j\}}$ machine by simulating with c_l-fold replication each k_l-tuple of its paths (i.e., accepting and replicating the tuple if all k_l paths accept). Also, add k dummy accepting paths. If the $\mathrm{CP}_{\{i,j\}}$ machine has $i(j)$ accepting paths, we have $f(i)$ $(f(j))$ accepting paths, where $f(m) = \sum c_l\binom{m}{k_l} + k$. If the number of paths is neither i nor j, we will *not* get either $f(i)$ or $f(j)$ paths, as f is strictly increasing in the range $[\min(k_l) - 1, \infty)$. □

4. The boolean hierarchy and sparse oracles for NP. This section studies the relations between sparse oracles and the boolean hierarchy. We say NP has a *sparse oracle* if there is a sparse set S so that $\mathrm{NP} \subseteq \mathrm{P}^S$. Mahaney [M82] showed that if NP has a sparse many-one-complete set then P = NP. Karp and Lipton [KL80], [H82] showed that if NP has a sparse Turing-hard set or a sparse oracle, then $\Sigma^{\mathrm{P}}_2 = \Pi^{\mathrm{P}}_2$. Does the existence of a sparse oracle for NP imply something stronger than $\Sigma^{\mathrm{P}}_2 = \Pi^{\mathrm{P}}_2$?

By displaying a relativized world where NP has a sparse oracle but in which the boolean hierarchy is infinite, we show that the conclusion in [KL80] is near the strongest possible among proofs that relativize.[4] In terms of circuits, this result shows that (in

[4] Kadin has optimally strengthened the Karp–Lipton result for the special case of sparse Turing-complete sets for NP [K87a], [M82].

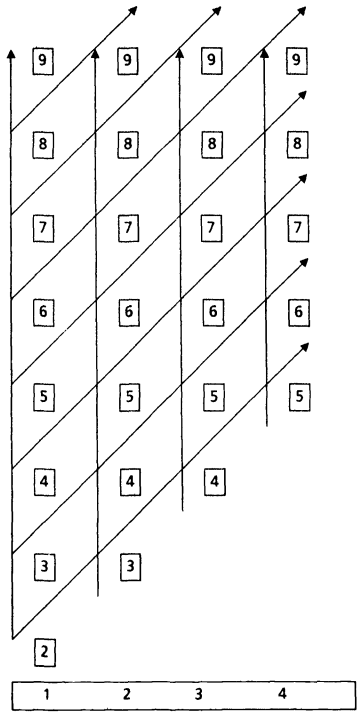


FIG 3. *Simple containments.*

relativized worlds) the existence of small circuits for NP does not imply that P = NP or even that the boolean hierarchy is finite, and improves the work of Kurtz, Immerman, and Mahaney reported in [HIS83]. Similarly, by displaying a relativized world where PSPACE has a sparse oracle but where the boolean hierarchy is infinite, we show that the Karp-Lipton result on sparse oracles for PSPACE (i.e., if PSPACE has a sparse oracle then PSPACE = Σ_2^P) is also nearly as strong as possible. Indeed, we show the stronger result that $P^S = PSPACE^S$, S sparse, can be consistent with an infinite boolean hierarchy.

Our proofs combine the methods of diagonalization and encoding of computations. By diagonalization, we win a block of space to use. We store the location of this space in the sparse oracle. We devote the space to encoding the computations of machines. Thus $NP^A \subseteq P^{A,S}$, because the sparse oracle S lets us find NP^A computations encoded in A. We carefully diagonalize to separate the boolean hierarchy. For simplicity, we prove the case of $D^P(A) \neq coD^P(A)$.

LEMMA 4.1. *There is a recursive set A and a sparse set S so* $NP^A \subseteq P^{A,S}$ *and* $D^P(A) \neq coD^P(A)$.

THEOREM 4.1. *There is a relativized world where the boolean hierarchy is infinite and* PSPACE *has sparse oracles (and, indeed, in which there is a sparse S so* $P^S = PSPACE^S$).

COROLLARY 4.1. *No proof that relativizes can show the following: if* NP *has a sparse oracle then the polynomial hierarchy equals the boolean hierarchy.*

Proof of Lemma 4.1. Let $\{N_i\}$ and $\{A_i\}$ be enumerations of standard clocked NP and coNP machines, respectively. Without loss of generality, N_i and A_i run at most $n^{\log^* i} + \log^* i$ steps on input of size n for $i > 20$, and one step for $i \leqq 20$. Define an

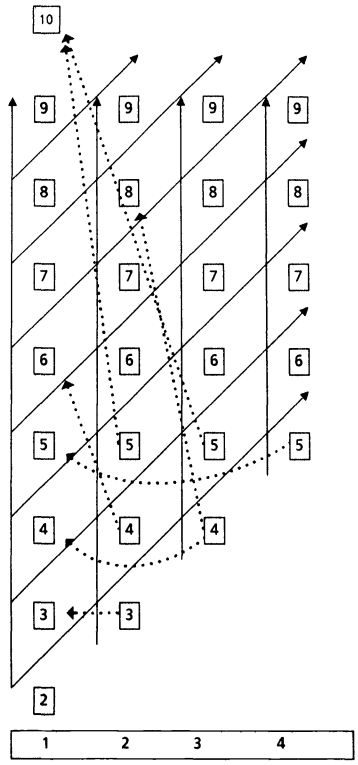


FIG. 4. *Additional containments.*

enumeration $\{H_i\}$ of standard clocked $\mathrm{coD^P}$ machines so $L(H_i) = L(N_j) \cup L(A_k)$, $i = \langle j, k \rangle$.

Let $U_A = \{N_i \# x \# 1^l \mid N_i^A(x)$ accepts in at most l steps$\}$. U_A encodes the computations of NP^A machines. We will code U_A into A and S so that $\mathrm{NP}^A \subseteq \mathrm{P}^{A,S}$.

Let $L_A = \{0^n \mid (\exists y)[|y| = 3n+1 \wedge y \in A]\} \cap \{0^n \mid (\forall y)[|y| = 3n+2 \Rightarrow y \notin A]\}$. Surely $L_A \in \mathrm{D^P}(A)$. We diagonalize over the H_i's to show that $L_A \notin \mathrm{coD^P}(A)$.

Stage q, encoding section. Choose an *address*, a_q, $|a_q| = q$, so no length $3q$ string starting with a_q has yet been fixed. The H_i's fix few enough strings that such an a_q can always be found. Now use the block of length $3q$ strings starting with a_q as a "treasure box," to encode the information of all length q strings of U_A: $[a_q y 0^q \in A \Leftrightarrow y \in U_A]$, for all $|y| = q$. Crucially, the computations being coded query strings of length at most q, and our length q strings have long since been settled.

Now we must code the precious address string, a_q, into S. For $j \leqq q$ put $1^{\langle q,j \rangle}$ in S if and only if bit j of a_q is 1. Now S codes a_q; with q calls to S we can recover a_q. Note that S will be sparse.

Stage q, diagonalizing section. Suppose H_i is the first $\mathrm{coD^P}$ machine that still might accept L_A. We will extend A so that $L(H_i^A) \neq L_A$.

Recall $L(H_i^A) = L(N_j^A) \cup L(A_k^A)$, $i = \langle j, k \rangle$. Run $N_j(0^q)$ with all possible extensions of A as its oracle.

Case 1. It accepts for some extension. Freeze all queried elements on some accepting path. Thus $L(H_i^A)$ accepts 0^q. Now, put an unfixed length $3q+2$ string into A, so $0^q \notin L_A$. H_i^A does not accept L_A.

Case 2. It rejects on all extensions. So $L(H_i^A) = L(A_k^A)$, and a coNP machine is left with the task of accepting a D^P set. Consider all extensions of A in which $(\forall z)\ [|z| = 3q+2 \Rightarrow z \notin A]$.

Case 2a. For some such extension, B, $A_i^B(0^q)$ rejects. Freeze in A the elements on a rejecting path and add to A an unfixed length $3q+1$ string. Thus $H_i^A(0^q)$ rejects but $0^q \in L_A$.

Case 2b. For all such extensions, A_i accepts. Then it accepts with A untouched, but $0^q \notin L_A$ so it should reject. □

Sketch of the proof of Theorem 4.1. The same method used above in the proof of the lemma can be used with the "sawing" diagonalization of [CGI], which allows an infinite separation of the levels. This combination creates a relativized world in which NP (or, as we note below, PSPACE) has a sparse oracle and the boolean hierarchy is infinite.

Note that once we win ourselves some coding space we can use the space to code the computations of *any* machine class that queries strings of length at most polynomial in its input's size. We just code the universal set of that class into our addressed treasure boxes. Thus the same proof works for PSPACE (part b), with the standard conventions [FSS81] about relativized PSPACE. Indeed, with a relativization we could even make the boolean hierarchy infinite while finding a sparse oracle for the badly nonrecursive class RE_{poly}, the class of sets accepted by machines that query strings of length at most polynomial in their input size. □

Acknowledgments. We thank Professors R. Book, P. Odifreddi, U. Schöning, and S. Zachos for enjoyable discussions.

REFERENCES

[B74] R. BOOK, *Tally languages and complexity classes*, Inform. and Control, 26 (1974), pp. 186–193.

[BG82] A. BLASS AND Y. GUREVICH, *On the unique satisfiability problem*, Inform. and Control, 55 (1982), pp. 80–88.

[BS79] T. BAKER AND A. SELMAN, *A second step towards the polynomial hierarchy*, Theoret. Comput. Sci., 8 (1979), pp. 177–187.

[C71] S. COOK, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual Symposium on the Theory of Computation, 1971, pp. 151–158.

[CGI] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The boolean hierarchy* I: *Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[CH85] J. CAI AND L. HEMACHANDRA, *The boolean hierarchy: Hardware over* NP, Tech. Report TR85-724, Computer Science Department, Cornell University, Ithaca, NY, 1985.

[CH86] ———, *The boolean hierarchy: Hardware over* NP, Structure in Complexity Theory, A. Selman, ed., Lecture Notes in Computer Science, 233, Springer-Verlag, Berlin, New York, 1986, pp. 105–124.

[CM85] J. CAI AND G. MEYER, *Graph minimal uncolorability is* D^P*-complete*, Tech. Report TR85-688, Computer Science Department, Cornell University, Ithaca, NY, June 1985; SIAM J. Comput. 16, (1987), pp. 259–277.

[FSS81] M. FURST, J. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial-time hierarchy*, in Proc. 22nd Annual Symposium on Foundations of Computer Science, 1981, pp. 260–270.

[GJ79] M. GAREY AND D. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP*-Completeness*, W. H. Freeman, San Francisco, 1979.

[GW87] T. GUNDERMANN AND G. WECHSUNG, *Counting classes with finite acceptance types*, Computers and Artificial Intelligence, Bratislava, Czechoslovakia, to appear.

[H82] J. HARTMANIS, *On the structure of feasible computations*, Tech. Report TR82-484, Department of Computer Science, Cornell University, Ithaca, NY, March 1982.

[HH86a] J. HARTMANIS AND L. HEMACHANDRA, *On sparse oracles separating feasible complexity classes*, in Proc. 3rd Annual Symposium on Theoretical Aspects of Computer Science (STACS '86), Lecture Notes in Computer Science, 210, Springer-Verlag, Berlin, New York, 1986, pp. 321–333; Inform. Process. Lett., to appear.

[HH86b] J. HARTMANIS AND L. HEMACHANDRA, *Complexity classes without machines: on complete sets for* UP, Proc. 13th Colloquium on Automata, Languages and Programming (ICALP '86), Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, 1986, pp. 123-135; Theoret. Comput. Sci., 58 (1988), pp. 129-142.

[HIS83] J. HARTMANIS, N. IMMERMAN, AND V. SEWELSON, *Sparse sets in* NP − P: EXPTIME *versus* NEXPTIME, Proc. 15th Annual Symposium on the Theory of Computation, 1983, pp. 382-391.

[HY84] J. HARTMANIS AND Y. YESHA, *Computation times of* NP *sets of different densities*, Theoret. Comput. Sci., 34 (1984), pp. 17-32.

[K87a] J. KADIN, *The polynomial hierarchy collapses if the boolean hierarchy collapses*, Tech. Report TR87-843, Computer Science Department, Cornell University, Ithaca, NY, 1987; SIAM J. Comput., 17 (1988), pp. 1263-1282.

[KL80] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, Proc. 12th Annual Symposium on the Theory of Computation, 1980, pp. 302-309.

[M82] S. MAHANEY, *Sparse complete sets for* NP: *solution to a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130-143.

[PW85] C. PAPADIMITRIOU AND D. WOLFE, *The complexity of facets resolved*, in Proc. 26th Annual Symposium on Foundations of Computer Science, 1985.

[PY82] C. PAPADIMITRIOU AND M. YANNAKAKIS, *The complexity of facets (and some facets of complexity)*, in Proc. 14th Annual Symposium on the Theory of Computation, 1982, pp. 255-260.

[R85] D. RUSSO, *Structural Properties of Complexity Classes*, Ph.D. thesis, Department of Mathematics, University of California, Santa Barbara, CA, March 1985.

[SB84] U. SCHÖNING AND R. BOOK, *Immunity, relativizations, and nondeterminism*, SIAM J. Comput., 13 (1984), pp. 329-337.

[W85a] G. WECHSUNG, *On the boolean closure of* NP, in Proc. 1985 International Conference on Fundamentals of Computation Theory, Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, 1985, pp. 485-493. (This paper was coauthored by K. Wagner.)

[W85b] ———, *On the boolean closure of* NP, Tech. Report N/85/44, Friedrich-Schiller-Universität, Jena, December 1985.

[Y85] A. YAO, *Separating the polynomial-time hierarchy by oracles*, in Proc. 26th Annual Symposium on Foundations of Computer Science, 1985, pp. 1-10.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 112-138, February 1989

008

ALGORITHMS AND DATA STRUCTURES FOR AN EXPANDED FAMILY OF MATROID INTERSECTION PROBLEMS*

GREG N. FREDERICKSON† AND MANDAYAM A. SRINIVAS‡

Abstract. Consider a matroid of rank n in which each element has a real-valued cost and one of $d > 1$ colors. A class of matroid intersection problems is studied in which one of the matroids is a partition matroid that specifies that a base has q_j elements of color j, for $j = 1, 2, \cdots, d$. Relationships are characterized among the solutions to the family of problems generated when the vector $(q_1, q_2, \cdots, q_d)$ is allowed to range over all values that sum to n. A fast algorithm is given for solving such matroid intersection problems when d is small. A characterization is presented for how the solution changes when one element changes in cost. Data structures are given for updating the solution on-line each time the cost of an arbitrary matroid element is modified. Efficient update algorithms are given for maintaining a color-constrained minimum spanning tree in either a general or a planar graph. An application of the techniques to the problem of finding a minimum spanning tree with several degree-constrained vertices is described.

Key words. data structures, degree-constrained spanning tree, matroid intersection, minimum spanning tree, on-line updating, partition matroid

AMS(MOS)subject classification. 68Q

1. Introduction.

Matroids are discrete mathematical structures that appear in a variety of applications. They are structures for which the greedy algorithm gives an optimal solution, and when intersected characterize such problems as minimum-weight maximum-cardinality bipartite matching [L1]. In this paper we study a class of combinatorial problems from a matroid point of view. Consider a matroid in which each element has a real-valued cost, and one of d colors, for some constant $d > 1$. Given positive integers $q_1, q_2, \cdots, q_d$, we seek a base of the matroid that is of smallest cost subject to the constraint that the base must contain q_j elements of color j, for $j = 1, 2, \cdots, d$. For example, we can generalize the minimum spanning tree problem to a problem in which the edges have colors, and we desire a spanning tree of minimum cost subject to constraints on the number of edges of each color that are in the tree.

A *matroid* M consists of a set E of elements, and rules describing a property, called *independence*, of certain subsets of E. The rules satisfy axioms that may be found in [L1], [W]. A maximal independent subset of E is called a *base*. A *matroid optimization problem* is the problem of finding a minimum cost base in a matroid in which a cost is associated with each element. For example, finding a minimum spanning tree of a connected graph is a matroid optimization problem, where the matroid consists of the set of edges in the graph, and independence corresponds to acyclicity. As stated above, matroid optimization problems can be solved by the greedy algorithm.

A *matroid intersection* involves two matroids defined on the same set E of elements, but with different sets of rules determining the independence of subsets in each matroid. A *matroid intersection problem* is an optimization problem whose solution is a subset of E of maximum cardinality that is independent in both matroids simultaneously, and is of minimum cost among all such subsets of E. There are algorithms for solving

* Received by the editors July 5, 1987; accepted for publication (in revised form) May 20, 1988.

† Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported in part by the National Science Foundation under grants DCR-8320124 and CCR-8620271, and by the Office of Naval Research under contract N00014-86-K-0689.

‡ Department of Computer Science, California State Polytechnic University, Pomona, California 91768. The research of this author was supported in part by the Defense Advanced Research Projects Agency under contract N00014-82-K-0193.

any given matroid intersection problem in polynomial time whenever independence of a set in the matroid can be tested in polynomial time [BCG1], [L2]. However the polynomial is large: at least $O(n^2m)$, where m is the number of elements, and n is the cardinality of the largest independent set. The special type of matroid intersection problem that we focus on in this paper is one in which each of the elements is labeled with one of d colors, and one of the matroids (a *partition* matroid) specifies that a certain number of elements of each color must be in the solution. For $d=2$ colors, very efficient special-purpose algorithms have been presented for a variety of problems in [GT], [G]. In this paper we explore the structure of d-color problems that allows for their efficient solution when $d>2$.

The solution techniques of [GT], [G] rely on finding a minimum cost solution from among only red elements and a minimum cost solution from among only green elements, and then pairing these red elements and green elements. However, for $d>2$ colors, the analogue of such a pairing does not seem to exist. We overcome this difficulty by generalizing other characterization results in [GT], [G]. We characterize the relationships among the solutions to a family of problems generated when the vector $(q_1, \cdots, q_d)$ is allowed to vary over all combinations that sum to n. The number of problems in this family is thus $(n+d-1)!/(n!(d-1)!)$, which is $\Theta(n^{d-1}/(d-1)!)$ for small d. The key relationship that we establish is the property of *dominance*, which allows us to search efficiently within the set of solutions to these problems. Dominance means that if one constrained minimum cost base dominates another with respect to the color constraints, then all elements of a certain color in the second base are in the first.

The dominance property makes possible a divide-and-conquer approach for finding a constrained minimum cost base that is efficient for small values of d. For a variety of matroids possessing certain desirable properties, the algorithm runs in time $O(dT_0(m, n)+(d!)^2T(n, 2))$, where $T_0(m, n)$ is the time to solve an uncolored version of the problem, and $T(n, 2)$ is the time to solve the 2-color version given a solution for each of the two colors. For graphic matroids, it has been shown in [FT], [GGST] that $T_0(m, n)$ is slightly larger than proportional to m, and in [GT] it has been shown that $T(n, 2)$ is $O(n \log n)$.[1] Our algorithm handles any d-color matroid intersection problem, such as scheduling unit-time jobs with integer release times and deadlines [GT], in essentially the same time bound. While the algorithm is factorial in d, it matches the bound in [GT] for $d=2$ and is significantly more efficient than the previously known algorithms when d is a small constant.

We also address the problem of updating a solution repeatedly, as the cost of elements changes one at a time. This on-line updating problem is a generalization of the 2-color update problem discussed in [FS]. We show how to use the dominance property to generate and maintain efficiently a sparse description of the $(n+d-1)!/(n!(d-1)!)$ solutions to all problems as the vector $(q_1, \cdots, q_d)$ ranges over all valid possibilities. We can update a d-color minimum spanning tree in $O(d^2m^{1/2}+d^{11/3}(d!)^2n^{1/3}\log n)$ time, and in $O(d^3(d!)^2(\log d)^{-1/2}2^{2(2\log(2d)\log n)^{1/2}}(\log n)^{3/2})$ time if the graph is planar. These match the update times in [FS] for the case when $d=2$.

Our d-color algorithm can be used to find a multiple-degree-constrained spanning tree of a communications network. Suppose the degrees of a number d of the nodes are prespecified, because of the number of ports that they have. When $d=1$, the problem is a special case of the 2-color minimum spanning tree problem [GT]. However, many interesting problem instances may require d degree-constrained nodes, where d

[1] All logarithms are to the base 2.

is a small constant greater than one. We reduce this problem to a set of $(d+1)$-color problems, one of which yields the solution. While the problem is NP-hard for general d [GJ, p. 206], our algorithm is efficient for small d. If the set of vertices for which there are degree constraints is an independent set, then finding a multiple-degree-constrained spanning tree is tractable, and an $O(n^3)$ algorithm exists [BCG2].

The remainder of the paper is organized as follows. In § 2 we introduce some terminology and new concepts that facilitate the later discussion. In § 3 we characterize the structure of d-color problem solutions, and establish the overall minimum cost, convexity, and dominance properties. In § 4 we apply these characterizations to develop an efficient divide-and-conquer algorithm for the static d-color problem, and illustrate its efficiency for graphic matroids. In §§ 5 and 6 we generalize the 2-color results of [FS] to d colors, and describe how to maintain a sparse description of certain arrangements of solutions to d-color problems to permit fast on-line update. In § 7 we discuss an application of our methods.

2. Definitions. We identify some additional matroid terminology; a more complete discussion can be found in [L1], [W]. The *rank* of a set $E' \subseteq E$, denoted as $rank(E')$, is the cardinality of a maximal independent subset of E'. Let B be a base, and f an element in $E-B$. The *circuit* $C(f, B)$ is the set consisting of every element that can be deleted from $B \cup \{f\}$ to restore independence. Let e be an element in B. The *cocircuit* $\bar{C}(e, B)$ is the set consisting of every element that restores rank to $B-\{e\}$. We will sometimes refer to an element in $C(f, B)-\{f\}$ as one *that f can replace in B*, and an element in $\bar{C}(e, B)-\{e\}$ as one *that can replace e in B*. Let M/E' denote the *contracted* matroid obtained from M by contracting the elements $E' \subset E$. The elements of M/E' are $E-E'$. Suppose E' is independent. Then the independent sets (bases) of M/E' are those sets $X \subset E-E'$ for which $X \cup E'$ is independent (a base) in M, and $rank(M/E') = rank(M) - rank(E')$.

For our problems on graphs, read *edge* for *element*, *spanning tree* for *base*, *cycle* for *circuit*, and *forest* for *independent set*. The *rank* is the number of edges in a spanning tree. Thus a minimum spanning tree is a minimum cost base of a graphic matroid. Similarly, for our unit-time job-scheduling problem, read *job* for *element*, a *set of jobs with a feasible schedule* for an *independent set*, a *maximal such set of jobs* for a *base*, and a *minimal infeasible set of jobs* for a *circuit*. Thus a maximum-profit set of jobs with a feasible schedule is a maximum-cost base of a job-scheduling matroid. Let $m = |E|$ and $n = rank(M)$.

We associate a *color* $j, j \in \{1, \cdots, d\}$ with each element in set E. For any set $E' \subset E$, let *colors* (E') be a d-tuple $(i_1, i_2, \cdots, i_d)$ giving the count of elements of each color in E'. Let $c_0(e)$ be the positive, real-valued cost of element e, and $c_0(E')$ the total cost of elements in a set E'. For a given cost function, we refer to a base B in such a matroid as *a constrained minimum cost base*, or a *minimum cost base for its vector colors*(B), if B is of minimum cost over all bases with the same *colors* vector. We assume that E has been augmented with elements of cost ∞ as necessary so that a base of each color $1, \cdots, d$ exists. Thus a *monochromatic minimum cost base* is a constrained minimum cost base whose *colors* vector has exactly one nonzero component.

Following [GT], we find it advantageous to extend the cost function so that each constrained minimum cost base B is unique for its vector *colors*(B). We make two different extensions, both similar to extensions given in [GT]. We assume that a unique index is associated with each element. Let $\alpha = \min(\{|c_0(E') - c_0(E'')|: E', E'' \text{ are sets of elements, } |E'| = |E''|, c_0(E') \neq c_0(E'')\} \cup \{c_0(e): e \text{ in } E\})$. We define $c(e) =$

$c_0(e)-\alpha/3^i$, where i is the index of e. By our choice of α, we note that for any two distinct bases B_1 and B_2, $c(B_1)\neq c(B_2)$, and for any three distinct bases B_1, B_2, and B_3, $2c(B_2)\neq c(B_1)+c(B_3)$.

The second extension $c_L(\cdot)$ of $c_0(\cdot)$ is based on lexicography. A real function $g(\cdot)$ is said to be *convex* if for any choice of values $x_1<x_2<x_3$, $(g(x_2)-g(x_1))/(x_2-x_1)\leqq(g(x_3)-g(x_2))/(x_3-x_2)$. Let $\bar{f}=(f_1(\cdot),f_2(\cdot),\cdots,f_d(\cdot))$ be a d-tuple of convex functions, and let π be any permutation on d-tuples. Let E' be a set of edges. We assume that $\bar{f}(colors(E'))$ yields d-tuple $(f_1(i_1),\cdots,f_d(i_d))$. Let *indices*$(E')$ be a sorted ordering of the indices of the elements in E'. Then we define $c_L(E')$ as the tuple $(c_0(E'),\ \pi(\bar{f}(colors(E'))),\ indices(E'))$. Comparisons between costs are resolved by lexicography on the tuples.

Note that for any two bases B_1 and B_2, $c_L(B_1)=c_L(B_2)$ implies that $B_1=B_2$. It is clear that for any two bases B_1 and B_2 with identical *colors* vectors, and any $\bar{f}$ and π, $c(B_1)<c(B_2)$ if and only if $c_L(B_1)<c_L(B_2)$. Thus a constrained minimum cost base under $c(\cdot)$ is a constrained minimum cost base under $c_L(\cdot)$. We find $c(\cdot)$ more convenient in proving several key properties about d-color matroids, and $c_L(\cdot)$ more appropriate to use when designing algorithms for d-color matroids. When the cost function ensures that there is a unique base of minimum cost over all bases with *colors* vector $\bar{i}$, we call this base $B_{\bar{i}}$.

We next define the notion of a uniform cost adjustment with respect to each of the extended cost functions. The notion of a uniform cost adjustment comes from [G], where it has been applied in handling 2-color matroids. A *uniform cost adjustment* with respect to $c(\cdot)$ consists of adding a constant δ_j to the cost of every element of color j in the matroid, for $j=1,2,\cdots,d$, and is specified by the d-tuple $\bar{\delta}$. A uniform cost adjustment with respect to $c_L(\cdot)$ consists of adjusting costs according to a d-tuple $\bar{\delta}$ and introducing a new d-tuple $\bar{f}$ of functions, along with permutation π. Since only differences in cost between elements of a particular color are significant in determining any constrained minimum cost base $B_{\bar{i}}$, the base $B_{\bar{i}}$ remains of minimum cost over the vector $\bar{i}$ after a uniform cost adjustment. Note that only differences in cost between various colors are significant in determining the relative costs of bases with different *colors* vectors. Furthermore, we can always assume without loss of generality that a uniform cost adjustment in a d-color matroid has at most $d-1$ nonzero components. The purpose of a uniform cost adjustment is to make some constrained minimum cost base $B_{\bar{i}}$ of overall minimum cost.

Let j_1 and $j_2\neq j_1$ be integers in $\{1,2,\cdots,d\}$. We say that a vector $\bar{i}'$ is a (j_1,j_2)-*neighbor* of $\bar{i}=(i_1,i_2,\cdots,i_d)$ if $i'_{j_1}=i_{j_1}-1$, $i'_{j_2}=i_{j_2}+1$, and $i'_j=i_j$ for all other j. Let the j_1-*negative neighbors* of $\bar{i}$ be the set of all (j_1,j_2)-neighbors of $\bar{i}$. Let the j_1-*positive neighbors* of $\bar{i}$ be the set of all (j_2,j_1)-neighbors of $\bar{i}$. When there is a unique minimum cost base for each vector $\bar{i}$, we extend the notion of neighbor from vectors to the bases that they index in the natural way. Let $\bar{i}$ and $\bar{i}'$ be the *colors* vectors of two bases. Suppose there is a *unique* color j for which $i_j>i'_j$. Then we say that $\bar{i}$ *dominates* $\bar{i}'$ *with respect to color* j, or that $\bar{i}$ j-*dominates* $\bar{i}'$.

Given a base B, a *swap* $s=(e,f)$ *available in* B is an ordered pair of elements, where $e\in B$, $f\notin B$, e and f are of different colors, and $C(f,B)$ contains e. Element f can be swapped in to replace element e, resulting in a base $B-\{e\}\cup\{f\}$ (denoted by $B\oplus s$ or $B-e+f$). Let S be a sequence of ordered element pairs $s_1,\cdots,s_r$, where each $s_i=(e_i,f_i)$. Given a base B, we say that S is *a swap sequence available in* B if s_1 is a swap available in B and if $r>1$ then $s_2,\cdots,s_r$ is a swap sequence available in $B\oplus s_1$. If S is a swap sequence available in B then $B\oplus S$ denotes the base obtained by applying S to B. Consider any cost function on E. Suppose swap sequence S is

available in a constrained minimum cost base B. Let $s_i = (e_i, f_i)$ for $i = 1, \cdots, r$. We say that the sequence S is *optimal* if bases $B \oplus s_1, \cdots, B \oplus s_1 \oplus \cdots \oplus s_r$ are all constrained minimum cost bases. The sequence S is *color-conserving* if $colors(f_i) = colors(e_{i+1})$ for $i = 1, \cdots, r-1$. The sequence S is *acyclic* if $colors(e_i) \neq colors(e_j)$ for $i, j \in \{1, \cdots, d\}$ and $i \neq j$. Finally, the sequence S is *regular* if it is optimal, acyclic, and color-conserving. Note that any subsequence of a regular swap sequence is regular. We refer to a regular swap sequence S with $colors(e_1) = j_1$ and $colors(f_r) = j_2$ as a *regular* (j_1, j_2) *sequence.*

Let D be a set of bases with distinct *colors* vectors. The set D is *tight* if, for every pair of bases B_1 and B_2 in D, B_1 and B_2 are neighbors. A tight set D with $|D| = k > 1$ is *negative* if colors $j_1, \cdots, j_k$ can be uniquely assigned to bases in D such that for any base B in D, if base B is assigned color j, then every base in $D - \{B\}$ is a j-negative neighbor of B. A *positive* tight set is defined analogously, using j-positive neighbors instead of j-negative neighbors. If $|D| = 1$, then we arbitrarily assign the single base in D the color 1, and call D negative. We say that $hue(B)$ is the color assigned to B, and for any subset D' of D, $hue(D') = \bigcup_{B \in D'} hue(B)$. Let D be a negative tight set, B a base in D with $colors(B) = \bar{i}$, and $r = \sum_{j \in hue(D)} i_j$. Let $hspan(D)$ be the set of bases with *colors* vectors $\bar{i}'$ such that $\sum_{j \in hue(D)} i'_j = r$, and $i'_j = i_j$ for $j \notin hue(D)$. A tight set D is *complete* if $|D| = d$. We denote the unique complete, negative, tight set associated with a base B and color j by $D(B, j)$. Note that if B, $B' \in D(B, j)$ and B' is B's (j, l) neighbor, then $D(B, j) = D(B', l)$.

Let D be a negative, tight set of bases. The *swap graph* G_D associated with D has vertex set D and contains an edge (B_1, B_2) if and only if bases B_1 and B_2 are related by a single swap. If every constrained minimum cost base is unique for its *colors* vector, then there is a close relationship between negative tight sets of minimum cost bases and regular swap sequences. If D is negative tight set of minimum cost bases and G_D is its swap graph, then every simple path in G_D corresponds to a regular swap sequence.

3. Characterization results. In this section we first give several properties of 2-color matroids identified in [GT], [G]. We then consider d-color matroids for $d > 2$ and establish the following important properties regarding constrained minimum cost bases and their neighbors which hold for the modified cost function $c(\cdot)$. First, there is a uniform cost adjustment that makes each constrained minimum cost base the overall (unconstrained) minimum cost base. Second, every pair of adjacent constrained minimum cost bases is related by a regular swap sequence of at most $d - 1$ swaps. Third, if the *colors* vector of one minimum cost base dominates that of another with respect to a certain color, then all elements of that color in the dominated base are contained in the dominating base. Finally, we characterize how a constrained minimum cost base changes when the cost of one element changes.

LEMMA 1 [GT, Thm. 3.1]. *Consider a matroid with elements of two colors, red and green. Consider any positive, real-valued cost function. Let B_i be a constrained minimum cost base with i red elements. Executing a lowest cost red-green swap available in B_i transforms B_i into a constrained minimum cost base B_{i+1} with $i+1$ red elements.* □

LEMMA 2 [GT, Cor. 3.3]. *Consider a matroid with elements of two colors, red and green. Consider any positive, real-valued cost function $c'(\cdot)$. Let B_{i-1}, B_i and B_{i+1} be constrained minimum cost bases with $i-1$, i and $i+1$ red elements, respectively. Then* $c'(B_i) - c'(B_{i-1}) \leqq c'(B_{i+1}) - c'(B_i)$. □

The following result is implicitly stated in [G]. We supply an explicit proof, using Lemma 2.

LEMMA 3. *Consider a matroid with elements of two colors, red and green. Consider any positive, real-valued cost function $c'(\cdot)$. Let B_i be a constrained minimum cost base*

with i red elements. There exists a uniform cost adjustment that makes the cost of B_i less than or equal to the cost of every other base.

Proof. Let l be the smallest index such that B_l exists, and u the largest index such that B_u exists. It is observed in [GT] that B_i exists for each i, $l \leqq i \leqq u$. Assume as boundary conditions that $c'(B_{l-1}) = 2c'(B_l) - c'(B_u)$ and $c'(B_{u+1}) = 2c'(B_u) - c'(B_l)$. Take $\delta_{\text{red}} = c'(B_{i-1}) - c'(B_i)$ and $\delta_{\text{green}} = 0$. It follows from Lemma 2 by induction that $c'(B_{i'}) \geqq c'(B_{i-1}) = c'(B_i) \leqq c'(B_{i''})$ for $l \leqq i' < i$ and $i < i'' \leqq u$. □

The following lemma, which is a variation of a lemma in [FS], establishes a fundamental property of bases in matroids.

LEMMA 4. *Let B be a base and e_1, e_2, f_1, f_2 be distinct matroid elements. Suppose $B - e_1 + f_1$ and $B - e_2 + f_2$ are bases, but $B - e_1 - e_2 + f_1 + f_2$ is not a base. Then both $B - e_1 + f_2$ and $B - e_2 + f_1$ are bases.*

Proof. The proof is similar to that of Lemma 3 of [FS]. □

We next present some lemmas that will be useful in the proof of the overall minimum cost and dominance theorems for matroids with elements of $d > 2$ colors. Lemma 5 establishes that if an overall minimum cost property holds for constrained minimum cost bases, then the convexity property holds. Lemma 6 shows that if an overall minimum cost property holds for a certain subset of constrained minimum cost bases centered on a negative tight set, then a stronger version of an overall minimum cost property holds. Lemma 7 establishes how the overall minimum cost property for a negative, tight set of constrained minimum cost bases impacts the connectedness of the corresponding swap graph. Finally, Lemma 8 uses the connectedness of the swap graph to establish the exact relationship between two neighboring constrained minimum cost bases for which the overall minimum cost property holds.

LEMMA 5. *Consider a matroid with elements of $d > 2$ colors. Let B_1, B_2, and B_3 be constrained minimum cost bases with respect to cost function $c(\cdot)$, such that B_2 is B_1's (j_1, j_2) neighbor and B_3 is B_2's (j_1, j_2) neighbor, for some j_1, j_2. Suppose each of B_1, B_2, and B_3 can be made an overall minimum cost base through some uniform cost adjustment. Then $c(B_2) - c(B_1) < c(B_3) - c(B_2)$.*

Proof. Suppose in contradiction that $c(B_2) - c(B_1) \geqq c(B_3) - c(B_2)$. Since B_1, B_2, and B_3 are distinct, this inequality must be strict, by definition of the modified cost function. Without loss of generality, suppose that B_1 is an overall minimum cost base. Let $\bar{\delta}$ be any cost adjustment vector that makes B_2 an overall minimum cost base. (By our initial assumption, $\bar{\delta}$ exists.) Make all the adjustments of $\bar{\delta}$, except those for colors j_1 and j_2. Note that the new costs $c'(B_1)$, $c'(B_2)$, and $c'(B_3)$ have the same relative values as $c(B_1)$, $c(B_2)$, and $c(B_3)$. Now make the adjustments for colors j_1 and j_2, yielding costs $c''(B_1)$, $c''(B_2)$, and $c''(B_3)$. Since B_2 becomes an overall minimum cost base, we must have $c'(B_2) - c'(B_1) \leqq \delta_{j_1} - \delta_{j_2}$. We also get $c''(B_3) - c''(B_2) = c'(B_3) - c'(B_2) - (\delta_{j_1} - \delta_{j_2})$, which by the preceding argument is less than $c'(B_2) - c'(B_1) - (\delta_{j_1} - \delta_{j_2})$, which is at most $\delta_{j_1} - \delta_{j_2} - (\delta_{j_1} - \delta_{j_2}) = 0$. Thus $c''(B_3) < c''(B_2)$, which contradicts our assumption that a suitable δ exists. □

Note that Lemma 5 will hold for any cost function $c'(\cdot)$ derived from $c(\cdot)$ by a uniform cost adjustment.

LEMMA 6. *Consider a matroid with elements of $d > 2$ colors. Let D be a negative, tight set of constrained minimum cost bases for cost function $c(\cdot)$. Suppose for each base B in hspan(D) there is a uniform cost adjustment that makes B an overall minimum cost base. Then there is a uniform cost adjustment that simultaneously makes every base in D of overall minimum cost and every base in hspan$(D) - D$ not of overall minimum cost.*

Proof. The proof is by induction on $p = |D|$. The basis case for $p = 1$ follows from our assumption that every base in *hspan*(D), and therefore every base in D, can

individually be made of overall minimum cost through a uniform cost adjustment. For the inductive step, with $p > 1$, assume that the lemma holds for any negative tight set D' of cardinality less than p. Let B_1 be a base in D, of hue j_1. Let B_2 be a second base in D, with hue $j_p \neq j_1$. Consider the negative, tight set of bases $D_1 = D - \{B_2\}$, which is of size $p-1$. Since $|D_1| < |D|$, by the induction hypothesis there is a uniform cost adjustment $\bar{\delta}$ that makes every base in D_1, but no other base in $hspan(D_1)$, of overall minimum cost. We next decrease the cost of color j_p so that the B_1 and B_2 are of the same cost, yielding uniform cost adjustment $\bar{\delta}'$ with respect to the original costs. This does not affect which bases in $hspan(D_1)$ are of minimum cost among those in $hspan(D_1)$, since all bases in $hspan(D_1)$ have the same number of elements of color j_p.

With respect to adjustment $\bar{\delta}'$, all bases in D have identical, though not necessarily overall minimum, cost. We claim that with respect to $\bar{\delta}'$, the bases in D are the *only* bases in $hspan(D)$ that are of minimum cost within $hspan(D)$. To prove the claim, we consider two cases. For $|D| = 2$, the claim follows directly from Lemma 5. For $|D| = p > 2$, consider the following. For any color k in $hue(D)$, let j_k be the minimum number of elements of color k in any base in D. (Note that the base of hue k in D will have $j_k + 1$ elements of color k, and all other bases in D will have j_k elements of color k.) Let c_m be the cost of each base in D.

Suppose there is some base B_3 in $hspan(D) - D$ with $c(B_3) \leqq c_m$. For some r in hue (D), B_3 has $j'_r < j_r$ elements of color r. Let D' be the set of all constrained minimum cost bases in D with exactly j_r elements of color r. We assert that with respect to adjustment $\bar{\delta}'$ all bases in $hspan(D') - D'$ have cost greater than c_m. We apply the inductive hypothesis to D' to prove the assertion. With respect to cost function $c(\cdot)$, there is a uniform cost adjustment $\bar{\delta}''$ that makes every base in D' of overall minimum cost, and every base in $hspan(D') - D'$ not of overall minimum cost. We argue that $\bar{\delta}'$ has the same effect as $\bar{\delta}''$ over the set of bases in $hspan(D')$. The adjustments in $\bar{\delta}''$ for colors not in hue (D') do not affect the relative costs of bases in $hspan(D')$ and can thus be equal to the corresponding values in $\bar{\delta}'$. Since bases in D' have identical cost under $\bar{\delta}'$, and also identical cost under $\bar{\delta}''$, then for any pair of colors k_1, k_2 in $hue(D')$, $\delta''_{k_1} - \delta'_{k_1} = \delta''_{k_2} - \delta'_{k_2}$. Subtracting $\delta''_{k_1} - \delta'_{k_1}$ from the adjustment δ''_k for each k in $hue(D')$ does not affect the relative costs of bases in $hspan(D')$, and gives $\bar{\delta}'$. Thus the adjustment $\bar{\delta}'$ has the same effect as $\bar{\delta}''$ over the set of bases in $hspan(D')$. We have proved the assertion that with respect to $\bar{\delta}'$, all bases in $hspan(D')$ have cost greater than c_m.

Now collapse all the hues in D *except* r to a new color s. Consider the set J of constrained minimum cost bases in this new matroid that have $1 + \sum_{k \in hue(D)} j_k$ elements of colors r and s combined. The base in J with j_r elements of color r has cost c_m, since the bases in $hspan(D') - D'$ have cost greater than c_m. The base in J with $j_r + 1$ elements of color r has cost at most c_m, since the base of hue r in D has cost c_m. By induction we can show that each base in J that has fewer than j_r elements of color r has cost greater than c_m, using Lemma 5. But the base in J with $j'_r < j_r$ elements of color r has cost at most c_m, since B_3 has cost at most c_m. Thus we achieve a contradiction, and prove the claim that with respect to $\bar{\delta}'$, the bases in D are the *only* bases in $hspan(D)$ that are of minimum cost within $hspan(D)$.

Finally, we make all colors in $hue(D)$ red, and the rest green. Note that one of the constrained minimum cost bases B_4 in this new problem is one of the bases of minimum cost in $hspan(D)$ under adjustment $\bar{\delta}'$. By Lemma 3, there is a uniform cost adjustment $(\gamma_{\text{red}}, \gamma_{\text{green}})$ that makes B_4 of overall minimum cost. We define the desired adjustment $\bar{\delta}'''$ from $\bar{\delta}'$ and $(\gamma_{\text{red}}, \gamma_{\text{green}})$ by adding γ_{red} to δ'_k for each k in $hue(D)$, and adding γ_{green} to δ'_k for each k not in $hue(D)$. The adjustment $\bar{\delta}'''$ will not alter the

relative costs of any bases in $hspan(D)$ under $\bar{\delta}'$, but will ensure that B_4, and thus all the bases in D, will be of overall minimum cost. □

LEMMA 7. *Consider a matroid M with elements of $d \geqq 2$ colors. Suppose that for any matroid M' with elements of $d' < d$ colors, and for any constrained minimum cost base B in M', there exists a uniform cost adjustment that makes B of overall minimum cost with respect to $c(\cdot)$ in M'. Let D be a complete negative tight set of constrained minimum cost bases with respect to $c(\cdot)$ in M. Let D_1 be a negative tight subset of D such that every base in $hspan(D_1)$ can be made of overall minimum cost through a uniform cost adjustment, and no base in $D - D_1$ can be made of overall minimum cost by a uniform cost adjustment. Then the swap graph G_{D_1} is connected.*

Proof. The proof is by induction on d. The basis is with $d = 2$. From Lemma 1, it is clear that the swap graph is connected. For the induction step, with $d > 2$, assume that for any matroid M' with elements of $d' < d$ colors, and sets D' and D_1' as specified, the swap graph $G_{D_1'}$ is connected. If $|D_1| = 1$, then G_{D_1} is connected. If $|D_1| > 1$, then consider a connected component D_2 in G_{D_1}.

We first argue that $|D_2| > 1$. Suppose $|D_2| = 1$. Let $B_1 \in D_2$, and without loss of generality assume that $hue(B_1) =$ green. Since $B_1 \in D_1$, we can adjust costs uniformly so that B_1 is a base of overall minimum cost. Temporarily change every color other than green to red, so that the resulting matroid has only red and green elements. Note that B_1 is the minimum cost base for its *colors* vector. By Lemma 1, B_1 is related by a swap to some constrained minimum cost base B_2 with one fewer green element than B_1. If we restore the original element colors, it is apparent that B_2 is in $D_1 - \{B_1\}$, since these are the only green-negative minimum cost neighbors of B_1. By the definition of swap graphs, D_2 should then include B_2, a contradiction. Thus $|D_2| > 1$

By Lemma 6, we can perform a uniform cost adjustment such that every base in D_2 is of overall minimum cost, and no other base in $hspan(D_2)$ is of overall minimum cost. We then change to green all colors in $hue(D_2)$. One of these bases, say B_1, will represent the component D_2 as a constrained minimum cost base in a matroid M' with $d - |D_2| + 1 < d$ colors. Clearly, $D' = D - D_2 \cup \{B_1\}$ is a complete negative tight set of bases M'. By assumption, for each constrained minimum cost base B in M', there exists a uniform cost adjustment that makes B of overall minimum cost with respect to $c(\cdot)$ in M'. Take $D_1' = D'$. Thus D_1' is a negative tight subset of D', and no base in $D' - D_1'$ can be made of overall minimum cost. Note that two bases in the same connected component of $G_{D_1'}$ will be in the same connected component of G_{D_1}. By the inductive hypothesis, $G_{D_1'}$ is connected. Since the bases in $D_1 - D_2 \cup \{B_1\}$ are in the same connected component of G_{D_1}, and the bases of D_2 are in the same connected component of G_{D_1}, G_{D_1} is connected. □

LEMMA 8. *Consider a matroid M with elements of $d \geqq 2$ colors. Suppose that for any matroid M' with elements of $d' < d$ colors, and any constrained minimum cost base B in M', there exists a uniform cost adjustment that makes B of overall minimum cost with respect to $c(\cdot)$ in M'. Let B_1 and B_2 be any two constrained minimum cost bases in M with respect to $c(\cdot)$ such that B_2 is B_1's j-negative neighbor, for some j. Let $B_2 \in D_1 \subseteq D(B_1, j)$. Suppose any base in $hspan(D_1)$ can individually be made of overall minimum cost through a uniform cost adjustment, and no base in $D(B_1, j) - D_1$ can be made of overall minimum cost by a uniform cost adjustment. Then B_1 and B_2 are connected by a regular swap sequence of length at most $d - 1$.*

Proof. Since $D_1 \subseteq D(B_1, j)$, the swap graph G_{D_1} has at most d vertices. By Lemma 7, G_{D_1} is connected. Thus there is a simple path p of length at most $d - 1$ between B_1 and B_2 in G_{D_1}. Let S be the corresponding swap sequence relating B_1 and B_2. Since p is acyclic and of length at most $d - 1$, so is S. Since D_1 is tight and negative, S is

color-conserving. Finally, since all bases in D_1 are constrained minimum cost bases, S is optimal. □

We now establish the overall minimum cost and dominance properties.

THEOREM 1 (Overall Minimum Cost). *Let M be a matroid with elements of d colors, $d > 1$. Let B be a constrained minimum cost base with respect to cost function $c(\cdot)$. There exists a uniform cost adjustment that makes B of overall minimum cost.*

Proof. The proof is by double induction, with the outer induction on d. The basis case, in which $d = 2$, follows from Lemma 3. For the inductive hypothesis, assume that the theorem is true for all matroids that have elements of at most $d-1$ colors. For the inductive step, consider a matroid of $d > 2$ colors. We prove the inductive step by induction on k, the number of elements of color 1. We will refer to color 1 as "*green.*"

For the inner basis, in which $k = 0$, we increase the cost of green elements by an amount sufficient to ensure that no constrained minimum cost base contains a green element. This is clearly equivalent to deleting every green element in the original matroid, obtaining a $(d-1)$-color matroid. The inner basis then follows from the outer inductive hypothesis. For the inner inductive hypothesis, assume that the theorem is true for all constrained minimum cost bases with at most $k-1$ green elements. For the inductive step, suppose $k > 0$.

Suppose the overall minimum cost property did not hold for some base B_1 with k green elements. We proceed to establish a contradiction. Consider the complete, negative, tight set $D(B_1, 1)$ and the negative, tight set $D_1 = D(B_1, 1) - \{B_1\}$. Every base in D_1 has $k-1$ green elements. By the inner inductive hypothesis, every base in $hspan(D_1)$ can be made of overall minimum cost. Thus by Lemma 6, we can adjust costs uniformly such that every base in D_1 is of identical, overall minimum cost in m, and no other base in $hspan(D_1)$ is of overall minimum cost. By temporarily changing every color other than green to red and applying Lemma 1, we conclude that for every base B in D_1 there is a base $mate(B)$ with k green elements such that B and $mate(B)$ are related by a swap. By Lemma 3, the cost of green elements can be uniformly adjusted, without disturbing the overall minimum cost property of any base in D_1, such that every base in $D_2 = \{mate(B) | B \in D_1\}$ is also of overall minimum cost. We have thus succeeded in uniformly adjusting costs such that every base in $D_1 \cup D_2$ is of identical, overall minimum cost. We now restore the original colors to the elements.

Now consider any base B_2 in D_1. Suppose B_2 is B_1's (green, red) neighbor, and $mate(B_2)$ is B_2's (blue, green) neighbor. (Since, by our assumption, B_1 cannot be made of overall minimum cost and $mate(B_2)$ can, $B_1 \neq mate(B_2)$ and therefore $mate(B_2)$ cannot be a (red, green) neighbor of B_2.) Let s_1 be the (blue, green) swap that transforms B_2 to $mate(B_2)$. Since B_2 and $mate(B_2)$ are of identical cost by our earlier cost adjustment, $c(s_1) = 0$.

We claim that swap s_1 is available in *any* base in D_1. In particular, s_1 is available in B_1's (green, blue) neighbor (and B_2's (red, blue) neighbor) B_3. This provides the desired contradiction: $B_3 \oplus s_1$ has the same color combination as B_1 and the same cost as B_3, which is of overall minimum cost. Thus $c(B_1) \leqq c(B_3)$, i.e., B_1 can be made of overall minimum cost through a uniform cost adjustment.

To prove the claim, we consider the regular (red, blue) swap sequence S_1 that, by Lemma 8, transforms B_2 into B_3. (The conditions of Lemma 8 apply by the inner and outer inductive hypotheses, and the assumption about B_1.) Let $|S_1| = p$. Note that every base in the sequence of bases induced by B_2 and S_1 is in D_1, and therefore every swap in S_1 is of zero cost. We establish by induction on p that s_1 remains available in a base B that is obtained from B_2 as a result of performing a sequence of p zero-cost swaps from a regular swap sequence.

The basis case for $p=0$ is trivial. For the inductive step, let $S_1 = S_2 s_2$, where S_2 is a regular (red, purple) swap sequence of length $p-1$ consisting of zero-cost swaps, and s_2 is a (purple, blue) zero-cost swap. By the inductive hypothesis, s_1 is available in $B_4 = B_2 \oplus S_2$, which is in D_1. Now suppose s_1 is not available in $B_3 = B_4 \oplus s_2$. Then, by Lemma 4, a (blue, blue) swap s_1' and a (purple, green) swap s_2' are available in B_4. Since $B_4 \in D_1$, it is of overall minimum cost. Therefore $c(s_1') \geqq 0$. Since $c(s_1') + c(s_2') = c(s_1) + c(s_2) = 0$, $c(s_2') \leqq 0$. Since $B_4 \oplus s_2'$ has the same color combination as B_1, it follows that $c(B_1) \leqq c(B_4 \oplus s_2') \leqq c(B_4)$, which is of overall minimum cost. By our assumption about B_1, this is impossible. Thus s_1 is available in B_3.

This completes the inductive step for k and the proof. □

THEOREM 2 (Dominance). *Let M be a matroid with elements of d colors, $d>1$. Let $B_{\bar{i}}$ and $B_{\bar{i}'}$ be constrained minimum cost bases with respect to $c(\cdot)$, such that $\bar{i}$ j-dominates $\bar{i}'$. Then every j-colored element in $B_{\bar{i}'}$ is in $B_{\bar{i}}$.*

Proof. If $d=2$, then the theorem follows from Lemma 1 and the fact that each constrained minimum cost base with respect to $c(\cdot)$ is unique for its *colors* index. If $d>2$, we can construct a sequence of $k = i_j - i_j' + 1$ constrained minimum cost bases $B_{\bar{i}}, \cdots, B_{\bar{i}'}$, such that each base in the sequence is a j-negative neighbor of its predecessor. Consider any two bases B_1 and B_2 that are consecutive in this sequence, with B_2 the j-negative neighbor of B_1. By Theorem 1, every constrained minimum cost base can be made of overall minimum cost by a uniform cost adjustment. By Lemma 7, B_1 and B_2 are connected by a regular swap sequence S. Since S is regular, it is acyclic, which implies that every element of color j in B_2 is in B_1. The theorem then follows by induction on k. □

To illustrate the properties of Theorems 1 and 2, we give an example of a graphic matroid. The edges of the graph will be of three different "colors," solid, dotted, and dashed. Figure 1 gives the graph in terms of the three subgraphs of each color. Each edge is labeled with its cost. In Fig. 2 we list the solutions to all possible subproblems, each labeled with its cost. For example, the solution with one solid, one dotted, and two dashed edges is the third solution in the fourth row, and is labeled with the cost

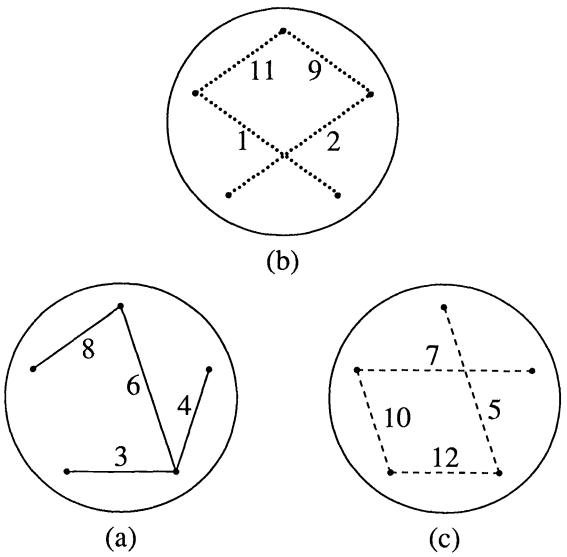


FIG. 1. *Subgraphs of a weighted graph with edges of three colors*: (a) *subgraph of solid edges*; (b) *subgraph of dotted edges*; (c) *subgraph of dashed edges.*

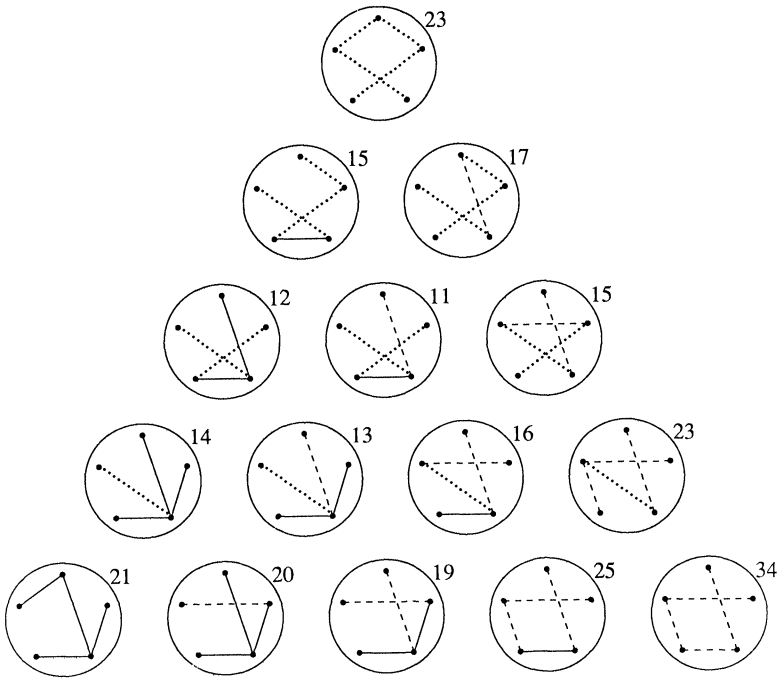


FIG. 2. *Solutions to all minimum spanning tree problems from Fig.* 1: *the tree with i solid edges, j dashed edges, and* $4-i-j$ *dotted edges is the* $(j+1)$*st tree in the* $(i+j+1)$*st row from the top.*

16. We illustrate the overall minimum cost property by making base $B_{\bar{i}}$ be the unconstrained minimum-cost base over all bases, where $\bar{i}$ is, for example, $(1, 1, 2)$. This can be done if we add 6 to the cost of every dotted element, and 4 to the cost of every solid element. To illustrate dominance, consider the solutions for $\bar{i} = (0, 1, 3)$ and $\bar{i}' = (1, 2, 1)$. (We assume that solid is color 1, dotted is color 2, and dashed is color 3.) Here $j_1 = 3$, i.e., there are fewer dashed elements in $B_{\bar{i}'}$ than in $B_{\bar{i}}$, and at least as many elements of every other color. Thus the one dashed edge (of cost 5) in $B_{\bar{i}'}$ is in $B_{\bar{i}}$.

Next we examine the impact of changing the cost of a single matroid element on a constrained minimum cost base. We begin as before with an earlier 2-color result, and proceed to generalize the result to $d > 2$ colors using the characterizations just developed.

LEMMA 9 [FS, Thm. 2]. *Let M be a matroid of red and green elements, with costs extended lexicographically to break ties. Let* B_{i-1}, B_i, *and* B_{i+1} *be the constrained minimum cost bases with* $i-1$, i *and* $i+1$ *red elements, respectively. If one element in M changes cost, then* B_i', *the new minimum cost base with i red elements, will result from either* B_{i-1}, B_i, *or* B_{i+1}, *with at most one element replaced in the appropriate base. Specifically, if a red element* r_t *increases in cost, then* B_i' *is the minimum cost base among the following three bases*:

(0) (*or* 3) B_i.

(1) $B_i - r_t + r_a$, *where* r_a *is the smallest cost red element that can replace* r_t *in* B_i.

(2) $B_{i+1} - r_t + g_a$, *where* g_a *is the smallest cost green element that can replace* r_t *in* B_{i+1}.

If a red element r_t *decreases in cost, then* B_i' *is the minimum cost base among the following three bases*:

(0) (*or* 3) B_i.

(1) *$B_i - r_a + r_t$, where r_a is the red element of greatest cost that r_t can replace in B_i.*

(2) *$B_{i-1} - g_a + r_t$, where g_a is the green element of greatest cost that r_t can replace in B_{i-1}.*

The cases for a green element changing in cost are analogous. □

We now give the generalization of the above result from two colors to d colors.

THEOREM 3. *Let M be a matroid with elements of d colors, $d > 1$. Let $B_{\bar{i}}$ be a constrained minimum cost base with respect to cost function $c(\cdot)$. If one element in M changes cost, then the new minimum-cost base $B'_{\bar{i}}$ will result from either $B_{\bar{i}}$ or one of its neighbors, with at most one element replaced in the appropriate base. Specifically, if a basic (respectively, nonbasic) element e (f) of color j_1 increases (decreases) in cost, then one of the following cases holds:*

(0) *The new base $B'_{\bar{i}} = B_{\bar{i}}$.*

(1) *$B'_{\bar{i}} = B_{\bar{i}} - e + f$, where e, f both have color j_1 and f (e) is the least (greatest) cost element of color j_1 that can replace e (be replaced by f) in $B_{\bar{i}}$.*

(2) *There is a color $j_2 \neq j_1$ such that $B'_{\bar{i}} = B_{\bar{i}'} - e + f$, where $\bar{i}'$ is a ($color(f)$, $color(e)$)-neighbor of $\bar{i}$ and f (e) is the least (greatest) cost element of color j_2 that can replace e (be replaced by f) in $B_{\bar{i}'}$.*

Proof. We first consider the case where a basic element e of color j_1 increases in cost. By Theorem 1 we can make $B_{\bar{i}}$ the unconstrained minimum-cost base, and therefore also the minimum-cost base over all bases with exactly i_{j_1} elements of color j_1, by uniformly adjusting the costs of all elements of colors $j \neq j_1$. Temporarily change the color of all j_1-colored elements to red and all other elements to green, so that $B_{\bar{i}}$ corresponds to red-green base $B_{i_{j_1}}$. We can then apply Lemma 9 with e in the role of r_t. If case (0) or (1) of Lemma 9 holds, then the corresponding case of our theorem holds. If case (2) of Lemma 9 holds, then there is a red-green base $B_{i_{j_1}+1}$ that differs from $B'_{i_{j_1}}$ by one element, g_a. Let f be the element corresponding to g_a in the original matroid, and let $j_2 = color(f)$. Since g_a is the least cost replacement element over all green elements, f is certainly the least cost replacement element of color j_2.

The symmetric case of a nonbasic element f decreasing in cost is handled similarly. □

Note that Theorems 1, 2, and 3 hold if cost function $c_L(\cdot)$ replaces cost function $c(\cdot)$ in the statement of the theorem. The use of $c_L(\cdot)$ has the advantage that arbitrarily many updates can be performed. This is not true for $c(\cdot)$, since changing the cost of one element can affect the value of α, which will alter the cost of every element.

4. Efficient solution of the static problem. We show how to find the constrained, lexicographically minimum cost base $B_{\bar{q}}$ consisting of q_j elements of color j, for $j = 1, 2, \cdots, d$, along with a uniform cost adjustment vector $\bar{\delta}$ that makes $B_{\bar{q}}$ of overall, unconstrained minimum cost. For matroids satisfying certain desirable properties, the time to do this will be $O(dT_0(m, n) + (d!)^2 T(n, 2))$, where $T_0(m, n)$ is the time to solve an uncolored, or monochromatic, problem, and $T(n, 2)$ is the time to solve a 2-color problem, given the constrained minimum cost bases for each color. Our algorithm DCOLOR first augments the set of elements with elements of large cost as necessary so that there is a base of each color, and finds monochromatic minimum cost bases for each color. This step accounts for the first term of the running time expression. Algorithm DCOLOR then calls a recursive routine DREC that is supplied with the d monochromatic bases and finds the desired base and associated vector $\bar{\delta}$. The call to DREC accounts for the second term in the running time expression.

Our presentation is organized as follows. We first review the 2-color algorithm of [GT], and explain how $\bar{\delta}$ can be computed in this case. We then augment the 2-color algorithm of [GT] with lexicographic cost comparisons to help handle calls from our

d-color recursive routine. We finally present our recursive routine DREC to find a d-color base.

The 2-color algorithm in [GT] is designed to find a minimum cost base constrained to have exactly s red elements, for some s. The algorithm calls a recursive routine to identify what is called a *restricted* swap sequence, which transforms a constrained minimum cost base of green elements to a constrained minimum cost base of red elements. The restricted swap sequence contains swaps in order of nondecreasing cost of the red element in each swap. The algorithm then sorts the swaps in order of nondecreasing cost of the swaps to yield an optimal swap sequence. The algorithm forms the desired base by taking the first portion of the swap sequence and applying it to the green constrained minimum cost base. Since the cost of a minimum cost base with i red elements is a convex function of i, the vector $\bar{\delta}$ can be readily determined by comparing the cost of swaps adjacent to the desired base.

We augment the algorithm to enforce a lexicographic tie-breaking scheme. In addition to its color, let each element have a unique index. Assign a *tag* to each element consisting of the pair (*j*, *index*), where j is the original color of the element. Ties in element costs are broken lexicographically using element tags. Ties in the costs of swaps are broken lexicographically as follows. Consider two swaps (e,f) and (e',f') of equal cost. Swap (e,f) will be lexicographically less than (e',f') if and only if either f or e' has the lexicographically smallest tag from among e, f, e', and f'. We can incorporate this lexicographic tie-breaking scheme into the 2-color algorithm of [GT] at constant cost for any comparison of two elements or two swaps.

We now describe our recursive routine DREC to find a d-color base. The input to this routine is a vector $\bar{q}$ and the set of dn elements that is the union of the d monochromatic bases. The routine uses a divide-and conquer approach, recursing first on fewer colors, and then again on fewer elements. The basis cases occur when either $d=2$ or $n<2d^2(d-1)$. If $d=2$ we use the augmented 2-color algorithm. We will discuss the other basis case later. If $d>2$ and $n\geqq 2d^2(d-1)$, we do the following. Order the colors so that $q_j \leqq q_{j+1}$, for $j=1, 2, \cdots, d-1$. Find the constrained minimum cost base $B_{\bar{i}}$, where $i_j = q_j + \lfloor (q_d + j - 1)/(d-1) \rfloor$ for $j=1, 2, \cdots, d-1$, and $i_d = 0$. This is a problem in $d-1$ colors, and is solved recursively by our routine. Note that $\bar{i}$ is defined so that for each color $j \neq d$, $B_{\bar{i}}$ has at least $\lfloor n/(d(d-1)) \rfloor$ more elements of color j than $B_{\bar{q}}$. Along with determining $B_{\bar{i}}$, the recursive call will supply the corresponding values $\delta(j)$, for $j=1, \cdots, d-2$ that make $B_{\bar{i}}$ of minimum cost among bases with no elements of color d.

Once $B_{\bar{i}}$ and $\bar{\delta}$ have been determined, temporarily add $\delta(j)$ to the cost of each element of color j in $B_{\bar{i}}$, for $j=1, \cdots, d-2$. Define $\bar{f}$ such that for any d-tuple $\bar{i}'$, $f_j(i'_j) = |i'_j - i_j|$, for $j=1, \cdots, d$. Note that by their definition the functions $f_j(\cdot)$ are convex. For any choice of π, $B_{\bar{i}}$ will be the minimum cost base among those with no elements of color d, with respect to the adjusted version of the cost function $c_L(\cdot)$, defined earlier.

Relabel the elements of base $B_{\bar{i}}$ with the color green, and label with the color red the elements in the constrained minimum cost base of color d. Now use the 2-color algorithm of [GT], augmented to use tags lexicographically to break ties in the costs of elements and swaps, to find the constrained minimum cost base B' that has $\lfloor q_d/(d-1) \rfloor - 1$ red elements and the rest green. Even though colors are reordered to satisfy $q_j \leqq q_{j+1}$, a permutation π can be chosen that undoes this reordering, and hence makes the use of the tags enforce $c_L(\cdot)$. Thus any base generated by the augmented 2-color algorithm will be a constrained minimum cost base with respect to $c_L(\cdot)$, and thus also $c(\cdot)$, in the original d-color matroid.

If we switch the elements in B' back to their original colors, we get a base $B_{\bar{k}}$ in which $k_d = \lfloor q_d/(d-1) \rfloor - 1$ and $k_j \geqq q_j + 1$ for $j = 1, 2, \cdots, d-1$. It is clear that the set of color vectors consisting of $\bar{q}$ and its immediate neighbors dominate $\bar{k}$ with respect to color d. By our dominance theorem, every element of color d in $B_{\bar{k}}$ is in $B_{\bar{q}}$, and also in every constrained minimum cost base that is an immediate neighbor of $B_{\bar{q}}$. Let D be the set of these elements of color d. Contract the matroid on set D, and decrease q_d by $|D|$. Since $q_d \geqq \lceil n/d \rceil$, the number of elements is reduced by at least $\lfloor \lceil n/d \rceil / (d-1) \rfloor - 1$, which is at least $\lceil n/d^2 \rceil$ if $n \geqq 2d^2(d-1)$. For $d > 2$ and $n \geqq 2d^2(d-1)$, note that the new value of q_d will be greater than 0. Solve the resulting smaller d-color problem recursively, yielding $\hat{B}$ and $\bar{\delta}$. Form $\hat{B} \cup D$, and return this set with $\bar{\delta}$ as the solution to the call on DREC.

We justify the contraction and union steps in the previous paragraph as follows. Let M/D be the contracted matroid. Note that $D \subset B_{\bar{q}}$, and $B_{\bar{q}} - d$ is a base in M/D. Let B be a base in M/D with the same index vector as $B_{\bar{q}} - D$ but not equal to $B_{\bar{q}} - D$. Now $c(B) > c(B_{\bar{q}} - D)$, since otherwise $B \cup D$ would be a base of M with index vector $\bar{q}$ but of smaller cost than $B_{\bar{q}}$, a contradiction to the definition of $B_{\bar{q}}$. We make use of the requirement that D be a subset of each neighbor of $B_{\bar{q}}$ in the following way. If $\hat{B}'$ is a neighbor of $B_{\bar{q}} - D$ in M/D, then $\hat{B}' \cup D$ will be the corresponding neighbor of $B_{\bar{q}}$ in M. This guarantees that the uniform cost adjustment $\bar{\delta}$ that makes $B_{\bar{q}} - D$ of overall minimum cost in M/D will also make $B_{\bar{q}}$ of overall minimum cost in M.

We now discuss the other basis case, when $n < 2d^2(d-1)$. Here we use the weighted matroid intersection algorithm [BCG1] to find $B_{\bar{q}}$ directly. We also need to determine the $\delta(j)$ values. This can be done by considering each of the elements not in $B_{\bar{q}}$. For each such element f, find the best swap in $B_{\bar{q}}$ for each color $j \neq color(f)$. We infer the values of $\delta(j)$ from the thresholds of these swaps as follows. Each best swap (e, f) yields a constraint $\delta(color(e)) - \delta(color(f)) \leqq c(f) - c(e)$. Choosing the $\delta(j)$'s then reduces to the following shortest path problem. Consider a graph with d vertices labeled from 1 to d. For each constraint $\delta(j_1) - \delta(j_2) \leqq c_{j_1 j_2}$ there is an edge from j_2 to j_1 of cost $c_{j_1 j_2}$. In the case of multiple edges, only the shortest edge is retained. Then choosing $\delta(j)$ to be the shortest distance from vertex d to vertex j, for all j, will give a consistent set of deltas. The shortest distances can be determined in $O(d^3)$ time using the Bellman-Ford algorithm in [L1]. This completes our presentation of the recursive routine DREC for the d-color algorithm.

LEMMA 10. *Let M' be a matroid of elements of $d > 2$ colors, that is comprised of the union of d monochromatic bases. Let $\bar{q}$ be a valid index for a base in M'. Routine* DREC *correctly computes a minimum cost base $B_{\bar{q}}$ and a uniform cost adjustment $\bar{\delta}$ that makes $B_{\bar{q}}$ of overall minimum cost in M'.*

Proof. Correctness can be established with a proof by induction. That the two basis cases are correct follows from the correctness of the algorithms in [GT], [BCG1] and the additional comments in the text. The correctness of the routine for the nonbasis case follows from the arguments that the set D of elements contracted is nonempty and is contained in $B_{\bar{q}}$ and each of its neighbors. Thus the solution $(\hat{B}, \bar{\delta})$ to the contracted problem can be augmented to $(\hat{B} \cup D, \bar{\delta})$, the solution to the given problem. □

We next discuss the running times of DREC and DCOLOR. The efficiency of DREC (and thus DCOLOR) depends on whether the matroid M under consideration possesses certain nice properties. Let $T(n, 2)$ be the time to solve the 2-color problem in M with elements recolored to just two colors, when the minimum-cost bases of each of the two colors are given. We identify the following properties as *desirable.*

(1) Independence testing in M is polynomial.

(2) The time to contract dn elements in m is $O(dT(n, 2))$.

(3) For $8/9 \leqq a < 1$ and $n \geqq 4/(1-a)$, $T(\lfloor an \rfloor, 2) \leqq aT(n, 2)$.

We note that the matroids handled in [GT], [G] possess the desirable properties. In particular, we discuss the motivation for assuming the bound of $dT(n, 2)$ on the time to contract $O(dn)$ elements in a matroid. By assigning color $d+1$ to each element to be contracted and solving d 2-color problems involving color $d+1$ and each original color, we can determine the elements in each monochromatic base in the contracted matroid. The correctness of this reduction follows from the definition of matroid contraction. It is also necessary to determine the new attributes of each element (e.g., endpoints of an edge in the case of a graphic matroid) in the contracted matroid. For all the matroids discussed in [GT], this can be done for each new base within time proportional to $T(n, 2)$.

THEOREM 4. *Let M be a matroid of rank n with m elements of $d > 2$ colors. Let $T_0(m, n)$ be the time to solve the uncolored (monochromatic) problem in M. Let $T(n, 2)$ be the time to solve the 2-color problem in M with elements recolored to just two colors, when the minimum-cost bases of each of the two colors are given. If M has the desirable properties, then the time to solve a d-color problem in M is $O(dT_0(m, n) + (d!)^2 T(n, 2))$. The space required is $O(d^3 n)$.*

Proof. Let $T(n, d)$ be the time to solve a d-color problem in a matroid of rank n, given that the monochromatic bases are provided. The intersection algorithm in [BCG1] uses $O(nm(n + I(m) + \log m))$ time, where $I(m)$ is the time to test independence. By assumption, $I(m) = m^k$ for some k. Since $m = nd$, this takes $O(d^4 n(d^3 + d^{4k}))$ time, which is $O(d^4(d^3 + d^{4k})T(n, 2))$, since $T(n, 2) \geqq n$. Finding the swaps to identify $\delta(j)$ values involves examining $O(d^4)$ elements f, at $O(d^3)$ time per element f, or $O(d^7)$ time altogether. For $n \geqq 2d^2(d-1)$, all work except for the recursive call on $d-1$ colors and the recursive call on fewer elements is $O(dT(n, 2))$. Thus for $d > 2$ we have the following recurrence:

$$T(n, d) \leqq c_1(d^7 + d^{4+4k})T(n, 2) \quad \text{for } n < 2d^2(d-1),$$

$$T(n, d) \leqq c_2 dT(n, 2) + T(n, d-1) + T(\lfloor n(1-1/d^2) \rfloor, d) \quad \text{for } n \geqq 2d^2(d-1)$$

where the c_i's are constants. We claim that

$$T(n, d) \leqq (c_3(d!)^2 - c_4 d)T(n, 2)$$

for $c_4 = 2c_2$ and $c_3 = (c_1 + c_2)c_5$, where c_5 is the maximum value of $(d^7 + d^{4+4k})/(d!)^2$ when d is chosen over the positive integers.

The proof is by double induction, with the outer induction on d, and the inner induction on n. For $d = 3$, we prove the claim by induction on n. For $n < 2d^2(d-1)$, $T(n, 3) \leqq c_1(3^7 + 3^{4+4k})T(n, 2) \leqq (c_3(3!)^2 - 3c_4)T(n, 2)$, for the choices of c_3 and c_4. For $n \geqq 2d^2(d-1)$,

$$T(n, 3) \leqq 3c_2 T(n, 2) + T(n, 2) + T(\lfloor 8n/9 \rfloor, 3)$$

which by the induction hypothesis is

$$\leqq (3c_2 + 1)T(n, 2) + (c_3(3!)^2 - 3c_4)T(\lfloor 8n/9 \rfloor, 2)$$

$$\leqq (3c_2 + 1)T(n, 2) + (36c_3 - 3c_4)(8/9)T(n, 2)$$

$$\leqq (36c_3 - 3c_4)T(n, 2)$$

for the choices of c_3 and c_4.

For $d > 3$, we assume as the outer induction hypothesis that the claim is true for all d', $3 \leqq d' < d$. We prove the claim by induction on n. For $n < 2d^2(d-1)$, $T(n, d) \leqq c_1(d^7 + d^{4+4k})T(n, 2) \leqq (c_3(d!)^2 - c_4 d)T(n, 2)$, for the choices of c_3 and c_4. For $n \geqq 2d^2(d-1)$, we assume as the inner induction hypothesis that the claim is true for all $n' < n$. We have

$$T(n, d) \leqq c_2 dT(n, 2) + T(n, d-1) + T(\lfloor n(1-1/d^2) \rfloor, d)$$

which by the inner and outer induction hypotheses is

$$\begin{aligned}
&\leqq c_2 dT(n, 2) + (c_3((d-1)!)^2 - c_4(d-1))T(n, 2) \\
&\quad + (c_3(d!)^2 - c_4 d)T(\lfloor n(1-1/d^2) \rfloor, 2) \\
&\leqq c_2 dT(n, 2) + (c_3(d!/d)^2 - c_4(d-1))T(n, 2) \\
&\quad + (c_3(d!)^2 - c_4 d)(1-1/d^2)T(n, 2) \\
&\leqq (c_3(d!)^2 - c_4 d + c_2 d + c_4 - c_4 d(1-1/d^2))T(n, 2) \\
&\leqq (c_3(d!)^2 - c_4 d)T(n, 2)
\end{aligned}$$

for the choice of c_4. This completes the inner induction, and then the outer induction.

As for the space required, either basis case will take $O(dn)$ space. For the nonbasis case, the space will satisfy the recurrence

$$S(n, d) \leqq \max\{n, d + S(n, d-1), dn + S(\lfloor n(1-1/d^2) \rfloor, d)\}.$$

The second term represents the space to store the color requirements for the base with $d-1$ colors and then to compute the base. The third term represents the space to represent the contracted matroid and then to compute a base in it. The solution to this recurrence is $O(d^3 n)$. □

Even though the running time involves factorials in terms of d, it is better than the running time for the weighted matroid intersection algorithm of [BCG1] whenever d is $o((\log n)/(\log \log n))$.

We suggest a modification to the algorithm that may yield a faster algorithm in practice. The 2-color algorithm in [GT] generates in succinct form the sequence of constrained minimum cost bases between the base of all one color and all the other color. Instead of specifying the number of elements of color d that we want in B', we take the swap sequence generated, switch back to original colors and find the furthest base $B_{\bar{k}}$ represented in the swap sequence such that $k_j \geqq q_j + 1$, for $j = 1, \cdots, d-1$. At least as many elements will be contracted as before.

Finally, as an illustration, we apply the above algorithm to graphic matroids. Here $T_0(m, n)$ is $O(m \log \beta(m, n))$ by the algorithm of [GGST], where $\beta(\cdot, \cdot)$ is a certain slowly growing function [FT]. $T(n, 2)$ is $O(n \log n)$ by the algorithm of [GT]. Independence is equivalent to acyclicity, and thus independence can be tested in $O(m)$ time. Contracting $O(dn)$ elements can be implemented in $O(dn)$ time. Therefore the time to find a constrained minimum cost spanning tree is $O(dm \log \beta(m, n) + (d!)^2 n \log n)$.

5. Basic on-line update strategy. In this section we give a basic description of our data structures for on-line updating of a constrained minimum cost base in a d-color matroid. This work is an extension of the updating approach in [FS] that handled 2-color problems. Let $B_{\bar{i}}^{(h)}$ represent the minimum cost base for *colors* vector $\bar{i}$ after h element cost updates have been performed. We first discuss data structures that allow us to find quickly base $B_{\bar{i}}^{(h+1)}$ given $B_{\bar{i}}^{(h)}$ and all of its neighbors after h updates. This operation, which relies on Theorem 3, is crucial to our on-line update technique. However, to compute $B_{\bar{i}}^{(h+2)}$ by this method, we need to have $B_{\bar{i}}^{(h+1)}$ and its neighboring

bases after $h+1$ updates, which in the worst case means we must have $B_{\bar{i}}^{(h)}$, its neighbors after h updates, and also the neighbors' neighbors after h updates. We therefore discuss how to maintain larger groups of neighboring bases, and introduce the notion of an *arrangement* of bases, generalizing the sequences employed in the 2-color algorithm. Since updating large groups of bases directly would be quite inefficient, we then discuss maintaining arrangements in an implicit form, which allows for efficient updating. Finally, we illustrate the technique with the example of a graphic matroid. Although our presentation of the d-color update technique is sufficiently detailed to be self-contained, familiarity with the 2-color update technique of [FS] will help greatly in understanding the details.

We recall from [FS] the definition of an *update structure* for a base in a matroid with uncolored elements. An update structure for a base B is a data structure that supports the following operations:

maxcirc(f, B): finds the maximum cost element in the circuit $C(f, B)$.
mincocirc(e, B): finds the minimum cost element in the cocircuit $\bar{C}(e, B)$.
swap(e, f, B): converts the update structure for B into an update structure for $B-e+f$ (assuming that $f \in \bar{B}$ and $e \in C(f, B)$).

Let $U(m, n)$ represent the maximum of the execution times of these three operations for a particular matroid. Thus a minimum cost base in a matroid with uncolored elements can be updated in time at most $2U(m, n)$ when the cost of a single matroid element is modified. Let $S(m, n)$ be the space required by the update structure.

In the case of a matroid with elements of d colors, the update structure is generalized to allow the color of the appropriate element to be specified. Thus for $j=1, 2, \cdots, d$, the operation *maxcirc*(j, f, B) finds the maximum cost element of color j in $C(f, B)$, and *mincocirc*(j, e, B) finds the minimum cost element of color j in $\bar{C}(e, B)$. The operation *swap*(e, f, B) is as before. The generalized update structure for d-colored matroids can be derived from the corresponding structure for uncolored matroids in a straightforward manner. For each field relating to costs in the uncolored update structure, maintain d fields in the new structure, with the jth field accessed for operations on color j. The values in the fields should be such that the cost of an element not of color j should be treated as $-\infty$ in handling a *maxcirc*$(j, \cdot, \cdot)$, and ∞ in handling a *mincocirc*$(j, \cdot, \cdot)$. The space requirement of the generalized update structure is then $O(dS(m, n))$.

Using Theorem 3, a generalized update structure can be used to find an updated base $B_{\bar{q}}^{(h+1)}$ from $B_{\bar{q}}^{(h)}$ and its neighbors after h updates. For instance, if a basic element e increases in cost, then $B_{\bar{q}}^{(h+1)}$ would be the least cost base in the set consisting of $B_{\bar{q}}^{(h)}$ and $B_{\bar{i}}^{(h)}-e+$ *mincocirc*$(j, e, B_{\bar{i}}^{(h)})$, where either the color of e is not j, and $B_{\bar{i}}$ is a neighbor of $B_{\bar{q}}$ containing one element fewer of color j, or j is the color of e, and $B_{\bar{i}}$ is $B_{\bar{q}}$. If a cobasic element f decreases in cost, then $B_{\bar{q}}^{(h+1)}$ would be the least cost base in the set consisting of $B_{\bar{q}}^{(h)}$ and $B_{\bar{i}}^{(h)}-$ *maxcirc*$(j, f, B_{\bar{i}}^{(h)})+f$, where either the color of f is not j, and $B_{\bar{i}}$ is a neighbor of $B_{\bar{q}}$ containing one more element of color j, or j is the color of f, and $B_{\bar{i}}$ is $B_{\bar{q}}$. The update is concluded by performing the appropriate *swap*.

As stated at the beginning of the section, maintaining just $B_{\bar{q}}^{(h)}$ and its neighbors after h updates is not enough, since there is not sufficient information to compute efficiently all neighbors of $B_{\bar{q}}^{(h+1)}$ after $h+1$ updates. For $l>0$, let $R_{\bar{i},l}$ be the set of bases $\{B_{\bar{i}'} | i_j' \leqq i_j+l-1, j=1, 2, \cdots, d\}$. We shall represent groups of bases in sets such as $R_{\bar{i},l}$, which we call *arrangements*. We say that arrangement $R_{\bar{i},l}$ is *centered on* $\bar{i}$ and has *radius* l. Our update procedure is periodic with period z. By this we mean that for

the hth element cost change the update procedure handles data in the same form (e.g., radius of arrangement) as the data during the $(h+z)$th element cost change, for any $h>0$. Here, z is a parameter that will be specified later, when we discuss the running time. Our update procedure consists of three parts. For clarity, we will uncover the parts one by one.

Consider h to be an integer in the range from 0 to z. Suppose after the hth update we keep an arrangement $A_0^{(h)} = R_{\bar{q},z-h}^{(h)}$. The superscript on R and on A_0 indicates how many element cost changes have been supplied, and will be omitted unless the context demands it. As long as $h<z$, there is sufficient information to generate $R_{\bar{q},z-h-1}^{(h+1)}$, no matter what type of element cost change occurs. Thus $z-1$ element cost changes can be successfully handled, but when the zth update occurs, $B_{\bar{q}}$ is lost. This follows, since $A_0^{(z-1)}$ is an arrangement consisting of one base $B_{\bar{q}}^{(z-1)}$, so there is insufficient information remaining in order to compute $B_{\bar{q}}^{(z)}$. We say that A_0 *decays* during this sequence of z updates. Of course, for large z, explicitly maintaining and updating the arrangement A_0 requires considerable time per cost change. In due course, we will show how to circumvent this problem by introducing an implicit representation for A_0.

When A_0 has completely decayed, we need to replace it with an arrangement containing many bases. But this means that certain work must be done in advance. We therefore discuss the second part of our solution. Thus we now consider unrestricted values of h. Whenever A_0 is initialized, i.e., $h \bmod z = 0$, we initiate a computation to solve a number of d-color problems on the current matroid, in order to generate a new arrangement of bases, given the minimum cost base after h updates containing only elements of color j, for $j=1,2,\cdots,d$. Note that any constrained base after h updates contains only elements from the union of these monochromatic bases. Let $P(n, d)$ be the time required to determine for a given d-color problem an arrangement of bases in an appropriate form. Assume that copies of the d monochromatic bases are maintained from one update to the next. Since just one of these monochromatic bases changes, a cost of $U(m, n)$ is charged to the update. Each static d-color problem will be solved during the time in which A_0 decays, by performing $O(P(n, d)/z)$ work over each of z update steps.

However, when all static d-color problems are completed, after $h=kz$ updates, we cannot just reconstitute A_0 with the appropriate bases. This is because each such base will be *out of date* by z element cost changes, since the element costs used in solving the static problems were extracted after $(k-1)z$ updates, and z further element cost updates have been applied to the matroid in the meantime. Thus we introduce the third part of our update strategy. We use a second arrangement A_1, centered at $B_{\bar{q}}$ and initially with $l=3z$, which is extracted from the out-of-date solution to the static d-color problems. Thus when A_1 is created after $h=kz$ updates have occurred, we have $A_1^{(h)} = R_{\bar{q},3z}^{(h-z)}$.

Since the bases in $A_1^{(kz)}$ will initially be out of date with respect to $A_0^{(kz)}$ by z element cost changes, we need to bring them up to date over the next z update steps of A_0, using the z element cost changes that have not yet been applied to A_1. These previous element cost changes can be saved in a queue as the static d-color problems are being solved. Thus, when $A_1^{(kz)}$ is created, the queue will contain element cost changes numbered $(k-1)z+1, (k-1)z+2, \cdots, kz$. Consider the hth update step, that transforms $A_1^{(h-1)}$ to $A_1^{(h)}$, Let $h=kz+r$, where $0<r\leqq z$. We first add the hth element cost change to the rear of the queue. We then delete the two element cost changes (namely, those numbered $h-z+r-1$ and $h-z+r$) for the front of the queue and apply them both to $A_1^{(h-1)}$, obtaining $A_1^{(h)}$. Thus $A_1^{(h)}$ will be the arrangement $R_{\bar{q},3z-2r}^{(h-z+r)}$. A_1 will then become up to date with respect to A_0, and also be of the correct radius,

precisely when A_0 has completely decayed. We then replace A_0 by the current arrangement A_1.

We can view our three-part update technique as three concurrent processes going on at once. Times at which $h > 0$ and $h \bmod z = 0$ are regarded as *renewal points* for A_0. At a renewal point, A_0 has completely decayed, A_1 has caught up with A_0 and can replace it, the static d-color problems have completed from which a new A_1 can be constituted, and new static problems can be initiated.

We now discuss how to avoid the expense of repeatedly updating each base in the arrangements A_0 and A_1. We do this by maintaining an implicit representation of each arrangement. An *extremal base of color j* of arrangement $R_{\bar{q},l}$ is a base $B_{\bar{i}}$, where $i_j = q_j - (d-1)(l-1)$ and $i_{j'} = q_{j'} + l - 1$ for $j' \neq j$. We denote this base as $B_{\bar{q},l,j}$. We also use the base $B_{\bar{q},l-1,j}$ and call this a *near-extremal base of color j*. For $g = 0$, 1 and $0 \leqq r < z$, let $a = g(z-r)$, and $b = z - r + g(2z-r)$. For each arrangement $A_g^{(h)}$ with $h = kz + r$, $0 \leqq r < z$ and $g = 0$, 1, except for when $g = 0$ and $r = z - 1$, we maintain for each color j, $B_{\bar{q},b,j}^{(h-a)}$ and its j-positive neighbors, and $B_{\bar{q},b-1,j}^{(h-a)}$ and its j-negative neighbors. For $d = 3$, this amounts to four bases near (and including) each of three extremal bases, for a total of twelve bases. For $d > 3$, there will be $2d$ bases near (and including) each of d extremal bases, for a total of $2d^2$ bases. We call the set of these bases the *extreme bases*. For each extreme base we maintain its update structure. Using the algorithm from the previous section, each of the $2d^2$ bases can be found in $T(n, d)$ time, and thus $P(n, d)$ is $O(d^2 T(n, d))$. (We provide a better bound on $P(n, d)$ in the proof of Theorem 5.) A symbolic representation of solutions to all problems for a matroid with $d = 3$ and $n = 24$ is given in Fig. 3. An arrangement centered at the base marked with an "X" and with radius $l = 4$ is shown in bold, with the extreme bases shown as the boldest. The extremal bases are the bases at the corners of the arrangement.

We now describe how the hth element cost change (involving an element of color j) is applied to the implicit representation of an arrangement $A_g^{(h-1)}$ to obtain the

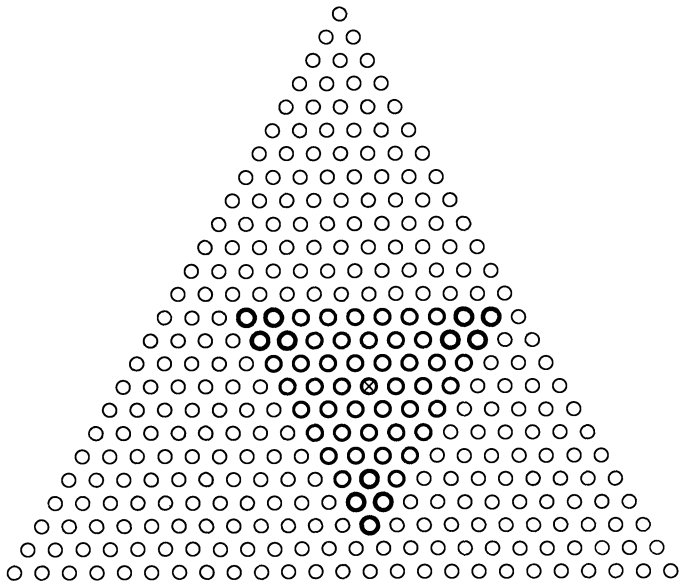


FIG. 3. *Symbolic representation of solutions to all problems in a matroid with three colors and rank* 24. *An arrangement centered at the base marked with an "X" and with radius* $l = 4$ *is in bold, and the extreme bases are the boldest of the bases.*

implicit representation of the updated arrangement $A_g^{(h)}$. We first update the monochromatic base of color j, and suitably modify the update structures of the extreme bases to reflect any change in this monochromatic base. We then compute new versions of a particular set of d extreme bases, one corresponding to each color, that we call the *cardinal bases* of the arrangement. We then compute a contracted matroid associated with the cardinal bases that is significantly smaller than the original matroid, but one that includes all the necessary elements. We finally extract and solve several static d-color problems in the contracted matroid; each static problem generates one extreme base in the implicit representation of the new arrangement.

The cardinal bases are chosen depending on the type of element cost change. For each color j, either the extremal base $B^{(h)}_{\bar{q},z-r,j}$ and its j-positive neighbors, or the near-extremal base $B^{(h)}_{\bar{q},z-r-1,j}$ and its j-negative neighbors are used to compute the cardinal bases. If the cost of a basic element of color j' increases, then the cardinal bases are generated using extremal bases and their j-positive neighbors. In this case the cardinal bases will be $B^{(h)}_{\bar{q},z-r,j'}$ and the (j',j)-neighbor of $B^{(h)}_{\bar{q},z-r,j}$ for all $j \neq j'$. We have previously discussed how $B^{(h)}_{\bar{q},z-r,j'}$ may be obtained from $B^{(h-1)}_{\bar{q},z-r,j'}$ and its j'-positive neighbors. When $j \neq j'$, let B denote $B_{\bar{q},z-r,j}$, and B' denote B's (j',j)-neighbor. Since the complete, positive, tight set consisting of B' and its j'-positive neighbors is identical to the complete, positive, tight set consisting of B and its j-positive neighbors, the sparse representation of the arrangement has sufficient information to generate the updated version of base B'. If the cost of a nonbasic element of color j' decreases, then the near-extremal bases and their j-negative neighbors are used. In this case the cardinal bases will be $B^{(h)}_{\bar{q},z-r-1,j'}$ and the (j,j')-neighbor of $B^{(h)}_{\bar{q},z-r-1,j}$ for all $j \neq j'$.

The details of how the cardinal bases and their associated contracted matroids are computed depends on the type of matroid. There are certain matroids (for instance, graphic matroids) for which update structures for bases in a contracted matroid can be maintained efficiently when elements are inserted into or deleted from its associated *contraction set* (the set of elements contracted). In such cases, we can save both space and time if we maintain a contracted matroid associated with the extremal bases of each arrangement. The contraction set consists of the union, over all colors j, of the j-colored elements in the extremal base of color j. Note that each element in the contraction set is common to all bases in the arrangement. In the contracted matroid associated with the cardinal bases, cardinal bases play the roles of extremal bases in the above definition. Once the cardinal bases are determined, the contracted matroid associated with the cardinal bases can be derived from the contracted matroid associated with the extremal bases by performing, for each color j, insertions and deletions corresponding to all elements of color j in the symmetric difference between the extremal and cardinal bases of color j. The time to compute these elements is charged to the cost of (subsequently) solving the static d-color problems. For matroids where efficient maintenance of contracted bases is not possible, we instead explicitly maintain the contraction set, and contract the elements each time the contracted matroid is required.

We discuss further the case in which the contracted matroid is explicitly maintained. If the update potentially involves a change in the contraction set, the contracted matroid associated with the extremal bases must be modified before computing the cardinal bases. Suppose an element e of color j in the contraction set increases in cost. If e remains in the monochromatic base of color j, then e should be deleted from the contraction set (yielding a contracted matroid of rank one greater), and the update structures for the extreme bases modified accordingly. If e is replaced by an element e' in the monochromatic base of color j, then e should be deleted from the

contraction set, and then replaced by e' in the contracted matroid, with the update structures for the extreme bases modified accordingly at each step. When an element e is removed from the contraction set, not only does e return to the contracted matroid, but also one element of each other color, which were deleted in various previous contractions. To facilitate identifying these other elements that should also return to the contracted matroid, we maintain for each color j' a base $\hat{B}_{j'}$. The base $\hat{B}_{j'}$ is the union of the contraction set with the elements of color j' in the contracted matroid. When element e of color j is removed from the contraction set, then for each $j' \neq j$ perform a *mincocirc*$(j', e, \hat{B}_{j'})$ to identify the element of color j' that should return to the contracted matroid.

Suppose an element f of color j decreases in cost. If f is in the monochromatic base of color j, but is in neither the contraction set nor the contracted matroid (such an element would have been deleted when some element in the contraction set was contracted), then there is some element e in the contraction set, which if deleted in the contraction set would cause f to be in the contracted matroid. The element e is found by performing a *maxcirc*$(j, f, \hat{B}_j)$, and is then deleted from the contraction set, with the update structures for the extreme bases modified accordingly. Finally, if f is not in the monochromatic base of color j and replaces an element f' in the monochromatic base of color j, then we handle f' as though it were an element that increases in cost and was replaced by f in the monochromatic base of color j. The cardinal bases can now be selected from the bases obtained by performing the appropriate update operations on the extreme bases, and the associated contracted matroid obtained as previously described.

Each extreme base in the new arrangement is then generated by extracting and solving a d-color problem in the contracted matroid associated with the cardinal bases. We also derive the contracted matroid associated with the extremal bases of the new arrangement from the contracted matroid associated with the cardinal bases of the old arrangement. As before, this is done by computing symmetric differences. The size of the contraction set associated with the extremal bases of an arrangement of radius l is $\sum_{j=1}^{d} (q_j - (d-1)(l-1)) = n - d(d-1)(l-1)$. Since the contracted elements are independent in the original matroid, the resulting contracted matroid will have rank $d(d-1)(l-1)$. We also note that since the original matroid has a monochromatic base of each color, so will the contracted matroid; thus the contracted matroid, like the original matroid, can be viewed as the union of d monochromatic bases. In what follows we will assume that, whenever appropriate, update structures are maintained for these smaller monochromatic bases in the contracted matroid.

To summarize, each update step h, where $h = kz + r$ and $0 \leqq r < z$, involves the following operations. The monochromatic minimum cost base is updated for the color of the element whose cost has changed. The arrangement $A_0^{(h-1)}$ is transformed to $A_0^{(h)}$ by applying the hth element cost change to it as follows. The cardinal bases are computed. Either the contracted matroid or the contraction set is updated, and in the latter case, the elements in the contraction set are contracted. Let the computation of the cardinal bases and the appropriate contraction structures be completed in $Q(n, d, z)$ time. A total of $2d^2+1$ static d-color problems of rank $n' = \Theta(d^2 z)$ are then extracted in the contracted matroid and each solved in $T(d^2 z, d)$ time, generating $B_{\bar{q}}^{(h)}$ and the extreme bases for the new arrangement $A_0^{(h)}$. For those matroids in which the update structures for the contracted matroid can be maintained efficiently under element insertion and deletion, the update structures for the extreme bases in $A_0^{(h-1)}$ are modified via swaps to yield update structures for these new bases, respectively. We then have the implicit representation for $A_0^{(h)}$ after the update step.

Finally, $A_1^{(h-1)}$ is transformed to $A_1^{(h)}$. The hth element cost change is added to the rear of the queue of element cost changes that we maintain for A_1. Two element cost changes from the front of the queue are then deleted and each is applied to $A_1^{(h-1)}$ in the same manner as the cost changes were applied to A_0, obtaining $A_1^{(h)}$.

THEOREM 5. *Let M be a matroid of rank n with m elements of d colors. Consider constrained minimum cost bases with respect to cost function $c_L(\cdot)$. The on-line update problem for such bases can be solved in $O(d^2U(m,n)+Q(n,d,z)+d^2T(d^2z,d)+dT(n,d)/z+d^2T(d^2z,d)/z)$ time and $O(dS(m,n)+d^3(d^2z+n))$ space.*

Proof. For each of the $O(d^2)$ extreme bases of each arrangement, an update operation will be performed. Then d cardinal bases in each arrangement are selected from these $O(d^2)$ updated bases. An updated arrangement $A_g^{(h)}$ is generated by solving $O(d^2)$ static d-color problems. This can be done by finding the new extreme bases of the arrangement for each color on a contracted matroid of rank $n'=O(d^2z)$. The space required for computing one of these bases is $O(d^3(d^2z))$, which is $O(d^5z)$ for computing all of them, since they are computed one at a time. The space required for storing the update structures for each of the extreme bases will be $O(d^3z)$, or $O(d^5z)$ overall. Solving the static d-color problems will take time $O(d^2T(d^2z,d))$. Thus each update step in A_0 or A_1 will take $O(d^2U(m,n)+Q(n,d,z)+d^2T(d^2z,d))$, time.

In addition, $O(d^2)$ static d-color problems of rank n must be solved over z updates in order to regenerate the arrangements. For each color j, compute the extremal bases of color j. Then contract the matroid to one of rank $n'=O(d^2z)$. The remaining extreme bases can be found in the contracted matroid. Thus the time spent per update step on solving these static d-color problems is $O((dT(n,d)+d^2T(d^2z,d))/z)$. The static d-color problems of rank n will be solved one at a time and thus require $O(d^3n)$ space overall. □

To illustrate the above technique, we describe the construction of update structures for graphic matroids and analyze their efficiency. The update structure for a minimum spanning tree uses dynamic tree data structures [ST] and two-dimensional topology trees [F]. The former allows us to perform the operations *maxcirc* and *swap* in time $O(\log n)$. The latter allows us to perform the operations *mincocirc* and *swap* in time $O(\sqrt{m})$. Thus for this update structure $U(m,n)=O(\sqrt{m})$. The space used by the structures is $O(m)$.

A contracted matroid is maintained in the form of a contracted graph. A topology tree [F] is used to maintain a heap of the edges incident on each vertex of the contracted graph. Each such vertex corresponds to a tree-structured connected component of contracted edges from the current constrained minimum spanning tree. Since topology trees of size d^2z support insert, delete, split, and merge operations in $O(\log(d^2z))$ time, updating the contracted graph can be implemented efficiently. Given the monochromatic bases, the time to solve a static d-color problem is $T(n,d)=O((d!)^2n\log n)$. We therefore have the following theorem.

THEOREM 6. *Let G be a graph with n vertices, and with m edges of d colors. Consider constrained minimum spanning trees with respect to cost function $c_L(\cdot)$. The on-line update problem for such spanning trees can be solved in $O(d^2\sqrt{m}+d^{5/2}(d!)^2\sqrt{n}\log n)$ time and $O(dm+d^3n)$ space.*

Proof. We have $U(m,n)=O(\sqrt{m})$, $T(n,d)=O((d!)^2n\log n)$ and $Q(n,d,z)$ will be $O(d^2U(d^2z,z)+d^2\log(d^2z))$, which is $O(d^3z^{1/2})$. Each update step in the arrangements will take $O(d^2\sqrt{m}+d^4(d!)^2z\log(d^2z))$ time. We must also replenish the second arrangement by solving a number of static problems of rank n, which will cost $O((d(d!)^2n\log n+d^4(d!)^2z\log(d^2z))/z)$ time per update. We choose $z=\Theta(n^{1/2}/d^{3/2})$. The space bound follows from our choice of z and $S(m,n)$. □

6. A recursive representation of arrangements. We can achieve better update times by using a more complex representation of arrangements. Consider the example in the last section involving graphic matroids. We can use a two-level approach for representing A_0 and A_1. Consider update step h, where $h = kz + r$ and $0 \leqq r < z$. Recall that $A_0^{(h)} = R_{\bar{q},z-h}^{(h)}$, where $R_{\bar{i},l}$ is the set of bases $\{B_{\bar{i}'} | i_j' \leqq i_j + l - 1, j = 1, 2, \cdots, d\}$. Arrangement A_0 was represented implicitly by the extreme bases, their associated data structures, and either the contracted matroid or the contraction set corresponding to the extremal bases. On an update step in A_0, d cardinal bases were determined, the contracted matroid (or contraction set) was updated using them, and $2d^2+1$ static problems of rank $n' = O(d^2 z)$ were solved to find the new extreme bases.

In our modified method, a base at each extreme is computed as before. However, instead of solving a number of static problems with respect to A_0 on each update step in A_0, we do the following. We maintain smaller arrangements A_{0j}, $j = 2, 3, \cdots, d+1$, centered near the extreme bases of A_0, and two smaller arrangements A_{00} and A_{01} centered at $B_{\bar{q}}$. We call these smaller arrangements *subarrangements.* Only when the subarrangements A_{0j}, $j = 2, 3, \cdots, d+1$, decay to single bases are a number of static problems solved with respect to A_0. A_{00} and A_{01} are maintained to be able to access $B_{\bar{q}}$ meanwhile.

Let l_{0j} be the radius of subarrangement A_{0j}, $j = 0, 1, \cdots, d+1$. For $j = 2, \cdots, d+1$, A_{0j} will be centered on $\bar{q}_j$, where $q_{jk} = q_k - (d-1)(l_0 - l_{0j})$ for $k = j$ and $q_{jk} = q_k + l_0 - l_{0j}$ for $k \neq j$. Let y be a parameter to be specified subsequently. At a renewal point for A_{0j}, $l_{0j} = y$ if $j = 0$, $l_{0j} = 3y$ if $j = 1$, and $l_{0j} = 2y$ if $j = 2, 3, \cdots, d+1$. Each subarrangement is represented implicitly by its $2d^2$ extreme bases, their associated data structures, and the contracted matroid (or contraction set). If the contracted matroid is maintained, the extreme bases are of rank $n' = \Theta(d^2 z)$; otherwise the bases are of rank n. After the A_{0j}, $j = 2, \cdots, d+1$, have decayed to radius 1, $2d^3$ static problems with $n' = d(d-1)(l - l_{0j})$ will be initiated to determine the extreme bases for the new A_{0j}, $j = 2, \cdots, d+1$.

At a renewal point for A_0, A_{00} will be up to date with respect to A_0, A_{0j}, $j = 1, 2, \cdots, d+1$, will be out of date with respect to A_{00} (and therefore A_0) by y element cost changes. Times at which $h \bmod z > 0$ and $h \bmod y = 0$ are regarded as renewal points for A_{0j}, $j = 0, 1, \cdots, d+1$. At a renewal point for A_{00}, A_{00} has completely decayed, A_{01} has caught up with A_{00} and can replace it, arrangements A_{0j}, $j = 2, \cdots, d+1$, have caught up with A_{00} but have decayed to single bases, the $(d+1)2d^2$ static problems have been completed which yield the extreme bases for the new arrangements A_{0j}, $j = 1, \cdots, d+1$, and a new set of static problems can be initiated using the single bases from the previous A_{0j}, $j = 2, \cdots, d+1$. As before, two update steps in an out-of-date arrangement will be performed for every update step in A_0. We will assume that $z \bmod y = 0$, so that A_{0j}, $j = 1, l, \cdots, d+1$, will catch up with A_0 precisely when A_0 reaches its next renewal point. Arrangement A_1 is represented in a similar fashion. Subarrangements A_{1j}, $j = 1, \cdots, d+1$, will initially be out of date with respect to A_1 by y element cost changes. Since A_1 is itself out of date with respect to A_0, four update steps will be performed in each of A_{1j}, $j = 1, \cdots, d+1$, for every update step in A_0.

We discuss how to perform an update in A_0. The update for A_1 is similar. For each of the extreme bases of A_0, an update operation is performed. Then the d cardinal bases are selected from those $O(d^2)$ updated bases. The contracted matroid (or contraction set) corresponding to the cardinal bases is computed. In addition, for all extreme bases of A_{0j} that are not extreme bases of A_0, an update operation is performed. For each group of d bases in this set, a cardinal base is computed. Then, for each

$j = 0, 1, \cdots, d+1$, the contracted matroid (or contraction set) corresponding to the cardinal bases of A_{0j} is computed. A total of $2d^2+1$ static d-color problems of rank $n' = \Theta(d^2 y)$ are solved for each of the $d+1$ subarrangements of A_0. From the extreme bases of the A_{0j} that correspond to extreme bases of A_0, swaps that transform the old extreme bases of A_0 into the new extreme bases can be inferred. The new contracted matroids (or contraction sets) for A_0 and its subarrangements can then be determined.

In addition, the following static problems must be solved over a sequence of updates. To generate the extreme bases for A_1, $O(d^2)$ static d-color problems of rank n must be solved over z updates. To generate the extreme bases for A_{gj}, $g = 0$, 1 and $j = 1, 2, \cdots, d+1$, $O(d^3)$ static d-color problems of rank $n' = \Theta(d^2 z)$ must be solved over y updates.

THEOREM 7. *Let G be a graph with n vertices, and with m edges of d colors. Consider constrained minimum spanning trees with respect to cost function $c_L(\cdot)$. The on-line update problem for such spanning trees can be solved in $O(d^2\sqrt{m} + d^{11/3}(d!)^2 n^{1/3} \log n)$ time and $O(dm + d^3 n)$ space.*

Proof. For each of the $O(d^2)$ extreme bases of arrangements A_0 and A_1, an update operation will be performed. Then d cardinal bases in each arrangement are selected from these $O(d^2)$ updated bases. The time required is $O(d^2 U(m, n))$. For each of the $O(d^3)$ extreme bases of subarrangements A_{0j} and A_{1j}, an update operation will be performed. Then d new extreme bases in each subarrangement are selected from its $O(d^2)$ updated bases. The total time required is $O(d^3 U(d^2 z, n))$. An updated arrangement A_{kj} is generated by solving $O(d^2)$ static d-color problems. This can be done by finding the extreme bases for each color on a contracted matroid of rank $n' = O(d^2 y)$. Thus solving the static d-color problems will take time $O(d^3 T(d^2 y, d))$. Thus each update step in the arrangements and subarrangements will take $O(d^2 U(m, n) + Q(n, d, z) + d^3 U(d^2 z, n) + d^3 T(d^2 y, d))$ time.

In addition, $O(d^2)$ static d-color problems of rank n must be solved over z updates. As in the proof of Theorem 5, this will take $O((dT(n, d) + d^2 T(d^2 z, d))/z)$ time per update step. Also, $O(d^3)$ static d-color problems of rank $\Theta(d^2 z)$ must be solved over y updates. The time spent per update step on solving these static d-color problems will be $O((d^3 T(d^2 z, d))/y)$. The time spent handling each element cost change is $O(d^2\sqrt{m} + d(d!)^2((n \log n)/z + d^4(z \log z)/y + d^4 y \log y))$. Choosing $z = \Theta(n^{2/3}/d^{8/3})$ and $y = \Theta(n^{1/3}/d^{4/3})$ yields the time claimed by the theorem.

For the space, proceeding in a fashion similar to that in the proof of Theorem 5, we obtain a bound of $O(dS(m, n) + d^3(n + d^3 z))$, which is $O(dm + d^3 n)$ for our choice of z and $S(m, n)$. □

For fixed d, the time for the above approach is limited by the $O(\sqrt{m})$ time to update a minimum spanning base in an uncolored graph. If the graph is planar however, then the update time in an uncolored graph has been shown to be $O(\log n)$ in [GS], and hence is not a limiting factor. We thus extend recursively the implicit representation of arrangements. The representations will be of two types, centered and uncentered. Let $a(d)$ be a value depending on d, which we shall specify subsequently. An arrangement, centered or uncentered, of radius at most $a(d)$, is the set of extreme bases, their associated data structures, and the contracted matroid (or contraction set) corresponding to the extremal bases. Let $f(\cdot)$ be a function to be defined subsequently. For an arrangement A_λ of radius l_λ initially equal to $z > a(d)$, *a centered representation* consists of the above items, in addition to the following:

(1) A centered representation of a subarrangement $A_{\lambda 0}$, which is centered on the same position as A_λ, with radius $l_{\lambda 0}$ initially equal to $f(z)$, and which is up to date with respect to A_λ.

(2) A centered representation of a subarrangement $A_{\lambda 1}$, which is centered on the same position as A_λ, with radius $l_{\lambda 1}$ initially equal to $3f(z)$, and which is out of date with respect to A_λ by $l_{\lambda 0}$ element cost changes.

(3) Uncentered representations of subarrangements $A_{\lambda j}$, $j=2, \cdots, d+1$, which are positioned at the extremes of A_λ, with radius $l_{\lambda j}$ initially equal to $2f(z)$, and which are out of date with respect to A_λ by $l_{\lambda 0}$ element cost changes.

(4) $2d^2$ static problems that have just been initiated. Of these, $2d$ will be of rank $n'=\Theta(d^2 z)$, and the remainder of rank $\Theta(d^2 f(z))$.

An *uncentered representation* consists of all items in a centered representation except items (1) and (2).

Let $f^{(0)}(x)=x$ and $f^{(i)}(x)=f(f^{(i-1)}(x))$, for $i>0$. Then we choose the function $f(\cdot)$ such that $f^{(i+1)}(n) \bmod f^{(i)}(n)=0$ for $i>0$. This can be done easily by forcing $f(\cdot)$ to be a power of 2. This choice of $f(\cdot)$ ensures that each $(i+1)$st level arrangement will have caught up with the appropriate ith level arrangement at an ith level renewal point.

Let $T_C(z)$ and $T_U(z)$ be the update times for centered and uncentered arrangements of radius z, respectively. The update times are described by the following recurrences:

$$T_U(z)=cd^3(d!)^2(z \log z)/f(z)+2dT_U(2f(z)),$$

$$T_C(z)=cd^3(d!)^2(z \log z)/f(z)+2dT_U(2f(z))+2T_C(3f(z))+T_C(f(z))$$

where c is a constant. The first term in each recurrence represents the time spent per update step on solving the static problems of rank $\Theta(d^2 z)$ and updating the data structures. The remaining terms represent the time for recursively updating subarrangements of radius $\Theta(f(z))$, and reflect the fact that two update steps are required for out-of-date subarrangements for each update step in the primary arrangement.

THEOREM 8. *Let G be a planar graph with n vertices, and edges of d colors. Consider constrained minimum spanning trees with respect to cost function $c_L(\cdot)$. The on-line update problem for such spanning trees can be solved in $O(d^3(d!)^2(\log d)^{-1/2} 2^{(2\log(2d)\log n)^{1/2}}(\log n)^{3/2})$ time and $O(d^3 n)$ space.*

Proof. We have $U(m, n)=O(\log n)$, $P(n)=O(n \log n)$ and $Q=0$. If we choose $f(x)=\Theta(x/2^{(2\log(2d)\log x)^{1/2}})$ and observe that

$$\sqrt{\log f(x)}=\sqrt{\log x-\sqrt{2\log(2d)\log x}}<\sqrt{\log x}-\sqrt{(\log(2d))/2},$$

then both $T_U(n)$ and $T_C(n)$ are $O(d^3(d!)^2(\log d)^{-1/2}2^{(2\log(2d)\log n)^{1/2}}(\log n)^{3/2})$, provided $a(d)$ is small enough, so that the basis of the recurrences satisfies these bounds.

For the space, the recursive representation has at most $(d+2)^i$ subarrangements, each using data structures of size $\Theta(f^{(i)}(n))$ at level i. With $d+2 \leqq 2d \leqq 2^{(2\log(2d)\log n)^{1/2}}$, the sizes of these structures sum to $O(n)$ over all levels.

Solving for n in the above inequality suggests the choice of $a(d)=\sqrt{2d}$. Since arrangements of radius at most $a(d)$ are represented explicitly, the space for representing arrangements is $O(n)$, aside from the space for the static problems being solved. At level i, there are $\Theta(d^{1+i})$ static d-color problems of rank $\Theta(f^{(i)}(n))$ and $\Theta(d^{2+i})$ static d-color problems of rank $\Theta(f^{(i+1)}(n))$ being solved. These static problems are solved one at a time, and the space requirement for computing and recording their solutions sums to $O(d^3 n)$ over all levels.

If the general matroid intersection algorithm is used for updating arrangements of radius at most $a(d)$ in the centered and uncentered representations, then the basis in the recurrences is polynomial in d. Thus the basis satisfies the claimed bounds on $T_U(n)$ and $T_C(n)$. □

7. An application. The techniques of § 4 can be used to solve the minimum spanning tree problem when d vertices have degree constraints. Assume that the vertices with degree constraints are indexed $v_1, v_2, \cdots, v_d$. Label each edge incident on two constrained vertices with color 0. Label each edge incident on exactly one constrained vertex v_i with color i. Label each edge incident on two unconstrained vertices with color $d+1$.

Since there are d constrained vertices, there are at most $d(d-1)/2$ edges of color 0. In turn we consider every subset of edges of color 0 that is a forest, such that the degree of each v_i in the forest does not exceed its degree requirement r_i. We generate a candidate solution for each such forest. The idea is to include all the forest edges in the solution and then choose remaining edges so as to satisfy the degree constraints in a minimum cost fashion. The minimum cost solution over all such forests is then the minimum spanning tree satisfying the degree constraints.

For each forest, we generate a reduced graph as follows. Make a copy of the graph, and initialize r_i' to be r_i for $i=1, 2, \cdots, d$. Delete from the graph all edges of color 0 that are not in the forest. For each edge (v_i, v_j) in the forest, decrease by 1 the degree requirements r_i' and r_j'. Then contract the remaining edges of color 0 in the graph. To get the candidate solution corresponding to this forest solve a $(d+1)$-color static problem on the reduced graph, where r_i' edges of color i are desired, for $i=1, 2, \cdots, d$, and the remaining edges are of color $d+1$.

THEOREM 9. *The time to solve a minimum spanning tree problem with degree constraints on d of the vertices is* $O(T_0(m, n)+((d+1)!)^2 d^{d-1} T(n, 2))$, *and the space is* $O(d^3 n)$.

Proof. For each forest, the set of edges of any color $j>0$ in the corresponding $(d+1)$-color problem is the same. The only monochromatic minimum spanning tree that cannot be inferred by definition is the one of color $d+1$. Thus the first term reflects the time to solve a minimum spanning tree problem on edges of color $d+1$.

We next derive a bound on the number of undirected labeled forests, and thus the number of $(d+1)$-color problems that must be solved. We first count directed labeled graphs in which each vertex has outdegree 1, with self-loops allowed. This quantity is an upper bound on the number of directed labeled forests, and is a loose bound since it allows directed cycles other than self-loops. The edge directed out of each vertex can be any one of d vertices. Hence at most d^d such graphs are possible. To obtain a slightly tighter bound for undirected labeled forests, observe that at least one vertex in a directed labeled forest is a root. In counting undirected labeled forests, it makes no difference which vertex this is. So in generating the above directed labeled graphs we arbitrarily choose vertex v_1 to be a root. Thus we choose from among d possible edges out of each of the remaining $d-1$ vertices, which means at most d^{d-1} undirected labeled forests are possible.

The space required is dominated by the space needed to find one $(d+1)$-color spanning tree. □

Acknowledgment. We thank the referee for a careful reading of the paper and for many helpful suggestions.

REFERENCES

[B] O. BORUVKA, *O jistem problemu minimalnim*, Praca Moravske Prirodovedecke Spolecnosti, 3 (1926), pp. 37-58.

[BCG1] C. BREZOVEC, G. CORNUEJOLS, AND F. GLOVER, *Two algorithms for weighted matroid intersection*, Math. Programming, 36 (1986), pp. 39–53.

[BCG2] ———, *A matroid algorithm and its application to the efficient solution of two optimization problems on graphs*, in Math. Prog., Series B, to appear.

[F] G. N. FREDERICKSON, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[FS] G. N. FREDERICKSON AND M. A. SRINIVAS, *On-line updating of solutions to a class of matroid intersection problems*, Inform. and Comput., 74 (1987), pp. 113–139.

[FT] M. FREDMAN AND R. E. TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 596–615.

[GGST] H. N. GABOW, Z. GALIL, T. H. SPENCER, AND R. E. TARJAN, *Efficient algorithms for finding minimum spanning trees in directed and undirected graphs*, Combinatorica, 6 (1986), pp. 109–122.

[GS] H. N. GABOW AND M. STALLMANN, *Efficient algorithms for graphic matroid intersection and parity*, in Proc. International Conference on Automata, Languages, and Programming, Napflion, Greece, Lecture Notes in Computer Science 194, Springer-Verlag, Berlin, New York, 1985, pp. 210–220.

[GT] H. N. GABOW AND R. E. TARJAN, *Efficient algorithms for a family of matroid intersection problems*, J. Algorithms, 5 (1984), pp. 80–131.

[GJ] 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.

[G] D. GUSFIELD, *Matroid optimization with the interleaving of two ordered sets*, Discrete Appl. Math., 8 (1984), pp. 41–50.

[L1] E. L. LAWLER, *Combinatorial Optimization*: *Networks and Matroids*, Holt, Reinhart and Winston, New York, 1976.

[L2] ———, *Matroid intersection algorithms*, Math. Programming, 9 (1975), pp. 31–56.

[ST] D. D. SLEATOR AND R. E. TARJAN, *A data structure for dynamic trees*, J. Comput. System Sci., 26 (1983), pp. 362–391.

[W] D. J. A. WELSH, *Matroid Theory*, Academic Press, New York, 1976.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 139-151, February 1989

009

OPTIMAL BIN PACKING WITH ITEMS OF RANDOM SIZES II*

WANSOO T. RHEE† AND MICHEL TALAGRAND‡

Abstract. Consider a probability measure μ on $[0, 1]$ and independent random variables $X_1, \cdots, X_n$, distributed according to μ. Let $Q_n(X_1, \cdots, X_n)$ be the minimum number of unit-size bins needed to pack items of size $X_1, \cdots, X_n$. In this paper it is proven that $|E(Q_n(X_1, \cdots, X_n))/n - c(\mu)| \leqq K((\log n)/n)^{1/2}$, where K is a universal constant and $c(\mu)$ depends on μ, $c(\mu) = \lim_{n\to\infty} E(Q_n)/n$.

Key words. stochastic bin packing, asymptotic occupancy, size distribution, compactness, weak convergence

AMS(MOS) subject classifications. primary 90B05; secondary 05B99, 46A50, 28A33

1. Introduction. The bin-packing problem requires finding the minimum number of unit size bins needed to pack a given collection of items with sizes $X_1, \cdots, X_n$ in $[0, 1]$. This problem has many applications and has been shown to be NP-complete [3], [4]. Recently, a number of authors have been interested in analyzing stochastic models for bin packing. Coffman, Garey, and Johnson [1] gave an up-to-date survey of results on this question. Most authors have analyzed approximation algorithms under a model of elements drawn independently from the uniform distribution on $[0, 1]$. The main results of this paper are deterministic. They are however connected with a more general stochastic model. Consider a probability measure μ on $[0, 1]$. (No regularity assumption is made on μ.) Consider n items, and the sizes of these items $X_1, \cdots, X_n$ which are independent identically distributed random variables distributed according to μ. (For simplicity, we denote by X_k both item names and item sizes.) We let $Q_n = Q_n(X_1, \cdots, X_n)$ denote the minimum number of unit-size bins needed to pack $X_1, \cdots, X_n$. It is well known that Q_n is a subadditive process (see [5]). So we have $\lim_n Q_n/n = c(\mu)$ almost surely for some constant $c(\mu)$ depending on μ, and $E(Q_n)/n \geqq c(\mu)$ for each n. In [12], the authors have shown that Q_n is very concentrated around its expectation. More precisely, for all $t > 0$,

$$\Pr(|Q_n - E(Q_n)| \geqq t) \leqq 2\exp(-t^2/2n).$$

This however gives no information on the value of $E(Q_n)$. Among other results, we will find a sharp bound for the difference $E(Q_n) - nc(\mu)$ (see Theorem D below). In [11], Rhee has given a complete description of the distributions μ for which $c(\mu) = E(X_1)$. In order to state the full strength of our results, we need first to state an extension of Rhee's result. For $k \geqq 1$, let

$$R_k = \left\{(x_1, \cdots, x_k) \in R^k : 0 \leqq x_1 \leqq \cdots \leqq x_k, \sum_{i\leqq k} x_i \leqq 1\right\}.$$

For $x \in [0, 1]$, we denote by δ_x the probability measure concentrated at x, that is $\delta_x(G) = 1$ if x belongs to G and $\delta_x(G) = 0$ otherwise. For a compact metric space S, we denote by $M_1(S)$ the set of probability measures on S.

For $\nu \in M_1(R_k)$, we define $\mathcal{P}(\nu) \in M_1([0, 1])$ in the following way: for each Borel set G of $[0, 1]$,

$$\mathcal{P}(\nu)(G) = (1/k)\int_{R_k} \sum_{i\leqq k} \delta_{x_i}(G)\, d\nu(x_1, \cdots, x_k).$$

* Received by the editors March 5, 1986; accepted for publication (in revised form) December 30, 1987.
† Academic Faculty of Management Sciences, Ohio State University, Columbus, Ohio, 43210-1399.
‡ Equipe d'Analyse-Tour 46, Université de Paris VI, Paris, France.

In short, this formula (and similar formulae) will be written

$$\mathcal{P}(\nu) = (1/k) \int_{R_k} \sum_{i \leqq k} \delta_{x_i} \, d\nu(x_1, \cdots, x_k).$$

The simple, but essential, idea is that if ν is the distribution of items $X_1, \cdots, X_k$, these items can always be packed in one bin and $\mathcal{P}(\nu)$ is the distribution of X_i, where $i \in \{1, \cdots, k\}$ is uniformly distributed independently of $X_1, \cdots, X_k$.

THEOREM A. *For μ in $M_1([0,1])$, there is a nonnegative sequence $(\alpha_k)_{k \geqq 0}$ with $\sum_{k \geqq 0} \alpha_k = 1$, and for each $k \geqq 1$, a ν_k in $M_1(R_k)$ such that*

$$\mu = \alpha_0 \delta_0 + \sum_{k \geqq 1} \alpha_k \mathcal{P}(\nu_k)$$

and that

$$c(\mu) \geqq \sum_{k \geqq 1} \alpha_k / k.$$

Here is the main result.

THEOREM B. *Consider a nonnegative sequence $(\alpha_k)_{k \geqq 0}$ with $\sum_{k \geqq 0} \alpha_k = 1$, and a sequence $\nu_k \in M_1(R_k)$. Consider the measure*

$$\mu = \alpha_0 \delta_0 + \sum_{k \geqq 1} \alpha_k \mathcal{P}(\nu_k). \tag{1.1}$$

For $l \geqq 1$, let $\beta_l = \sum_{k \geqq l} \alpha_k / k$. Consider a sequence $s_1, \cdots, s_n$ of items. Let

$$W = W(s_1, \cdots, s_n) = \operatorname*{Sup}_{0 \leqq t \leqq 1} \{\operatorname{card}\{i \leqq n;\ s_i \geqq t\} - n\mu([t,1])\}, \quad \text{and} \tag{1.2}$$

$$W+ = \operatorname{Sup}(W, 0).$$

Then

$$Q_n(s_1, \cdots, s_n) \leqq W^+ + n \sum_{k \geqq 1} (\alpha_k / k) + Kn^{1/2}\left(1 + \sum_{i \leqq n^{1/2}} (\beta_i / i)^{1/2}\right) \tag{1.3}$$

where K is a universal constant (independent of μ, n, and $s_1, \cdots, s_n$).

In particular,

$$Q_n(s_1, \cdots, s_n) \leqq W^+ + n \sum_{k \geqq 1} (\alpha_k / k) + K(n(1 + \log n))^{1/2}. \tag{1.3'}$$

Comments. (1) The packing procedure by which we obtain (1.3) is explicit and fairly simple. However it requires complete knowledge of the decomposition (1.1). We have not estimated its time complexity, since it is not clear what are realistic assumptions about the effort needed to compute the quantities related to (1.1) that we need.

(2) Our proof shows that (1.3) actually holds for $K = 8$, but we have used only simple estimates and have made no special efforts to find a small value for K; this would not be appropriate since we do not know what is the best possible order of the error term.

The following theorems are two consequences of Theorem B.

THEOREM C. *Let $s_1, \cdots, s_n \in [0,1]$. Let $\nu = \sum_{i \leqq n} \delta_{s_i} / n$. Then*

$$nc(\nu) \leqq Q_n(s_1, \cdots, s_n) \leqq nc(\nu) + K(n(1 + \log n))^{1/2}. \tag{1.4}$$

THEOREM D. *There is a universal constant K such that for any distribution μ,*

$$nc(\mu) \leqq E(Q_n(X_1, \cdots, X_n)) \leqq nc(\mu) + K(n(1 + \log n))^{1/2}. \tag{1.5}$$

If, moreover, the series $\sum_{i\geqq 1}(\beta_i/i)^{1/2}$ *converges (where* $\beta_i=\sum_{k\geqq i}\alpha_k/k$ *and* (α_k) *is as in Theorem* A), *we have*

$$nc(\mu)\leqq E(Q_n)\leqq nc(\mu)+K(\mu)n^{1/2} \tag{1.6}$$

where $K(\mu)$ *depends on* μ *only.*

When μ is the uniform distribution on $[0,1]$, (1.6) is due to Knödel [6] and Lueker [7]. In this context, it is of interest to note the following, related to a theorem of Lueker.

THEOREM E. *Let* μ *be a probability measure on* $[0,1]$. *Let* A *be a measurable subset of* $[0,1]$ *with* $\mu(A)=1$. *Let* $0<\theta\leqq 1$. *The following are equivalent*:

(a) $c(\mu)>\theta$.

(b) *There exists a nonnegative continuous function* f *on* $[0,1]$ *such that* $\int f\,d\mu>\theta$ *and* $f(0)=0$, *and that* f *satisfies the following condition*:

(*) *For any* $k\geqq 1$, *for each* $x_1,\cdots,x_k\in[0,1]$, $\sum_{i\leqq k}x_i\leqq 1\Rightarrow\sum_{i\leqq k}f(x_i)\leqq 1$.

(c) *There exists a Borel function* f *on* $[0,1]$ *such that* $\int f\,d\mu>\theta$ *and* $f(0)\leqq 0$, *and that* f *satisfies the following condition*:

(*) *For any* $k\geqq 1$, *for each* $x_1,\cdots,x_k\in A$, $\sum_{i\leqq k}x_i\leqq 1\Rightarrow\sum_{i\leqq k}f(x_i)\leqq 1$.

The paper is organized as follows: Section 2 contains all the proofs, except that of Theorem B; Section 3 prepares the proof of Theorem B, which is done in § 4.

2. Some simple facts. Let S be a compact metric space. We denote by $C(S)$ the class of continuous functions on S. Denote by $M_+(S)$ the set of all positive measures on S and by $M(S)$ the set of all bounded variation measures on S, so $M_1(S)\subset M_+(S)\subset M(S)$. The weak topology on $M(S)$ is the coarsest topology such that for every $f\in C(S)$, the map $\mu\to\int_S f\,d\mu$ is continuous. Provided with the weak topology, $M_1(S)$ is compact metric. Let $0\leqq\theta\leqq 1$. We define the set C_θ as the set of μ in $M_1([0,1])$, for which there exists a sequence $(\alpha_k)_{k\geqq 0}$ such that $\sum_{k\geqq 0}\alpha_k=1$ and $\sum_{k\geqq 1}\alpha_k/k\leqq\theta$, and a sequence $\nu_k\in R_k$, for $k\geqq 1$, such that

$$\mu=\alpha_0\delta_0+\sum_{k\geqq 1}\alpha_k\mathscr{P}(\nu_k).$$

PROPOSITION 1. C_θ *is convex. If a sequence* (θ_n) *converges to* θ *and* $\mu_n\in C_{\theta_n}$ *converges weakly to* μ, *then* $\mu\in C_\theta$. (*In particular* C_θ *is weakly closed.*)

Proof. The first assertion is obvious. The proof of the second needs only obvious modifications from the proof of Lemma 2 in [11], to which we refer the reader.

We now prove Theorem A. The proof is very similar to part of the proof of the main theorem of [11]. A well-known consequence of the Law of Large Numbers is that $(1/n)\sum_{i\leqq n}\delta_{X_i}\to\mu$ almost surely in the weak topology. Also, $(1/n)Q_n(X_1,\cdots,X_n)\to c(\mu)$ almost surely, so we can find a sequence (x_i) such that $\mu_n=(1/n)\sum_{i\leqq n}\delta_{x_i}$ converges weakly to μ and k_n/n converges to $c(\mu)$, where $k_n=Q_n(x_1,\cdots,x_n)$. (We denote item sizes and item names by lowercase letters to emphasize that they are not random.) We fix n and we consider a packing of items of size $x_1,\cdots,x_n$ in k_n bins. In the jth bin, $j\leqq k_n$, we have p_j items of size $y_{j,1},\cdots,y_{j,p_j}$ with $y_{j,1}\leqq\cdots\leqq y_{j,p_j}$. Let $\lambda_j=(1/p_j)(\sum_{l\leqq p_j}\delta_{y_{j,l}})$. We have

$$\mu_n=\sum_{j\leqq k_n}(p_j/n)\lambda_j. \tag{2.1}$$

We have $\sum_{j\leqq k_n}(p_j/n)/p_j=k_n/n$. It follows from (2.1) that $\mu_n\in C_{k_n/n}$, so $\mu\in C_{c(\mu)}$ by Proposition 1. This proves Theorem A.

We postpone the proof of (1.3) to §§ 3 and 4, and we deduce all the other results from it. Since $\beta_i = \sum_{k \geqq i} \alpha_k / k$, we have $\sum_{i \geqq 1} \beta_i = \sum_{k \geqq 1} \alpha_k \leqq 1$. From the Cauchy-Schwartz inequality,

$$\sum_{i \leqq n^{1/2}} (\beta_i / i)^{1/2} \leqq \left(\sum_{i \leqq n^{1/2}} \beta_i \right)^{1/2} \left(\sum_{i \leqq n^{1/2}} 1/i \right)^{1/2}.$$

Now $\sum_{i \leqq n^{1/2}} 1/i \leqq K_1(1+\log n)$ for some constant K_1. So, (1.3) implies (1.3′) (maybe with a different constant K).

Proof of Theorem C. We apply Theorem A to ν in order to get a decomposition of ν as in (1.1), where $\sum_{k \geqq 1} \alpha_k / k \leqq c(\nu)$. The right-hand side inequality of (1.4) then follows from (1.3′), since $W^+ = 0$. To prove the left-hand side inequality, if we draw items $X_1, \cdots, X_N$ distributed according to ν, for $i \leqq n$, let N_i be the number of items of size s_i and let $a_N = \operatorname{Sup}_{i \leqq n} N_i$. We partition the items into a_N collections, where each collection contains at most one item of each size and pack each collection optimally. We need at most $a_N Q_n(s_1, \cdots, s_n)$ bins. The Law of Large Numbers shows that $\lim_{N \to \infty} E(a_N)/N = 1/n$ and this implies the result, since

$$c(\nu) = \lim_{N \to \infty} E(Q_N(X_1, \cdots, X_N))/N$$

$$\leqq \lim_{N \to \infty} E(a_N Q_n(s_1, \cdots, s_n))/N = (1/n) Q_n(s_1, \cdots, s_n).$$

Proof of Theorem D. The left-hand side of (1.5) follows from subadditivity. For the right-hand side, if

$$W(X_1, \cdots, X_n) = \operatorname*{Sup}_{0 \leqq t \leqq 1} \{\operatorname{card}\{i \leqq n, X_i \geqq t\} - n\mu([t, 1])\},$$

the Kolmogorov-Smirnov theorem implies that $E(W^+) \leqq K n^{1/2}$ for some universal constant K. So Theorem D follows from Theorem B.

Remark. The argument also shows that for $\mu \in C_\theta$, we have $c(\mu) \leqq \theta$.

Proof of Theorem E. (a⇒b). Consider the set C of all measures γ in $M_+([0, 1])$ that can be written as a sum

$$\gamma = \alpha_0 \delta_0 + \sum_{k \geqq 1} \alpha_k \mathcal{P}(\nu_k)$$

where $\nu_k \in M_1(R_k)$, $\alpha_i \geqq 0$ for each $i \geqq 0$, and $\sum_{k \geqq 1} \alpha_k / k \leqq \theta$. It is clear that C is convex. Moreover, for γ in C, the probability $\gamma / \gamma([0, 1])$ belongs to $C_{\theta/\gamma([0,1])}$. We show now that μ does not belong to the weak closure of C. Otherwise, there is a sequence (γ_n) in C that converges weakly to μ. In particular, the sequence $a_n = 1/\gamma_n([0, 1])$ converges to 1, so the sequence $\gamma_n' = a_n \gamma_n$ converges weakly to μ. Moreover $\gamma_n' \in C_{a_n \theta}$, so Proposition 1 shows that $\mu \in C_\theta$. The remark before the proof shows that this is a contradiction.

We can now use the geometric form of the Hahn-Banach theorem (in the space $M(S)$ of all bounded variation measures provided with the weak topology) to get a continuous function g on $[0, 1]$ such that $\int g \, d\mu > \theta$ but $\int g \, d\nu \leqq \theta$ whenever $\nu \in C$. Since $\alpha \delta_0 \in C$ for all $\alpha \geqq 0$, we have $g(0) \leqq 0$. Now let $x_1, \cdots, x_k$ be in $[0, 1]$ with $\sum_{i \leqq k} x_i \leqq 1$. Since $\nu = \theta \sum_{i \leqq k} \delta_{x_i}$ belongs to C, we have $\int g \, d\nu = \theta \sum_{i \leqq k} g(x_i) \leqq \theta$. Now define $f = \operatorname{Max}(0, g)$, so f is continuous, and $\int f \, d\mu > \theta$. Given $x_1, \cdots, x_k \leqq 1$, let $J = \{i \leqq k; g(x_i) > 0\}$. Since $\sum_{i \in J} x_i \leqq 1$, we have $\sum_{i \in J} g(x_i) = \sum_{i \leqq k} f(x_i) \leqq 1$. This completes the proof.

(b⇒c). The proof is obvious.

(c⇒a). Suppose, if possible, that $c(\mu) \leqq \theta$. By Theorem A, we have $\mu \in C_\theta$. So there is a sequence α_k with $\sum_{k\geqq 0} \alpha_k \leqq 1$, $\sum \alpha_k/k \leqq \theta$ and $\nu_k \in M_1(R_k)$ such that $\mu = \alpha_0\delta_0 + \sum_{k\geqq 1} (\alpha_k/k)\mathcal{P}(\nu_k)$. In particular, for each Borel set G of $[0, 1]$, we have

$$\mu(G) = \alpha_0\delta_0(G) + \sum_{k\geqq 1} (\alpha_k/k) \int_{R_k} \sum_{i\leqq k} \delta_{x_i}(G)\, d\nu_k(x_1, \cdots, x_k).$$

Using this equality for $G = [0, 1]\backslash A$, we see that for each k such that $\alpha_k > 0$, each $i \leqq k$, we have $\delta_{x_i}(G) = 0$ for ν_k almost surely. In other words, $\delta_{x_i}(A) = 1$ ν_k almost surely. So ν_k is concentrated on

$$P_k = \{(x_1, \cdots, x_k);\ 0 \leqq x_1 \leqq \cdots \leqq x_k,\ x_1, \cdots, x_k \in A\}.$$

Now, we have

$$\int f d(\mathcal{P}(\nu_k)) = (1/k) \int_{R_k} \sum_{i\leqq k} f(x_i)\, d\nu_k(x_1, \cdots, x_k)$$

$$= (1/k) \int_{P_k} \sum_{i\leqq k} f(x_i)\, d\nu_k(x_1, \cdots, x_k).$$

It follows that

$$\int f d(\mathcal{P}(\nu_k)) \leqq \operatorname{Sup}\left\{(1/k) \sum_{i\leqq k} f(x_i);\ (x_i)_{i\leqq k} \in P_k\right\} \leqq 1.$$

We then have

$$\int f\, d\mu \leqq \alpha_0 f(0) + \sum_{k\geqq 1} \alpha_k/k \leqq \theta.$$

This contradiction concludes the proof.

3. Preparation.

LEMMA 1. (Well known; used and proved in [11] in the case $q = \operatorname{card} U$.) *Let $q \geqq 0$ and U be an index set. Consider two families $(a_i)_{i\leqq q}$, $(b_u)_{u\in U}$ and $0 \leqq a_i, b_u \leqq 1$. Let*

$$h = \sup_{0\leqq t\leqq 1} \{\operatorname{card}\{1 \leqq i \leqq q;\ a_i \geqq t\} - \operatorname{card}\{u \in U;\ b_u \geqq t\}\}. \tag{3.1}$$

Then there is a subset $H \subseteq \{1, \cdots, q\}$ with $\operatorname{card} H \geqq q - h$ and ϕ, a one-to-one map from H to U with $a_i \leqq b_{\phi(i)}$ for $i \in H$.

Note. We index the b's by an abstract index set U because we find it is more convenient to prove the next lemma.

LEMMA 2. *Let p, q, $n > 0$. Consider a family $(a_i)_{i\leqq q}$, $0 \leqq a_i \leqq 1$ and for $1 \leqq j \leqq p$, left-continuous nonincreasing functions $(f_j)_{j\leqq p}$ on $[0, 1]$. Set*

$$h = \sup_{0\leqq t\leqq 1} \left\{\operatorname{card}\{1 \leqq i \leqq q;\ a_i \geqq t\} - n \sum_{j\leqq p} f_j(t)\right\}.$$

Then there are disjoint subsets $(H_j)_{j\leqq p}$ of $\{1, \cdots, q\}$ with $\sum_{j\leqq p} \operatorname{card} H_j \geqq q - h - p$, and for each $j \leqq p$, for each $t \in [0, 1]$,

$$\operatorname{card}\{i \in H_j;\ a_i \geqq t\} \leqq nf_j(t). \tag{3.2}$$

Proof. The proof is easy, but very boring. For $x \in R$, let $[x]$ be the integer part of x. For $1 \leqq j \leqq p$, let $n_j = [nf_j(0)]$. For $1 \leqq l \leqq n_j$, let $b_{l,j} = \operatorname{Sup}\{t;\ 0 \leqq t \leqq 1,\ nf_j(t) \geqq l\}$. (Since $l \leqq n_j$, the set is not empty.) Since f_j is left-continuous, $nf_j(b_{l,j}) \geqq l$. Note also

that $b_{l,j}$ is nonincreasing in l, i.e., $b_{l+1,j} \leqq b_{l,j}$ for $1 \leqq l$, $l+1 \leqq n_j$. We first prove the following fact.

FACT. *For* $0 \leqq t \leqq 1$,

$$\text{card}\,\{1 \leqq l \leqq n_j;\ b_{l,j} \geqq t\} > nf_j(t) - 1. \tag{3.3}$$

Proof of the Fact. Let r be the largest integer with $r \leqq nf_j(t)$, so $r > nf_j(t) - 1$. By definition of $b_{l,j}$, $b_{l,j} \geqq t$ for $1 \leqq l \leqq r$. □

We return to the proof of Lemma 2. Consider the index set $U = \{(l,j);\ 1 \leqq j \leqq p,\ 1 \leqq l \leqq n_j\}$. For $u = (l,j) \in U$, let $b_u = b_{l,j}$. By summation of the inequalities (3.3), we get for each t,

$$\text{card}\,\{u \in U;\ b_u \geqq t\} > n \sum_{j \leqq p} f_j(t) - p.$$

So, by definition of h,

$$\sup_{0 \leqq t \leqq 1} \{\text{card}\,\{i \leqq q;\ a_i \geqq t\} - \text{card}\,\{u \in U;\ b_u \geqq t\}\} < h + p.$$

Using Lemma 1, we get $H \subseteq \{1, \cdots, q\}$ with card $H \geqq q - h - p$ and a one-to-one map ϕ from H to U such that $a_i \leqq b_{\phi(i)}$ for $i \in H$. For $j \leqq p$, define

$$H_j = \{i \in H;\ \phi(i) \text{ is of the form } (l,j) \text{ for some } l \leqq n_j\}.$$

Since $H = \bigcup_{j \leqq p} H_j$, we have $\sum_{j \leqq p} \text{card}\, H_j \geqq q - h - p$.

We now prove (3.2).

Fix $1 \leqq j \leqq p$ and $0 \leqq t \leqq 1$. Let r be the largest integer with $b_{r,j} \geqq t$. So $f_j(t) \geqq f_j(b_{r,j}) \geqq r/n$. Also, card $\{l \leqq n_j;\ b_{l,j} \geqq t\} = r \leqq nf_j(t)$. Since ϕ is one to one,

$$\begin{aligned} \text{card}\,\{i \in H_j;\ a_i \geqq t\} &\leqq \text{card}\,\{i \in H_j;\ b_{\phi(i)} \geqq t\} \\ &\leqq \text{card}\,\{l \leqq n_j;\ b_{l,j} \geqq t\} \leqq nf_j(t). \end{aligned}$$ □

Remark. The proof actually gives $\sum_{j \leqq p} \text{card}\, H_j \geqq q - m$, where m is the largest integer $< h + p$.

For two positive measures (not necessarily probabilities) η, ν on $[0,1]$, [11] shows the relevance of the relation $\eta \ll \nu$ given for each $t \in [0,1]$, $\eta([t,1]) \leqq \nu([t,1])$. Before we state Lemma 4, we explain its basic idea. There is a measure ν_i on $[0,1]$ with $\nu_i([t,1]) = f_i(t)$ (since f_i is nonnegative, nonincreasing, and left-continuous). Condition (3.2) reads

$$\text{for each } j \leqq p, \quad (1/n) \sum_{i \in H_j} \delta_{a_i} \ll \nu_j. \tag{3.4}$$

Also $H_j \subseteq \{1, \cdots, n_j\}$ and we have n_j bins. Bin number l, for $l \leqq n_j$, has occupancy level a_l. Then (3.4) also reads

$$(1/n_j) \sum_{l \in H_j} \delta_{a_l} \ll n\nu_j/n_j.$$

So, occupancy level of the bins is controlled by ν_j.

The basic idea of the construction is as follows: The decomposition of μ as in Theorem A contains (in an abstract way) directions on how to actually pack a given sequence of items (see [11]). We will try to follow these directions; the packing will be done in about $\sqrt{n}$ steps; after each step, we want to make sure (to make further steps possible) that the average level of bin occupancy is controlled (in the sense of (3.4)) by some suitable distribution.

The following lemma is purely technical, and will be needed in the proof of the basic Lemma 4.

LEMMA 3. *Let γ be a positive measure on $[0,1]^2$. Let $p>0$. Consider a sequence $(b_i)_{i\leqq p}$ such that $\sum_{i\leqq p} b_i = \|\gamma\|$. ($\|\cdot\|$ is the total variation norm. Since $\gamma \geqq 0$, $\|\gamma\| = \gamma(1) = \int d\gamma(x)$.) Then there is a sequence $0 = u_p \leqq u_{p-1} \leqq \cdots \leqq u_0 = 1$ and for $1 \leqq i \leqq p$ a positive measure γ_i such that*

$$\gamma = \sum_{i\leqq p} \gamma_i, \tag{3.5}$$

$$\gamma_i \text{ is supported by } [u_i, u_{i-1}]\times[0,1], \text{ and} \tag{3.6}$$

$$\|\gamma_i\| = b_i. \tag{3.7}$$

Proof. Define for $1 \leqq i \leqq p-1$,

$$u_i = \operatorname{Sup}\left\{0 \leqq t \leqq 1;\ \gamma([t,1]\times[0,1]) \geqq \sum_{j\leqq i} b_j\right\}.$$

Let $u_0 = 1$ and $u_p = 0$. We have for $1 \leqq i \leqq p-1$,

$$\gamma([u_i, 1]\times[0,1]) \geqq \sum_{j\leqq i} b_j. \tag{3.8}$$

If $u_i < 1$, then for $t > u_i$, we have

$$\gamma([t,1]\times[0,1]) \leqq \sum_{j\leqq i} b_j.$$

So,

$$\gamma(]u_i, 1]\times[0,1]) \leqq \sum_{j\leqq i} b_j. \tag{3.9}$$

This still holds if $u_i = 1$. We now construct γ_i by induction over i such that (3.6) and (3.7) hold, and also

$$\gamma - \sum_{j\leqq i} \gamma_j \text{ is supported by } [0, u_i]\times[0,1]. \tag{3.10}$$

We first construct γ_1. Let $b = \gamma([u_1,1]\times[0,1])$ and $a = \gamma(]u_1,1]\times[0,1])$. We have $a \leqq b_1 \leqq b$. Denote by ξ (respectively, η) the restriction of γ to $]u_1,1]\times[0,1]$ (respectively, $\{u_1\}\times[0,1]$). So $\|\xi\| = a$ and $\|\eta\| = b-a$. If $a = b$, we set $\gamma_1 = \eta$. Otherwise, we take $\gamma_1 = \xi + [(b_1 - a)/(b-a)]\eta$. We suppose now that γ_i has been constructed, for $i < p$, and we construct γ_{i+1}. Let $\gamma' = \gamma - \sum_{j\leqq i}\gamma_j$. By (3.10), γ' is supported by $[0, u_i]\times[0,1]$. By (3.5), (3.7), (3.8), and (3.9),

$$b = \gamma'([u_{i+1}, u_i]\times[0,1]) = \gamma'([u_{i+1},1]\times[0,1]) \geqq b_{i+1},$$

$$a = \gamma'(]u_{i+1}, u_i]\times[0,1]) = \gamma'(]u_{i+1},1]\times[0,1]) \leqq b_{i+1}$$

since $\gamma'(]u_i,1]\times[0,1]) = 0$. Denote by ξ (respectively, η) the restriction of γ' to $]u_{i+1}, u_i]\times[0,1]$ (respectively, $\{u_{i+1}\}\times[0,1]$), so $\|\xi\| = a$ and $\|\eta\| = b-a$. If $b = a$, we can take $\gamma_{i+1} = \xi$. Otherwise, we take

$$\gamma_{i+1} = \xi + \{(b_{i+1} - a)/(b-a)\}\eta.$$

The proof is complete.

The basic lemma is as follows.

LEMMA 4. *Let $1 \leqq m \leqq n$. Let γ be a positive measure on $[0,1]^2$. Assume $(m/n) \leqq \|\gamma\| \leqq (m+1)/n$. Consider the three positive measures, given by, for each Borel set U:*

$$\gamma'(U) = \gamma(U\times[0,1]),$$

$$\gamma''(U) = \gamma([0,1]\times U), \quad \text{and}$$

$$\eta(U) = \gamma(\{(x,y): x+y \in U\})$$

(in other words, γ' and γ'' are the marginals of γ, and η is the distribution of $x+y$).

Consider two sequences $(l_i)_{i\leqq m}$, $(y_i)_{i\leqq m}$ of numbers, $0\leqq l_i,\ y_i\leqq 1$. Assume that

$$\text{for each } t\in[0,1],\quad \operatorname{card}\{i\leqq m;\ l_i\geqq t\}\leqq n\gamma'([t,1]), \tag{3.11}$$

$$\text{for each } t\in[0,1],\quad \operatorname{card}\{i\leqq m;\ y_i\geqq t\}\leqq n\gamma''([t,1]). \tag{3.12}$$

Let $1\leqq k\leqq m/4$. Then there exist three subsets A, B, C of $\{1,\cdots,m\}$ such that $B\subseteq A$, $\operatorname{card} B=\operatorname{card} C$,

$$\operatorname{card} A\geqq m-\sqrt{m/k},\quad \text{and} \tag{3.13}$$

$$\operatorname{card} B\geqq m-7\sqrt{mk}, \tag{3.14}$$

and there exists a one-to-one and onto map ϕ from B to C such that if we set $z_i=l_i$ for $i\in A\backslash B$ and $z_i=l_i+y_{\phi(i)}$ for $i\in B$, then

$$\text{for each } t\geqq 0,\quad \operatorname{card}\{i\in A;\ z_i\geqq t\}\leqq n\eta([t,\infty[). \tag{3.15}$$

Interpretation. For simplicity, assume $\|\gamma\|=1$ and $m=n$. Consider a pair L, Y of random variables such that the pair (L, Y) has distribution γ. Then L is distributed according to γ', Y is distributed according to γ'', and $L+Y$ according to η. We are now given the items $L_1,\cdots,L_n$, $Y_1,\cdots,Y_n$, whose distributions of sizes are nicely controlled by γ' and γ'' as in (3.11) and (3.12). We try to pair the items by finding a one-to-one map ϕ: $\{1,\cdots,n\}\to\{1,\cdots,n\}$ such that the distribution of the sizes of the pairs is controlled by η, i.e.,

$$\text{for each } t\in[0,1],\quad \operatorname{card}\{i\leqq n;\ l_i+y_{\phi(i)}\geqq t\}\leqq n\eta([t,\infty[). \tag{3.16}$$

(Elements that are paired will go to the same bin.) This is not quite possible, so we have to settle for less, and dump some of the items. We will apply Lemma 4 to a situation where the items Y_i are of size $\leqq 1/k$, and the size of items L_i is of order 1. Since the items (Y_i) are smaller, we can afford to disregard more of them. We disregard the items Y_i for $i\notin B$, (at most $7\sqrt{mk}$ of them) and the items L_i for $i\notin A$ (at most $\sqrt{m/k}$ of them). If $i\in A\backslash B$, L_i will not be paired with any element (and will go alone to a bin). If $i\in A$, L_i will be paired with $Y_{\phi(i)}$. Formula (3.16) is then the appropriate substitute for (3.15).

Proof. Set $a=[\sqrt{m/k}]$. Since $m/k\geqq 4$, $\sqrt{m/k}\geqq 2$, so $a\geqq\sqrt{m/k}/2$. Let $p=[n\|\gamma\|/a]$. We have

$$p\leqq n\|\gamma\|/a\leqq 2\sqrt{k/m}\,n\|\gamma\|\leqq 2(m+1)\sqrt{k/m}\leqq 4\sqrt{mk}\quad(\text{since } m\geqq 1,\ m+1\leqq 2m).$$

For $i\leqq p$, let $b_i=a/n$. Let $b_{p+1}=\|\gamma\|-pa/n\leqq a/n$. We apply Lemma 3, to write $\gamma=\sum_{1\leqq i\leqq p+1}\gamma_i$, such that $\|\gamma_i\|=b_i$ and γ_i is supported by $[u_i,u_{i-1}]\times[0,1]$, where $0=u_{p+1}\leqq u_p\leqq\cdots\leqq u_1\leqq u_0=1$.

CLAIM. For $0\leqq t\leqq 1$, we have

$$\operatorname{card}\{1\leqq i\leqq p;\ u_i\geqq t\}\geqq(n/a)\gamma'([t,1])-1. \tag{3.17}$$

For $t=0$, this is equivalent to $p\geqq(n/a)\|\gamma\|-1$, which is true. Suppose now $t>0$, let r be the largest integer with $u_r\geqq t$, so $u_{r+1}<t$. We have

$$\gamma'([t,1])\leqq\gamma'(]u_{r+1},1])\leqq\|\gamma\|-\sum_{i>r+1}b_i\leqq(r+1)a/n.$$

On the other hand,

$$\operatorname{card}\{1\leqq i\leqq p;\ u_i\geqq t\}\geqq r$$

which proves the claim.

Let $W=\{(i,j);\ 1\leqq i\leqq p,\ 1\leqq j\leqq a\}$. For $w=(i,j)\in W$, set $v_w=u_i$. For $0\leqq t\leqq 1$, we have, by (3.17),

$$\begin{aligned}\operatorname{card}\{w\in W;\ v_w\geqq t\} &= a\ \operatorname{card}\{1\leqq i\leqq p;\ u_i\geqq t\}\\ &\geqq n\gamma'([t,1])-a.\end{aligned}$$

So,

$$\operatorname{card}\{i\leqq m;\ l_i\geqq t\}-\operatorname{card}\{w\in W;\ v_w\geqq t\}\leqq a.$$

It follows by Lemma 1 that there is a set $A\subseteq\{1,\cdots,m\}$ with $\operatorname{card} A\geqq m-a\geqq m-\sqrt{m/k}$, and a one-to-one map $\psi: A\to W$ such that $l_j\leqq v_{\psi(j)}$ for $j\in A$. For $i\leqq p$, let $K_i=\{j\in A;\ v_{\psi(j)}=u_i\}$. The sets K_i, $i\leqq p$ form a partition of A. So, $\sum_{1\leqq i\leqq p}\operatorname{card} K_i\geqq m-a$. Also $\operatorname{card} K_i\leqq a$. For $1\leqq i\leqq p+1$, define

$$f_i(t)=\gamma_i([0,1]\times[t,1]).$$

Since $\gamma=\sum_{1\leqq i\leqq p+1}\gamma_i$, we have for $0\leqq t\leqq 1$, $\gamma''([t,1])=\sum_{1\leqq i\leqq p+1}f_i(t)$. Since $|f_{p+1}(t)|\leqq b_{p+1}\leqq a/n$, we have $\sum_{1\leqq i\leqq p}f_i(t)\geqq\gamma''([t,1])-a/n$. It follows from (3.12) that

$$\operatorname*{Sup}_{0\leqq t\leqq 1}\left\{\operatorname{card}\{i\leqq m;\ y_i\geqq t\}-n\sum_{1\leqq i\leqq p}f_i(t)\right\}\leqq a.$$

From Lemma 2, there exist disjoint subsets $(H_i)_{1\leqq i\leqq p}$ of $\{1,\cdots,m\}$ such that $\sum_{1\leqq i\leqq p}\operatorname{card} H_i\geqq m-a-p$ and such that

(3.18) for each $i\leqq p$ and for each $t\in[0,1]$, $\operatorname{card}\{j\in H_i;\ y_j\geqq t\}\leqq nf_i(t)$.

For $1\leqq i\leqq p$, let M_i be a subset of K_i such that $\operatorname{card} M_i=\min\{\operatorname{card} K_i,\operatorname{card} H_i\}$. Let $B=\bigcup_{1\leqq i\leqq p}M_i$. For $1\leqq i\leqq p$, since $\operatorname{card} K_i\leqq a$, we have $\operatorname{card}(K_i\backslash M_i)\leqq a-\operatorname{card} H_i$. So,

$$\begin{aligned}\sum_{i\leqq p}\operatorname{card}(K_i\backslash M_i) &\leqq ap-\sum_{i\leqq p}\operatorname{card} H_i\\ &\leqq ap+a-m+p\leqq n\|\gamma\|-m+a+p\\ &\leqq a+p+1,\end{aligned}$$

since $n\|\gamma\|-m\leqq 1$ and $p=[n\|\gamma\|/a]$. So,

$$\operatorname{card} B=\sum_{i\leqq p}\operatorname{card} M_i=\sum_{i\leqq p}(\operatorname{card} K_i-\operatorname{card}(K_i\backslash M_i))\geqq m-2a-p-1\geqq m-7\sqrt{mk}.$$

Since $\operatorname{card} M_i\leqq\operatorname{card} H_i$, there is a one-to-one map ϕ_i from M_i into H_i. Consider the map ϕ from B to $\{1,\cdots,m\}$ given by $\phi(j)=\phi_i(j)$ for $j\in M_i$. We define $C=\phi(B)$. We set $z_j=l_j$ for $j\in A\backslash B$ and $z_j=l_j+y_{\phi(j)}$ for $j\in B$. For $j\in M_i$, $l_j\leqq v_{\psi(j)}=u_i$. We have for $1\leqq i\leqq p$,

$$\operatorname{card}\{j\in K_i;\ z_j\geqq u_i\}\leqq\operatorname{card} K_i\leqq a=n\|\gamma_i\|.$$

So, for $t\leqq u_i$, $\operatorname{card}\{j\in K_i;\ z_j\geqq t\}\leqq n\gamma_i([0,1]\times[t-u_i,\infty[)$. Now let $t>u_i$, $j\in K_i$, and $z_j\geqq t$. Since $z_j>u_i\geqq l_j$, we have $j\in M_i$ (otherwise $z_j=l_j$). Moreover, $z_j=l_j+y_{\phi(j)}\geqq t$, so $y_{\phi(j)}\geqq t-l_j\geqq t-u_i$. From (3.18), we have

$$\begin{aligned}\operatorname{card}\{j\in K_i;\ z_j\geqq t\} &\leqq\operatorname{card}\{l\in H_i;\ y_l\geqq t-u_i\}\\ &\leqq n\gamma_i([0,1]\times[t-u_i,\infty[).\end{aligned}$$

So for any t,

$$\operatorname{card}\{j\in K_i;\ z_j\geqq t\}\leqq n\gamma_i([0,1]\times[t-u_i,\infty[).$$

Since γ_i is supported by $[u_i, u_{i-1}]\times R$,

$$\begin{aligned}\operatorname{card}\{j\in A;\, z_j\geqq t\} &\leqq \sum_{i\leqq p} n\gamma_i([0,1]\times[t-u_i,\infty[)\\ &= \sum_{i\leqq p} n\gamma_i([u_i,u_{i-1}]\times[t-u_i,\infty[).\end{aligned}$$

Let η_i be the image of γ_i by the map $(x, y)\to(x+y)$, that is, for each $i\leqq p$

$$\begin{aligned}\eta_i([t,\infty[) &= \gamma_i(\{(x,y);\,(x+y)\geqq t\})\\ &\geqq \gamma_i([u_i,u_{i-1}]\times[t-u_i,\infty[).\end{aligned}$$

So,

$$\begin{aligned}n\eta([t,\infty[) &\geqq n\sum_{i\leqq p}\eta_i([t,\infty[) \geqq n\sum_{i\leqq p}\gamma_i([u_i,u_{i-1}]\times[t-u_i,\infty[)\\ &\geqq \operatorname{card}\{j\in A;\, z_j\geqq t\}.\end{aligned}$$

The proof is complete.

We finish the section with an obvious and well-known fact.

LEMMA 5. *A collection $(y_j)_{j\leqq m}$ of items of sizes $y_j\leqq\theta<1$ can be packed in at most $1+\sum_{j\leqq m} y_j/(1-\theta)$ bins.*

Proof. In an optimal packing, at most $(1-\theta)$ of each bin (except for the last one) is wasted.

4. Proof of Theorem B. We have the decomposition of μ:

$$\mu=\alpha_0\delta_0+\sum_{k\geqq 1}\alpha_k\mathcal{P}(\nu_k).$$

As was shown by the proof of Theorem A in [11], the natural interpretation is as follows: α_k is the proportion of items packed in bins that contain k elements, and ν_k is the distribution of sizes inside such bins. We need some distributions associated with μ. For $l\geqq 1$, consider the positive measure λ_l such that

$$\lambda_l=\sum_{k\geqq l}(\alpha_k/k)\int_{R_k}\delta_{x_{k-l+1}}\,d\nu_k(x_1,\cdots,x_k)$$

so $\|\lambda_l\|=\sum_{k\geqq l}\alpha_k/k=\beta_l$, and $\mu=\alpha_0\delta_0+\sum_{l\geqq 1}\lambda_l$. Since x_{k-l+1} is the lth largest element of $(x_1,\cdots,x_k)$, it is less than or equal to $1/l$, so λ_l is supported by $[0, 1/l]$. The natural interpretation is to say that (after normalization) λ_l is the distribution of the lth largest element in the bins that contain at least l elements. For $x\in R_k$, $k\geqq l$, let $u_{k,l}(x)=\sum_{k-l+1\leqq i\leqq k} x_i$. We note that $u_{k,l}(x)\leqq 1$. Consider the following two positive measures:

$$\eta_l=\sum_{k\geqq l}(\alpha_k/k)\int_{R_k}\delta_{u_{k,l}(x)}\,d\nu_k(x_1,\cdots,x_k),\quad\text{and}$$

$$\xi_l=\sum_{k\geqq l+1}(\alpha_k/k)\int_{R_k}\delta_{u_{k,l}(x)}\,d\nu_k(x_1,\cdots,x_k).$$

We note that $\eta_1=\lambda_1$, $\|\eta_l\|=\beta_l$, and $\|\xi_l\|=\beta_{l+1}$, and that η_l and ξ_l are supported by $[0, 1]$. The natural interpretation of η_l (respectively, ξ_l) is as follows. After normalization, η_l is the distribution of $\sum_{i\leqq l} X_i$ in bins that contain elements $X_1\geqq\cdots\geqq X_k$, $k\geqq l$ (respectively, $k\geqq l+1$).

We also need the distribution γ_l on $[0, 1]^2$ given by

$$\gamma_l=\sum_{k\geqq l+1}(\alpha_k/k)\int_{R_k}\delta_{v(x)}\,d\nu_k(x_1,\cdots,x_k)$$

where $v(x)=(u_{k,l}(x),\ x_{k-l})$. After normalization, γ_l expresses the joint distribution of $(u_{k,l}(x),\ x_{k-l})$ in bins that contain at least $l+1$ items. Note that $\|\gamma_l\|=\beta_{l+1}$. Using the fact that $u_{k,l}(x)+x_{k-l}=u_{k,l+1}(x)$, we have immediately the following lemma.

LEMMA 6. (a) *The first marginal of* γ_l *is* ξ_l, *the second is* λ_{l+1}.

(b) *The image of* γ_l *under the map* $(x, y)\to x+y$ *is* η_{l+1}.

We now set $q=[\sqrt{n}]$. For each i, let $f_i(t)=\lambda_i([t, 1])$, for $t>0$. So, $\sum_{i\geqq 1} f_i(t)=\mu([t, 1])$. For $t>1/i$, $f_i(t)=0$, so for $t\geqq 1/q$, we have $\sum_{i\leqq q} f_i(t)=\mu([t, 1])$. We note also that $f_i(0)=\beta_i$.

We now prove Theorem B. Consider a sequence $s_1, \cdots, s_n$ of items. Let $C_0=\{i\leqq n;\ s_i<1/q\}$. Let $\bar{C}=\{i\leqq n;\ s_i\geqq 1/q\}$. We have for each $1/q\leqq t\leqq 1$, card $\{i\in\bar{C};\ s_i\geqq t\}\leqq n\sum_{i\leqq q} f_i(t)+W^+$. We now use Lemma 2 (on the interval $[1/q, 1]$ instead of $[0, 1]$). Then we find disjoint subsets $(C_i)_{i\leqq q}$ of $\bar{C}$ with $\sum_{i\leqq q}$ card $C_i\geqq$ card $\bar{C}-q-W^+$ such that for each $i\leqq q$, for each $1/q\leqq t\leqq 1$, card $\{j\in C_i;\ s_j\geqq t\}\leqq nf_i(t)$. Now let $C'=\bar{C}\setminus\bigcup_{i\leqq q} C_i$, so card $C'\leqq q+W^+$. Since $s_j\geqq 1/q$ for $j\in\bar{C}$, we get for each $i\leqq q$, for each $t\in[0, 1]$,

$$\text{card}\,\{j\in C_i;\ s_j\geqq t\}\leqq nf_i(t). \tag{4.1}$$

We note that if we set $n_i=[n\beta_i]$, this implies card $C_i\leqq n_i$. We first pack the elements of C'. It can be done with card $C'\leqq q+W^+$ bins. To pack the elements of C_0, Lemma 5 shows that we need at most $1+(n/q)(1/(1-1/q))$ bins. For $n\geqq 4$, a simple computation shows that this number is less than or equal to $5\sqrt{n}/2$. If $1\leqq n\leqq 3$, we need at most n bins, and $n\leqq\sqrt{3n}$. Since $\sqrt{3n}\leqq 5\sqrt{n}/2$, we need at most $5\sqrt{n}/2$ bins for any n.

We now proceed to the main construction, the packing of the families $(C_l)_{l\leqq q}$. This is done in q steps. At step l, $l\leqq q$ we will pack the collection C_l. After the lth step is performed, we have two sets of bins. The first set D_l of bins is "dead." No elements will be added to these bins at further steps of the construction. The second set of bins (called live bins) has a cardinality $n_{l+1}=[n\beta_{l+1}]$, so for convenience we index it by $\{1, \cdots, n_{l+1}\}$, and denote these bins by $B_1, \cdots, B_{n_{l+1}}$. These bins already contain items. If L_i is the sum of the sizes of the items contained in bin B_i (i.e., level occupancy), we have the following condition (that allows us to continue the induction):

$$\text{for each } t \text{ in } [0, 1],\quad \text{card}\,\{i\leqq n_{l+1};\ L_i\geqq t\}\leqq n\xi_l([t, 1]). \tag{4.2}$$

Before we start the construction, we have no dead bins and a set of n_1 empty live bins. To pack C_1, we put each item in a different bin. Since $\eta_1=\lambda_1$, $\eta_1([t, 1])=f_1(t)$, (4.1) implies

$$\text{for each } t \text{ in } [0, 1],\quad \text{card}\,\{i\leqq n_1;\ L_i\geqq t\}\leqq n\eta_1([t, 1]).$$

We note that $\eta_1([t, 1])\leqq\xi_1([t, 1])+\alpha_1$. We use Lemma 2 (with $p=1$) to find a subset E_1 of $\{1, \cdots, n_1\}$ with card $E_1\geqq n_1-n\alpha_1-1$ such that

$$\text{for each } t \text{ in } [0, 1],\quad \text{card}\,\{i\in E_1,\ L_i\geqq t\}\leqq n\xi_1([t, 1]). \tag{4.3}$$

The bins of index which are in $\{1, \cdots, n_1\}\setminus E_1$ become "dead," so our collection D_1 of dead bins has now at most $n\alpha_1+1$ elements. From (4.3), since $\xi_1([0, 1])=\beta_2$, we have card $E_1\leqq n_2=[n\beta_2]$. By adding n_2-card E_1 bins to our collection of live bins, we can suppose that it has now n_2 elements and that (4.2) holds with $l=1$.

We suppose now that step l has been completed where $l<q$. So, the collections $C_1, \cdots, C_l$ are already packed and we pack C_{l+1}. We have n_{l+1} live bins $B_1, \cdots, B_{n_{l+1}}$, such that if L_i is the level of bin occupancy of B_i, (4.2) holds. Let us enumerate the

items of C_{l+1} as $Y_1, \cdots, Y_r$. From (4.1) we have

$$\text{for each } t \text{ in } [0,1], \quad \text{card}\{j \leqq r;\ Y_j \geqq t\} \leqq n f_{l+1}(t). \tag{4.4}$$

When we take $t=0$, this shows $r \leqq [nf_{l+1}(0)] = n_{l+1}$. By adding $n_{l+1}-r$ items of size zero to $Y_1, \cdots, Y_r$, we can assume that $r = n_{l+1}$ and that (4.4) still holds.

If $n_{l+1} \leqq 4(l+1)$, we can pack the n_{l+1} items into four new bins, since they are of size at most $1/(l+1)$. These bins become dead. (Each of the collections C_i, $i \geqq l+1$, can also be packed in four bins.) If $(l+1)4 \leqq n_{l+1}$, we can use Lemma 4 with $\gamma = \gamma_l$, $m = n_{l+1}$ and $k = l+1$; we note that $\gamma' = \xi_l$ and $\gamma'' = \lambda_{l+1}$, so (3.11) and (3.12) follow from (4.2) and (4.4), respectively. So, we can find three subsets A, B, and C of $\{1, \cdots, n_{l+1}\}$ such that $B \subseteq A$ with $\text{card}\, A \geqq n_{l+1} - \sqrt{n_{l+1}/(l+1)}$; $\text{card}\, C = \text{card}\, B \geqq n_{l+1} - 7\sqrt{n_{l+1}(l+1)}$, a one-to-one map ϕ from B to C such that if we set $Z_i = L_i$ for $i \in A \backslash B$ and $Z_i = L_i + Y_{\phi(i)}$ for $i \in B$, then for each $t \geqq 0$,

$$\text{card}\{i \in A;\ Z_i \geqq t\} \leqq n\eta_{l+1}([t, \infty[) = n\eta_{l+1}([t,1]). \tag{4.5}$$

If $i \notin A$, the bin B_i becomes dead. That creates at most $\sqrt{n_{l+1}/(l+1)}$ new dead bins. If $i \in A \backslash B$, the bin B_i stays alive but we add no new element to it. If $i \in B$, we put $Y_{\phi(i)}$ into B_i, and the bin B_i stays alive. At this point, the bins B_i, for i in A, are alive, and their level occupancy satisfies (4.5). We have packed all the $(Y_j)_{j \leqq n_{l+1}}$ except for $j \notin C$. This is at most $7\sqrt{n_{l+1}(l+1)}$ elements. Since these elements are of sizes $\leqq 1/(l+1)$, we can pack them into $7\sqrt{n_{l+1}/(l+1)}+1$ empty bins. These bins become dead. We now note that

$$\eta_{l+1}([t,1]) \leqq \xi_{l+1}([t,1]) + \alpha_{l+1}/(l+1).$$

Using (4.5) and Lemma 2 (with $p=1$) we can find $H \subseteq A$, with $\text{card}\,(A \backslash H) \leqq [n\alpha_{l+1}/(l+1)]+1$ such that

$$\text{for each } t \text{ in } [0,1], \quad \text{card}\{i \in H;\ Z_i \geqq t\} \leqq n\xi_{l+1}([t,1]). \tag{4.6}$$

The bins of $A \backslash H$ become dead. Taking into account the case $n_{l+1} \leqq 4(l+1)$, at this step, we have created at most $[n\alpha_{l+1}/(l+1)]+4+8\sqrt{n_{l+1}/(l+1)}$ dead bins. From (4.6), we have $\text{card}\, H \leqq [n\beta_{l+2}] = n_{l+2}$. The bins B_i, $i \in H$ stay alive, and we add to this collection $n_{l+2} - \text{card}\, H$ new empty bins. This gives us a collection of n_{l+2} live bins that we relabel $B_1, \cdots, B_{n_{l+2}}$. Obviously, their occupancy level that we call L_i again satisfies the following:

$$\text{for each } t \text{ in } [0,1], \quad \text{card}\{i \leqq n_{l+2};\ L_i \geqq t\} \leqq n\xi_{l+1}([t,1]).$$

This completes the induction. After the q steps are completed, we have created at most

$$\sum_{1 \leqq i \leqq q} n\alpha_i/i + \sum_{2 \leqq i \leqq q} 8\sqrt{n_i/i} + 4q$$

dead bins. From (4.2), our collection of live bins has at most $n\beta_{q+1}$ bins; since $n_i \leqq n\beta_i$ and $\beta_{q+1} = \sum_{i>q} \alpha_i/i$, we have succeeded in using $n\sum_{i \geqq 1} \alpha_i/i + 8\sqrt{n}(\sum_{2 \leqq i \leqq q} \sqrt{\beta_i/i}) + 4q$ bins to pack $C_1, \cdots, C_l$. To pack the whole family $\{s_1, \cdots, s_n\}$, we thus have used at most

$$W^+ + n\sum_{i \geqq 1} \alpha_i/i + 8\sqrt{n}\left(\sum_{2 \leqq i \leqq q} \sqrt{\beta_i/i}\right) + 15\sqrt{n}/2$$

bins. □

Acknowledgments. The authors thank the referees for their careful reading and helpful suggestions.

REFERENCES

[1] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Approximation algorithms for bin-packing—an updated survey*, in Algorithm Design for Computer System Design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, Berlin, New York, 1984, pp. 49–106.

[2] D. L. COHN, *Measure Theory*, Birkhäuser, Boston, 1980.

[3] M. R. GAREY AND D. J. JOHNSON, *Computers and Intractability; A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[4] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computers Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[5] J. F. C. KINGMAN, *Subadditive Processes*, Lecture Notes in Mathematics 539, Springer-Verlag, Berlin, New York, 1976, pp. 168–222.

[6] W. KNÖDEL, *A bin-packing algorithm with complexity* $O(n \log n)$ *and performance* 1 *in the stochastic limit*, in Proc. 10th Symposium on Mathematical Foundations in Computer Science, 1981; Strbske Pleso, Czechoslovakia, Lecture Notes in Computer Science 118, Springer-Verlag, Berlin, New York, 1981, pp. 269–278.

[7] G. S. LUEKER, *An average-case analysis of bin packing with uniformly distributed item sizes*, Tech. Report #181, Department of Information and Computer Science, University of California, Irvine, CA, 1982.

[8] G. LUEKER, *Bin packing with items uniformly distributed over intervals* [a, b], in Proc. 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, 1983, pp. 289–297.

[9] K. R. PARTHASARATHY, *Probability Measures on Metric Spaces*, Academic Press, New York, 1967.

[10] P. REVESZ, *The Laws of Large Numbers*, Academic Press, New York, 1968.

[11] W. RHEE, *Optimal bin-packing with items of random sizes*, Math. Oper. Res., 13 (1988), pp. 140–151.

[12] W. RHEE AND M. TALAGRAND, *Martingale inequalities and* NP-*complete problems*, Math. Oper. Res., 12 (1987), pp. 177–181.

[13] W. RUDIN, *Functional Analysis*, McGraw-Hill, New York, 1973.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 152-165, February 1989

010

MINIMAL THRESHOLD SEPARATORS AND MEMORY REQUIREMENTS FOR SYNCHRONIZATION*

EDWARD T. ORDMAN†

Abstract. Suppose that in a system of asynchronous parallel processes, certain pairs of processes mutually exclude one another (must not be in their critical sections simultaneously). This situation is modeled by a graph in which each process is represented by a vertex and each mutually excluding pair is represented by an edge. Henderson and Zalcstein have observed that if this graph is a threshold graph, then mutual exclusion can be managed by simple entrance and exit protocols using **PV**-chunk operations on a single shared variable whose possible values range from zero to t, the minimal threshold separator number of the graph. A new expression is given for this separator t of a threshold graph in terms of the normal decomposition of the threshold graph given by Zalcstein and Henderson. It is shown that $t+1$ values would be needed in the shared variable even if the mutual exclusion were being managed by the Fischer-Lynch test-and-set operator, which is considerably less restrictive than **PV**-chunk.

Key words. mutual exclusion, threshold graphs, synchronization primitives, test-and-set, **PV**-chunk

AMS(MOS) subject classifications. primary 68Q10; secondary 68R10, 05C70

1. Introduction.

Concurrent processing by several asynchronous processes presents control problems that have been widely studied [1], [2], [4]-[6], [9], [11], [14]. It may be, for instance, that due to a need to access shared resources, such as a printer or shared data, certain events must be prevented from happening simultaneously in two (or more) processes. One approach is to have a designated section of code in each process identified as a **critical section** and to have the processes execute a joint algorithm that controls access to the critical sections. This algorithm is typically represented within each process by two protocols: the **entry protocol**, which is a section of code executed by a process before it is admitted to its critical section (and in which it may loop for some time, if the shared resource is in use by other processes); and the **exit protocol**, which is executed when the process leaves its critical section and makes the shared resource available to other processes. See [2] for a more precise discussion.

The processes may communicate by sending and receiving messages or by manipulation of one or more **shared variables**. A large number of ways of accessing shared variables have been studied, such as elementary read and write operations [5], **P** and **V** operations [6], **PV**-chunk operations [9], and test-and-set operations [2], [14]. In this paper we will implicitly be using the model of critical sections and of test-and-set operations laid out in [2], which provided one of the principal motivations for this paper. One of our goals is to compare the **PV**-chunk and test-and-set operations in a certain context; we describe them further in § 4.

Many of the papers cited above study a form of the mutual exclusion problem in which only one process can be in a critical section at one time. In fact, there are problems of interest in which more than one process can be in a critical section at a time. An early example of such a problem in the literature is called the **dining philosophers problem** (see [10] and some earlier references cited therein); a very practical problem of this type is the **readers-and-writers problem** (see [17] and the references therein) in which several processes may be allowed to read a data item simultaneously, but a process may change the item (write it) only at a time when no other process is

* Received by the editors January 30, 1985; accepted for publication (in revised form) April 12, 1988. This work was partially supported by National Science Foundation grant DCR-8503922.

† Department of Mathematical Sciences, Memphis State University, Memphis, Tennessee 38152.

accessing it. A set of processes, some of which may not enter critical sections simultaneously, is sometimes called a generalized dining philosophers problem; one study of such problems is given in [12]. A second principal motivation for this paper was the hope that the sort of analysis done in [2] could be extended to generalized dining philosophers problems. Only a small start is made on that here: we discuss memory requirements for mutual exclusion (although not lockout prevention, starvation avoidance, etc.) for a limited class of problems, which do include some forms of readers and writers problems. We find that, for this very limited goal, **PV**-chunk operations are as powerful as test-and-set operations, although they appear not to be as memory-efficient in some other cases.

Some generalized dining philosophers problems correspond in a natural way to graphs. Suppose each vertex of a graph represents a process, and two vertices are **adjacent** (connected by an edge) if and only if the two corresponding processes cannot be executing their critical sections simultaneously. In this way each finite undirected graph without self-loops (we describe this more formally below) corresponds to a generalized dining philosophers problem. On the other hand, not every generalized dining philosophers problem corresponds to a graph. Consider four processes, each of which requires two tape drives, in an environment where four tape drives are available. Clearly any two processes can (if all goes well) obtain two drives each and proceed, so the corresponding "graph" would have no edges. However, three processes can never acquire enough tape drives (and enter their critical sections) at once, so a hypergraph would be required to model this situation. For some further discussion of this analogy see [9], [13]. We will be considering mutual exclusion in systems of processes represented by a certain class of graphs, the threshold graphs. Graph theory background is given in the next section.

2. Graph theory preliminaries. By a **graph** $G = (V, E)$ we mean a finite set of vertices V together with a finite set of edges E, each of which is a different unordered pair of distinct vertices: that is, a finite undirected graph without self-loops or parallel edges. If a and b are vertices we say that they are **adjacent** if (a, b) (or equivalently (b, a)) is an edge of G; we also say that this edge **connects** a and b.

Threshold graphs were introduced in [3] and have been studied extensively [9], [11], [13], [15], [16]; we will rely very heavily on [9]. A graph $G = (V, E)$ is a **threshold graph** if there is an integer t called the **threshold** (or sometimes the **separator**), and with each vertex x in V is associated a nonnegative integer label $a(x)$ such that a subset S of N is **stable** (no two nodes in it are connected by an edge in E) if and only if the sum of the $a(s)$ for all s in S is less than or equal to t. (We will call such a labeling, including knowing t, a **threshold labeling**). A great many other characterizations of threshold graphs are known (see, for example, [3], [9], [13]); some that we need are recalled at the start of § 4.

A graph is a threshold graph if and only if it has a threshold labeling. The labeling, however, is not unique. In [16] Orlin has given an algorithm for determining the **minimal separator** t (and an associated labeling) that will work for a given threshold graph. In § 4 we use a slight modification of the **normal form** for a threshold graph (implied in Corollary 1B, [3, p. 151] described in detail and named in [9]) to give a more closed form for the minimal value of t, and to see that this minimal value of t is important in determining the minimum amount of shared memory to do mutual exclusion.

In § 5 we study a mutual exclusion problem (Example 5.2) for a generalized dining philosophers problem that corresponds, in the sense described in § 1, to a threshold

graph. The connection between threshold graphs and **PV**-chunk operations has already been discussed in [9]. We find that the minimal separator t is a measure of the amount of memory needed to enforce mutual exclusion (without lockout prevention) in a system of processes represented by a threshold graph, using **PV**-chunk operations; we further see that at least this much memory is required, even if test-and-set operations are used instead. We see that test-and-set is, in cases other than threshold graphs, less demanding of memory than **PV**-chunk. We also look at a simple application to concurrent accesses of data bases.

3. The minimal separator of a threshold graph. We review some graph theory terminology. If $G=(V, E)$ is a graph and V' is a subset of V, the **subgraph** of G **induced** by V' is the graph whose vertices are the vertices of V' and whose edges are all edges of E which connect points in V'. A graph is a **clique** if every two points in it are adjacent; a subgraph of another graph G is a clique if it is a clique considered as a graph by itself, and is a maximal clique in G if it is not contained in any larger subgraph of G which is a clique. The clique with n vertices is denoted K_n.

The **degree** of a vertex is the number of vertices adjacent to it. We call a vertex **isolated** if it has degree zero, and **nonisolated** otherwise. We will call a vertex **dominating** (this is *not* a standard notation) if it is adjacent to every nonisolated vertex. By the **neighborhood** $N(x)$ of a vertex x we mean the set of vertices adjacent to x, together with x itself.

By way of making these terms clearer, we restate a well-known fact [3] about threshold graphs: no threshold graph may have as an induced subgraph any of the graphs shown in Fig. 1. These are the path on four vertices, P_4; the cycle on four vertices, C_4; and the union of two disjoint edges, $2K_2$ (note that a single edge is a clique on two points, K_2).

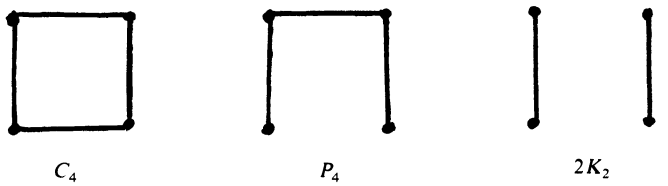


FIG. 1. *Graphs* C_4, P_4, *and* $2K_2$.

To prove that none of these graphs can be an induced subgraph of a threshold graph G, suppose that one of them is and label the four vertices w, x, y, and z. Suppose also that (w, z) and (x, y) are edges, but that (w, y) and (x, z) are not (e.g., label the vertices in each graph in Fig. 1 clockwise from upper left.) We will consider the labelings from a threshold labeling of G. Clearly, $a(w)+a(z)>t$ and $a(x)+a(y)>t$, since these pairs of vertices induce edges; clearly, $a(w)+a(y)\leqq t$ and $a(x)+a(z)\leqq t$, since these pairs do not. Adding the two pairs of inequalities produces a contradiction.

We also recall that an induced subgraph of a threshold graph must be a threshold graph; one simply restricts the threshold labeling to the subset of vertices and retains the same separator t.

Given a set of vertices V there may be various labelings $a(x)$ of the vertices and various separators t associated with them that lead to the same threshold graph G. For a given G, we want to determine the smallest possible t. This has been done in [16], which gives an algorithm to compute the smallest possible t; we approach the

problem slightly differently to get t in a more explicit form we can apply later in the paper. To this end, we need to recall and modify slightly the definition of "normal form" of a threshold graph given in [9].

LEMMA 3.1. *Let G be a threshold graph with an associated threshold labeling. Let x be a vertex with a label as large as any other label in G. Then x is a dominating vertex of G.*

Proof. Let z be any nonisolated node. Then it is connected to some node y and $a(z)+a(y)>t$. But $a(x)\geqq a(y)$, so $a(z)+a(x)>t$ and z is adjacent to x. □

In [3] threshold graphs are characterized by the fact that the three graphs of Fig. 1 cannot occur as induced subgraphs. We will need an additional characterization.

THEOREM 3.2. *For a graph G, the following are equivalent:*

(a) *G is a threshold graph.*

(b) *Every induced subgraph of G (including G itself) has a dominating node.*

(c) *G does not have as an induced subgraph the graphs P_4, $2K_2$, or C_4.*

Proof. Items (a) and (c) have been proved to be equivalent in [3]. Item (a) implies (b) by Lemma 3.1, since every induced subgraph of a threshold graph is threshold. Item (b) implies (c), since none of P_4, $2K_2$, or C_4 has a dominating vertex. □

The fact that (b) implies (a) is already implicit in [3]. That paper also contains the following Corollary 1B. A graph $G=(V, E)$ is threshold if and only if there is a partition of V into disjoint sets A, B, and an ordering $a_1, a_2, \cdots, a_k$ of A such that no two vertices in A are adjacent; every two vertices in B are adjacent; and if $j\leqq k$, then $N(a_j)\subseteq N(a_k)$. This fact is developed considerably in [9]; we will expand on it further here.

Given a threshold graph, we construct the normal form as follows. Choose all isolated vertices and put them in class D_0; take all dominating vertices and put them in class C_1. The subgraph of G induced by the remaining vertices we call G_1. For each consecutive subgraph G_k, place the isolated vertices in D_k, and the dominating vertices in C_{k+1}. Continue until G_{n+1} is empty.

Note the following:

(a) No vertex in any D_k is connected to any other vertex of any D_j (including $k=j$).

(b) Every vertex of every C_k is connected to every vertex of every C_j (including $k=j$).

(c) Every vertex of D_k is connected to every vertex of C_j for $j\leqq k$, but to no other vertices.

We may need to rearrange the last sets slightly to guarantee that both C_n and D_n are nonempty and that both C_{n+1} and D_{n+1} are empty. We distinguish, temporarily, two cases. If C_{n+1} is empty, D_n is nonempty (in fact it has at least two vertices, since if there were only one it would have been in C_n) and C_n is nonempty (else the construction would have stopped sooner). In the second case, C_{n+1} is nonempty. Then C_{n+1} must contain at least two vertices; otherwise the one vertex would have been in D_n. In this case we arbitrarily choose one vertex of C_{n+1}, move it to D_{n+1} (previously empty), and increase n by one. Thus we also have C_n and D_n both nonempty, and proceed with further analysis.

It is easy to see that all C_k and D_k are nonempty for $1\leqq k\leqq n$, since before each removal there are dominating vertices by Theorem 3.2 and their removal must leave some vertices isolated or else each dominating vertex of the newly reduced graph would have been already dominating prior to the reduction.

We call the resulting decomposition of G into $(D_0, D_1, \cdots, D_n, C_1, C_2, \cdots, C_n)$ the **normal form** of G. It is unique except perhaps for the choice of one node when we combined the two cases above; we tolerate that since it simplifies calculations below.

In Fig. 2 we illustrate the normal form, and the labeling that will be introduced below. On the left we show the graph G; the right shows the same graph with the vertex in C_1 labeled 11, that in C_2 labeled 8, those in D_1 labeled 1, and those in D_2 labeled 4. For this graph and labeling, $t = 11$. To convince ourselves that lower labels will not work, note that (once r, s, and u are labeled 1 and the separator is t) vertex x must have label at most $t-3$ since x, r, s, u induce no edges; now x and w induce an edge so w and similarly z must have label at least 4. Since r, s, u, z, and w induce no edges, t must be at least 11. The reader may find it helpful to delete vertex w from G and find the normal form and a labeling.

THEOREM 3.3. *Let G be a threshold graph and $(D_0, D_1, \cdots, D_n, C_1, C_2, \cdots, C_n)$ its normal form. Let d_k denote the number of vertices in D_k. Then the vertices can be labeled so as to give a threshold labeling with separator t satisfying*

$$t+1 = \prod (d_k+1), \qquad k = 1, \cdots, n.$$

Proof. Our labeling method is the same as that of [9] and [16]. For simplicity of formulas we define $g_i = \prod (d_j+1) - 1$, where the product is for $j = 1, \cdots, i$; by definition $g_0 = 0$. Assign each element of D_0 the label 0 (they lie on no edges). Assign each element of D_1 the label 1; the total of all the labels assigned so far is d_1, which is equal to g_1. Assign each element of D_2 the label d_1+1; the total of the labels in D_2 is $d_2(d_1+1)$ and the grand total so far is g_2. We show by induction that when we assign each element of D_k the label $g_{k-1}+1$, the labels in D_k will total $d_k(g_{k-1}+1)$ and the sum of the labels to this point will be g_k. The induction step is to observe that $g_k + d_{k+1}(g_k+1) = g_{k+1}$, that is,

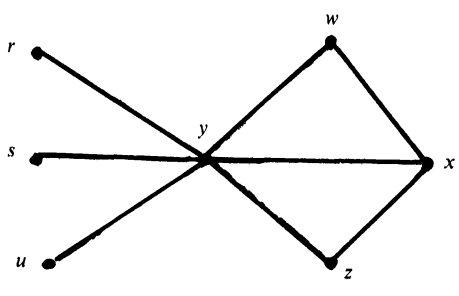
$$\begin{aligned} &\prod (d_j+1) - 1 + d_{k+1} \prod (d_j+1) && (j = 1, \cdots, k) \\ &\quad = (d_{k+1}+1) \prod (d_j+1) - 1 && (j = 1, \cdots, k) \\ &\quad = \prod (d_j+1) - 1, && (j = 1, \cdots, k+1) \end{aligned}$$

as desired.

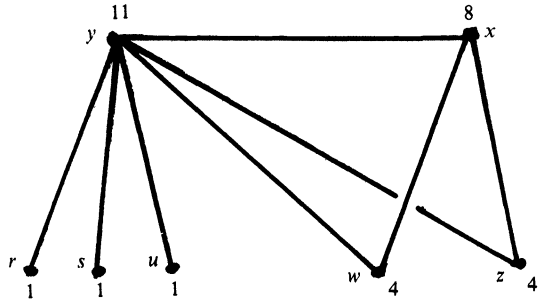
Now we let $t = g_n$, and for each k we assign each element of C_k the label $g_n - g_{k-1}$. We must show that this gives a threhold labeling of G. Note the following:

(a) No two vertices of any two D_k are connected, since the labels in all the D_k total g_n.

(b) Any two vertices in any C_k are connected, since each such point has label at least $g_n - g_{n-1}$. This must exceed half of g_n since $g_n = (g_{n-1}+1)(d_n+1) - 1$ and d_n is at least 1.



Graph G



Graph G with threshold labelling

FIG. 2.

(c) To test if a vertex in D_k is connected to a vertex in C_j, note that their labels total $(g_{k-1}+1)+(g_n-g_{j-i})$. This exceeds t exactly if k exceeds j, as required.

This completes the proof of Theorem 2.3. □

It can be shown that this is the same labeling as that in [16]. We could then rely on the proof there that the resulting t is minimal. However, a distinctly different proof of minimality will follow from the results of § 4 below.

4. Process synchronization. Now we turn to the problem of managing asychronous processes. We are motivated primarily by the considerations in [9] and in [2]; the reader is referred to [2] for a more complete background and terminology. Suppose we have a number of processes, some of which conflict with each other (e.g., they demand access to the same resources that cannot be shared simultaneously by all processes wanting them). We connect this to the considerations above by supposing that each vertex of the graph G represents a process, and that two processes conflict precisely if there is an edge connecting those vertices.

For example, suppose there is a record in a file consisting of two fields, a name (the key) and an address. Suppose there are five transactions in the system, each wanting to locate the same record, and then carry out the following tasks:

Process A: Read the name (that is, locate the record and confirm that it exists, no further use of it).
Processes B and C: Read the address.
Process D: Change the address in the existing record.
Process E: Change the name (key) and address.

This is a slight generalization of a conventional readers-and-writers problem. Processes A, B, and C are "readers" and all can access the record at once and still produce consistent results. Process D can proceed simultaneously with Process A but not with B or C; Process E cannot proceed at the same time as any of the others. Drawing a graph with vertices A through E and drawing the appropriate edges yields the graph of Fig. 3(a); this graph is, in fact, a threshold graph, with a threshold labeling as shown in Fig. 3(b). This example is further expanded in § 5.

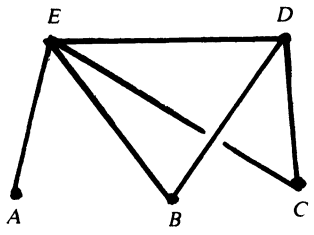


FIG. 3(a)

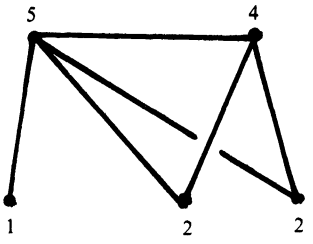


FIG. 3(b)

Suppose we are given a system of asynchronous processes and a set of pairs of these processes such that two processes in a pair cannot proceed simultaneously; i.e., members of each pair mutually exclude each other in the sense discussed in § 1 above and defined carefully in [2]. Each process has a piece of code called a "critical section"; we must provide entry and exit protocols for all processes in the system such that execution of these protocols will guarantee that two processes in a given pair will never be in their critical sections simultaneously. The graph G, with one vertex corresponding to each process and one edge for each mutually excluded pair of

processes, will be called the **exclusion graph** of the system. (For other work with this graph, see [15].)

Our treatment is much weaker than that in [2] in that we do not consider lockout prevention or any sort of fairness condition (e.g., bounded waiting). However, the treatment in [2] assumes that only one process out of the N processes in the system can be in its critical section at once: that is, the exclusion graph is a clique. Here we deal with an exclusion graph that is a threshold graph, and any set of processes corresponding to a stable set of vertices in the graph can be in their critical sections at the same time.

For communication between the processes, we will assume that there are one or more shared variables $V_1, \cdots, V_k$, each of which can be accessed by more than one process, perhaps by all the processes. One thing we are seeking is bounds on the storage necessary for these variables. We denote the size of the set of values assumable by V_k by $|V_k|$; thus if V_k can assume values from 0 to N, $|V_k| = N+1$.

The processes access these shared variables only by specified operations called synchronization primitives (see [1], [2], [5], [6], [9], [11], [14] for some guides to this rather extensive literature). In this paper we will have occasion to use two distinct synchronization primitives: **PV**-chunk [9] and test-and-set [2], [14].

The syntax of the test-and-set operator, given in more detail in [2], may be summarized as follows: a test-and-set operator allows a process to test a shared variable until it reaches a fixed (set of) values and then perform certain actions, including resetting the shared variable to a value, which may be determined by the process using its knowledge of the shared variable value. The statement may be written as follows:

test V until $V = x_1$ or x_2 or $\cdots$ or x_n
then $V :=$ function (V).

If V is not one of the indicated values then the statement is reexecuted from the beginning (busy waiting). As soon as V assumes one of the indicated values, function (V) is computed, V is set to the new value, and control passes to the next statement. (The computation and substitution is an atomic action; that is, if several processes are attempting to access V, only one at a time will actually change the value of V.) Note that $V :=$ function (V) is an acceptable form of the test-and-set statement, since by implication it tests V first and sees that it has any one of its finitely many possible values. In each case, the function in function (V) is an arbitrary programmable function; it may take significant computation time or space. It is this feature that makes the general test-and-set operator both extremely powerful and difficult to implement efficiently.

A **PV-chunk operator** [9] can be implemented as a special case of a test-and-set operator but, being much more restricted, is usually written rather differently. Essentially, it restricts the test to testing for a certain one-sided inequality and the function to incrementing or decrementing by a (freely user-chosen) constant. The syntax we will use is the following:

Test V until $V \geqq c_1$ then $V := V - c_1$.

Note that c_1 can vary from one occurrence of this statement to another. V is initialized to some positive integer at the start and will never become negative; it can be increased by any process by executing the statement with c_1 negative, in which case the test condition is met automatically.

Variations of both test-and-set and **PV**-chunk can be specified in which a "failure" message is returned if the condition is not met, instead of busy waiting until it is met.

An operation very much like **PV**-chunk is available as a system call "SEMOP" in UNIX System V. The operation there increments or decrements a variable (or even several variables) by varying amounts as an atomic operation, provided that the change does not carry the variable(s) below 0 or above $2^{15}-1$. Thus, there is real motivation for finding methods that coordinate large time-shared or networked systems using **PV**-chunk which require a limited range for the shared variable(s). In addition, hardware experimenters, such as those designing the New York University Ultra-computer, have been experimenting with parallel hardware designed to perform N changes on a variable in less than time proportional to N (e.g., time proportional to $\log N$). The operations proposed seem not inconsistent in complexity with **PV**-chunk, but are surely less complex than the general test-and-set. Test-and-set requires that an individual process receive a value from the shared memory, compute, and return a value to shared memory; **PV**-chunk allows a value to be sent to special hardware serving the shared memory, which can do the calculation internally and needs to pass very little back to the calling process (in the model given here, the calling process sleeps until V is large enough, then V is decremented and the process awakens; in an alternative model, a single-bit message "fails" or "succeeds" and may be returned to the caller by the special hardware.

A principal result of [9] is that given a collection of processes and conflicts forming a threshold graph, conflict avoidance (mutual exclusion) can be achieved by using **PV**-chunk operations on a single shared variable with range 0 to t; each process can enter its critical section if and only if it can decrement V by an amount given by the label of its node in the graph. Further, the threshold graphs are precisely the graphs for which **PV**-chunk operations on a single shared variable will achieve conflict avoidance.

The results in [2] assume that all processes in the system are deterministic and impose a technical requirement "No Memory": each process knows nothing about the state of the system other than what is in the shared variable(s). That is, each time a process enters or leaves its critical section when the shared variable(s) have a particular value, it makes the same change in the shared variable(s). We do require this in the proof of Theorem 4.2 below.

Our main result in this section is that managing mutual exclusion in a system where the exclusion graph is a threshold graph, both test-and-set and **PV**-chunk operations require the same amount of shared memory and that is the amount determined by Theorem 3.3.

THEOREM 4.1 [9]. *Let a system of processes have the exclusion graph $G=(V, E)$, which is a threshold graph with separator t. Then there are entry and exit protocols that achieve the desired mutual exclusion by using* **PV**-*chunk operations to access a single shared variable whose range includes the integers* 0 *to* t.

Proof [9]. It suffices to start the shared variable V with the value t. The entry protocol for a process whose corresponding vertex has label a is simply the following:

Test V until $V \geqq a$ then

$$V := V - a$$

and the exit protocol is simply

$$V := V + a$$

(as noted, the prefix "Test V until $V \geqq -a$" would add nothing). It is easy to see that a collection of processes can be in their critical sections at the same time if and only if their corresponding vertex labels total no more than t, i.e., if and only if there are no edges between their corresponding vertices. □

Clearly, the same theorem holds if the term **PV**-chunk is replaced by test-and-set, since **PV**-chunk is a special case of test-and-set. The main result of this section is that the range 0 to t in Theorem 4.1 is the best possible, even if we use several shared variables and the more general test-and-set operations.

THEOREM 4.2. *Let a system of processes have the exclusion graph $G=(V, E)$, which is a threshold graph with separator t as above. If there is a collection of entry and exit protocols that enforces the mutual exclusions in G, using test-and-set on a collection of shared variables $V_1, \cdots, V_k$ with $|V_j| = v_j$, then the number of different sets of values assumable by the V_j (and hence the product of the v_j) must be at least $t+1$.*

Proof. Suppose our synchronizing method stores adequate information in V_1 through V_k so that a process can determine from them if it can enter its critical section, and suppose the set of V_i assume no more than t values. We will obtain a contradiction.

Put G in normal form as in § 3, so that

$$t+1=\prod (d_k+1), \qquad (k=1, \cdots, n).$$

We will select $t+1$ distinct collections R_p of vertices from the union of the D_k, $k=1, \cdots, n$, and arrange them in order such that we have the following:

(a) Any two R_p and R_q differ, but their intersections with each D_k have one containing or equal to the other.

(b) R_p and R_{p+1} differ by only one vertex.

We do this by a process suggestive of Gray codes for numbers of mixed base. First, order each D_k. For each sequence of integers $(a_1, a_2, \cdots, a_n)$ with $0 \leqq a_k \leqq d_k$, select a set consisting of the first a_k elements of D_k for each k. This yields $\prod (d_k+1) = t+1$ sets meeting the conditions of (a). We must now order them to obey condition (b). We do this by starting with the set determined by $(0, \cdots, 0)$, then going to $(1, 0, \cdots, 0)$, $(2, 0, \cdots, 0), \cdots, (d_1, 0, \cdots, 0)$. On "filling" each position change the next position by one and then step "down" through the previous cases: $(d_1, 1, \cdots, 0)$, $(d_1-1, 1, 0, \cdots, 0), \cdots, (1, 1, 0, \cdots, 0)$, $(0, 1, 0, \cdots, 0)$, and then $(0, 2, 0, \cdots, 0)$ and so on. The resulting list includes all $t+1$ sets and consecutive sets differ in just one element.

Start the system G with all processes outside their critical sections. Clearly, the first element of D_1 can enter its critical section; let it do so. Now go through the sequence of starts and stops (entries and exits of critical sections) dictated by the sequence R_p above: each element of D_1 starts (enters its critical section), the first element of D_2 starts, each element of D_1 stops (exits its critical section), and so on. Each step involves one element of a D_i entering or exiting its critical section. At each stage some change may be made in one or more of the V_k. There are $t+1$ stages (starting with no processes in critical sections and going through all the R_p). By hypothesis, the collection of V_k can assume only t distinct values, so there are two stages, R_p and R_q, of the above process when the V_k are in identical states. We must now get from this to a contradiction.

Suppose the set R_p of processes in critical sections is represented by the n-tuple $(a_1, \cdots, a_n)$ and the set R_q by $(b_1, \cdots, b_n)$. These must differ; without loss of generality suppose $b_j < a_j$ and $b_i = a_i$ for $i > j$. Now, one at a time, stop each process (if any) represented by b_{j+1} through b_n and the corresponding process in a_{j+1} through a_n. Note that in each case we make the same process in each group leave its critical section, and the V_i have the same values afterward. Next we stop elements of D_j, one at a time, in both sets of processes (the same element in each) and do this b_j times so that one set has no elements of D_j left in its critical section and the other has one or more still in critical sections; the values of the V_i are still the same. But now we have our contradiction; i.e., a process from C_j can enter its critical section with the remnant of

the set R_q (where now no process in D_j or any later D_k is in its critical section) but cannot do so with the remnant of R_p, despite the fact that the V_k values are identical and no process in C_k can tell one situation from the other. This completes the proof of Theorem 4.2. □

Since this shows that $t+1$ values of shared variables are needed for a protocol using the test-and-set operation, it follows that at least $t+1$ values are needed using the more restrictive operation **PV**-chunk. From the fact that $\prod (d_k+1)$ values are needed we also obtain Corollary 4.3.

COROLLARY 4.3. *The value of t found in Theorem* 3.3 *is the smallest value of t allowing G to be labeled as a threshold graph.*

Proof. If the graph had a threshold labeling with a separator $s<t$, then by Theorem 4.1, mutual exclusion could be managed with a shared variable with $s+1$ distinct values. □

This is the minimality result of Orlin [16], proved in a quite different fashion. We also obtain a proof of a theorem on graph coverings [15].

COROLLARY 4.4. *Let G be a threshold graph and let* $G_1, G_2, \cdots, G_n$ *be subgraphs, which are threshold graphs and whose union covers G (i.e., includes all vertices and edges of G). Let G have separator t and let each* G_k *have separator* t_k. *Then* $\prod (t_k+1) \geqq t+1$.

Proof. The system of processes whose exclusion graph is G can have mutual exclusion enforced by enforcing it within each subgraph G_k. Each process will, in its entry protocol, attempt to decrement shared variables V_1 through V_n by the amount of its associated label in the respective G_k. While this approach is definitely lockout- and deadlock-prone, it does enforce the needed exclusions since every edge of G is in a G_k; hence the shared variables are capable of at least $t+1$ distinct values. (The tendency to deadlock can be overcome by having the set of **PV**-chunk operations be a single atomic action, as is the case in the UNIX System V implementation.)

We have now seen that for threshold graphs, $\prod (d_k+1) = t+1$ is a necessary and sufficient measure of the shared values needed to enforce mutual exclusion, both for **PV**-chunk and for test-and-set.

In the special case of a clique (all nodes connected), we have a threshold graph with (after adjustment for our normal form) one element in D_1, the rest in C_1, $t=1$, and a shared variable with two values is necessary and sufficient. This special case is Theorems 3.1 and 4.4 of [2].

5. Some examples. Section 4 shows that for processes whose conflict graph is a threshold graph, **PV**-chunk and test-and-set operations require similar storage to avoid conflicts. In Example 5.1 we show that when the conflict graph is not a threshold graph, test-and-set may require less storage.

Example 5.1. We consider a system **S** consisting of four processes A, B, C, and D (Fig. 4), such that each of A and C conflict with each of B and D. (This is the case $n=4$ of the famous dining philosophers problem. See, for instance, [10].) We can avoid conflict using test-and-set with one shared variable, values 0 through 3; to avoid

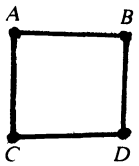


FIG. 4

conflict using **PV**-chunk requires at least two shared variables and four bits to store the shared variables.

We first give a solution for test-and-set. Let A and C each incorporate the entry protocol:

test V until $V=0$ or 1 then $V:=2*V+1$

(i.e., change 0 to 1 or change 1 to 3) and the following exit protocol:

$$V:=(V-1)/2.$$

Let B and D have the following entry protocol:

test V until $V=0$ or 2 then $V:=V/2+2$

(i.e., change 0 to 2 or change 2 to 3) and the following exit protocol:

$$V:=2*(V-2).$$

The effect is that V is 0 if no process is in its critical section (or entry or exit protocols), 1 if A or C is in its critical section, 2 if B or D is, and 3 if both A and C or both B and D are in critical sections. This shows that four possible shared values (one variable, occupying two bits of storage) enable test-and-set to enforce mutual exclusion in this system.

It is harder to show how many values are needed to control this system using **PV**-chunk. We use a method developed more fully in [15]. Suppose there are shared variables $V_1, \cdots, V_n$ which control the system **S**. Each of A, B, C, and D, while executing its entry protocol to enter its critical section, changes one or more of $V_1, \cdots, V_n$; it fails to enter its critical section if some V_k cannot be sufficiently decremented at this time. Denote by a_k through d_k the decrements that A through D, respectively, apply to each V_k and by t_k the maximum permissible value of each V_k. Now for each k, the labels a_k through d_k and maximum t_k induce a graph on the four vertices (possibly with no edges) showing the processes prevented from running at once by that V_k; in [9] it has been shown that each graph induced in this fashion is a threshold graph. The square denoting the system **S** is the union of these graphs. Since the square is not itself a threshold graph, there must be at least two V_k's: **PV**-chunk operations cannot control **S** with just one shared variable.

Here is a simple solution using two shared variables, both initialized to 2 before the processes begin execution. Note that it does enforce the necessary mutual exclusion, is deadlock-free, but is not lockout-free and does not have bounded waiting.

A's Entry Protocol:	Test V_1 until $V_1 \geqq 2$ then $V_1 := V_1 - 2$;
A's Exit Protocol:	$V_1 := V_1 + 2$;
B's Entry Protocol:	Test V_1 until $V_1 \geqq 1$ then $V_1 := V_1 - 1$;
	Test V_2 until $V_2 \geqq 1$ then $V_2 := V_2 - 1$;
B's Exit Protocol:	$V_1 := V_1 + 1$;
	$V_2 := V_2 + 1$;
C's Entry Protocol:	Test V_2 until $V_2 \geqq 2$ then $V_2 := V_2 - 2$;
C's Exit Protocol:	$V_2 := V_2 + 2$;
D's Protocols are the same as B's.	

In fact, the square can be a union of threshold graphs in very few ways: the threshold graphs that are subgraphs of the square are (i) the single edge, which we can take as having $t=1$ and each vertex labeled 1; and (ii) the union of two adjoining edges, which we can take as having the central vertex labeled 2, each end vertex 1,

and $t = 2$. Thus the minimal sets of V_i associated with the square, up to some reasonable notion of equivalence, must be one of (i) V_1 and V_2, each with values 0 to 2, each controlling two edges; (ii) V_1 to V_4, each with values 0 to 1, each controlling one edge; or (iii) one variable with values 0 to 2, controlling two edges, and two variables with values 0 to 1, each controlling one edge. Since a variable with values 0 through 2 requires two bits to store the shared variables, this means at least two shared variables and at least four bits of shared memory are needed to control this system. This completes Example 5.1. Some arguments very similar in philosophy to those in the latter part of this example have been given in [7] and [8].

Example 5.2. Here we illustrate a natural way in which threshold graphs arise in accessing data base management systems. The reader-writer problem is a well-known problem concerning synchronization of accesses to a data base; I was motivated to think about it in this context by [17], and other references may be found therein. This example is an expansion on this problem in the spirit of the example at the start of § 4. In the standard problem, there are in a system several transactions wishing to access the same record: some (readers) want to read data from it without changing it; others (writers) want to change the data (we oversimplify here by not distinguishing update or read-and-write transactions from simple writes). Any number of readers can proceed at once; a writer cannot proceed unless it has exclusive use of the record (a lock on the record), since simultaneous writes might produce nonsense and a read overlapped with a write might return, e.g., a partially changed, not internally consistent, record.

Suppose a database record contains keys with several parts. For example, the fields in a record might be

1: OFFICE# 2: SALESMAN# 3: ACCOUNT# 4: AMOUNT

where the first three are keys, that is, jointly they uniquely identify the customer's record so we can find the amount due from the customer; this amount is stored in field 4, "AMOUNT." It is possible that one or more of the keys, as well as the amount due, might change because of, e.g., the customer moving to a region served by a different office, the salesman being replaced, or two customer firms merging. Thus, for each of the four fields, we can image a transaction to read the record as far as that field, or to write the record from that field onward.

For example, a transaction of type 2R (read fields to 2) would answer a query of the form "Does salesman 3057 serve any accounts at office 103?" and a 3W (write from field 3) would serve a transaction of the form "Open account number 566 for salesman 3057 in office 103." Note that, in fact, these two transactions could proceed at the same time, if it is understood that the second will fail if there is no salesman 3057 in office 103. The second transaction could not proceed simultaneously with a 2W ("delete all records for salesman 3057 in office 103") or a 4R ("what is the balance due from account 566, office 103, salesman 3057?").

Interestingly, a set of transactions consisting of transactions of types 1R, 1W, $\cdots$, 4R, 4W (and similarly for longer sets of multiple keys) has an exclusion graph that is a threshold graph. The transactions of type 1W, $\cdots$, 4W correspond to vertices in the sets $C_1, \cdots, C_4$, respectively, and those of type 1R, $\cdots$, 4R correspond to vertices in the sets $D_1, \cdots, D_4$ in the normal form of a threshold graph. (All writes conflict with each other; no reads conflict with each other; jR conflicts with kW if $j \geqq k$.)

Thus, if we have an upper bound on the number of transactions of each type which might appear in the system at one time, we could manage record locking—including the indicated partial record locking—with **PV**-chunk operations on a single

shared variable. If we knew there would be at most four operations of each of the "read" types, the labels of the corresponding vertices would be 1 for 1R, 5 for 2R, 25 for 3R, 125 for 4R, and the separator would be $t = 624$. Mutual exclusion could be managed using a single shared variable capable of assuming 625 distinct values.

Interestingly, changing the number of writers—the transactions that appear to require the most extensive locks—does not change these numbers; to reduce t we must reduce the number of readers, not the number of writers. This is consistent in a broad sense with the observations in [17] where the added memory to avoid delays is determined by the number of potential readers (there, only one writer is considered). By contrast, changing reader transactions from one subtype to another does change t: if there were at most two transactions of types 1R and 2R, and at most six each of types 3R and 4R, then we would have

$$t = (2+1)(2+1)(6+1)(6+1) - 1 = 440.$$

6. Conclusions and additional problems. Earlier papers such as [2] present a careful analysis of the amount of shared memory required to solve the problem of mutual exclusion when all processes exclude all others. In real applications, it may be possible for some sets of processes, but not others, to enter critical sections simultaneously. A major step in this direction appears in [12], which bounds the delays occurring in the mutual exclusion algorithm by imposing a "locality" condition: particular processes are constrained to share resources only with a limited number of other "nearby" processes. It would be desirable to have efficient solutions, and bounds on possible solutions, for other more general cases. This paper considers systems of asynchronous parallel processes in which the desired mutual exclusions can be modeled by a threshold graph. These differ strongly from the cases concentrated on in [12], since threshold graphs always have a vertex that is adjacent to all other (nonisolated) vertices. In our limited case, simple mutual exclusion (without lockout prevention or other desirable features) can be managed by a single **PV**-chunk operation in the entry protocol preceding the critical section in each process, using a single shared variable with range from 0 to t, with t denoting the minimal separator of the corresponding threshold graph.

Establishing this requires giving a new algorithm for the minimal separator previously calculated by Orlin, and allowing the separator to be written as a product formula in terms of the numbers of vertices in certain classes in the graph (this could also be expressed in terms of vertex degrees or the size of maximal cliques; see also [15]).

For controlling mutual exclusion in this limited case, **PV**-chunk requires no more shared memory than the Lynch–Fischer test-and-set operation. An example suggests that for more general graphs, test-and-set will manage exclusion with fewer shared variables and fewer bits of shared variables than **PV**-chunk. A final example suggests that **PV**-chunk may, in fact, provide an efficient tool for certain kinds of partial-record locking and certain kinds of reader-writer management problems in data base systems.

This leaves a great many open problems. What are efficient ways of managing mutual exclusion situations modeled by more complex graphs than threshold graphs, or indeed, not modeled by graphs at all? (Surely the different form of graph models provided in [12] carry different information than the models here.) Can we find algorithms that incorporate lockout prevention or fairness as well as mutual exclusion, while still using **PV**-chunk operations or other operations that seem easier to implement in efficient hardware than the general test-and-set operation? Does test-and-set solve these other problems with significantly less memory, or significantly faster algorithms, than simpler synchronization primitives? A referee suggests that in some sense Theorem 4.2 does not appear to depend on the synchronization primitive used. The number

$t+1$ is in some sense a measure of the space needed to synchronize the processes represented in the graph, without regard to exact method. Can this be formalized? In Example 5.1, it appears that test-and-set needs less memory than **PV**-chunk; how can this be measured more systematically? Finally, is it connected to Lipton's concept of using one primitive to "simulate" another [11]?

Acknowledgments. The author had numerous helpful discussions with Y. Zalcstein. The referees suggested extensive revisions to improve clarity of the presentation; the remaining awkwardness is due to the author.

REFERENCES

[1] T. BLOOM, *Evaluating synchronization mechanisms*, in Proc. 7th Annual ACM Symposium on Operating Systems Principles, (ACM SIGOPS), Pacific Grove, CA, 1979, pp. 24–32.

[2] J. E. BURNS, P. JACKSON, N. A. LYNCH, M. J. FISCHER, AND G. L. PETERSON, *Data requirements for implementation of N-process mutual exclusion using a single shared variable*, J. Assoc. Comput. Mach., 29 (1982), pp. 183–205.

[3] V. CHVATAL AND P. HAMMER, *Aggregation of inequalities integer programming*, Ann. Disc. Math., 1 (1977), pp. 145–162.

[4] A. CREMERS AND T. HIBBARD, *Mutual exclusion of N processors using an $O(N)$-valued message variable*, in Lecture Notes in Computer Science 62, Springer-Verlag, Berlin, New York, 1978, pp. 165–176.

[5] E. W. DIJKSTRA, *Solution of a problem in concurrent programming control*, Comm. ACM, 8 (1965), p. 569.

[6] ———, *Cooperating sequential processes*, in Programming Languages, F. Genuys, ed., Academic Press, New York, 1968.

[7] K. ECKER AND S. ZAKS, *On a graph labeling problem*, Bericht Nr. 99, Gesellschaft für Mathematik und Datenverarbeitung MBH Bonn, Federal Republic of Germany, 1977.

[8] P. ERDÖS, A. W. GOODMAN, AND L. POSA, *The representation of a graph by set intersections*, Canad. J. Math., 18 (1967), pp. 106–112.

[9] P. B. HENDERSON AND Y. ZALCSTEIN, *A graph-theoretic characterization of the PV-chunk class of synchronization primitives*, SIAM J. Comput., 6 (1977), pp. 88–108.

[10] D. LEHMAN AND M. O. RABIN, *On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem*, in Proc. 8th Annual ACM Symposium on Principles of Programming Languages, 1981, pp. 133–138.

[11] R. LIPTON, L. SNYDER, AND Y. ZALCSTEIN, *A comparative study of models of parallel computation*, in Proc. 15th Annual Symposium on Switching and Automata Theory, New Orleans, LA, 1974, pp. 145–155.

[12] N. A. LYNCH, *Upper bounds for static resource allocation in a distributed system*, J. Computer System Sci., 23 (1981), pp. 254–278.

[13] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[14] N. LYNCH AND M. FISCHER, *On describing the behavior and implementation of distributed systems*, Theoret. Comput. Sci., 13 (1981), pp. 17–43.

[15] E. T. ORDMAN, *Threshold coverings and resource allocation*, in Proc. 16th Southeastern International Conference on Graph Theory, Combinatorics, and Computing, Congr. Numer., 49 (1985), pp. 99–113.

[16] J. ORLIN, *The minimal integral separator of a threshold graph*, Ann. Discrete Math., 1 (1977), pp. 415–419.

[17] G. L. PETERSON, *Concurrent reading while writing*, ACM Trans. Programming Lang. Syst., 1 (1983), pp. 46–55.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 166-185, February 1989

011

ALGORITHMS FOR PACKING SQUARES: A PROBABILISTIC ANALYSIS

E. G. COFFMAN, JR.† AND J. C. LAGARIAS†

Abstract. This paper gives a probabilistic performance analysis of simple algorithms for packing lists L_n of n squares into a strip or a set of bins. It is assumed that the square sizes are drawn independently from the uniform distribution on $[0, 1]$. The strip-packing problem is to pack the squares into a strip of width 1 so as to minimize the height of the packing. Let $\mathrm{OPT}(L_n)$ denote the height of an optimal strip packing for list L_n. A simple $O(n \log n)$ approximation algorithm, called Algorithm A, is described and it is proven that the height $A(L_n)$ of a packing of list L_n satisfies $A(L_n) - \mathrm{OPT}(L_n) \leqq \frac{1}{4}$ with probability $1 - O(e^{-c_0 n})$ for a positive constant c_0. It is also shown that $E[\mathrm{OPT}(L_n)] = 3n/8 + \Theta(n^{1/3})$.

The bin-packing problem is to pack the squares into square bins of size 1 so as to minimize the number of bins. Let $\mathrm{OPT}_B(L_n)$ denote the number of bins in an optimal packing of list L_n. The authors present another $O(n \log n)$ approximation algorithm and show that the number of bins it needs to pack a list L_n is precisely $\mathrm{OPT}_B(L_n)$ with probability $1 - O(e^{-c_1 n})$ for a positive constant c_1. Finally, it is shown that $E[\mathrm{OPT}_B(L_n)] = n/2 + \Theta(1)$. These bounds for packing squares into strips and bins are much tighter than those that have been obtained for packing rectangles.

Key words. packing squares, strip packing, two-dimensional bin packing, asymptotic probabilistic analysis

AMS(MOS) subject classifications. 05B40, 68Q30

1. Introduction.

Probabilistic analysis of packing algorithms has yielded important and unexpected insights into a number of NP-complete packing problems. For example, it has often been shown that the average-case behavior of a simple approximation algorithm is extremely good, even though its worst-case behavior is relatively poor. In this paper, where the two-dimensional problem of packing square objects is considered, we find further instances of such algorithms. The results are distinguished by the fact that, in a strong probabilistic sense, our square-packing algorithms provably obtain packings much closer to optimal than algorithms previously considered for rectangle packing.

Two principal variations of square packing will be studied: *strip packing* and *bin packing* (in two dimensions). In the former, squares no larger than $w \times w$ are to be packed into a semi-infinite strip of width w so as to minimize the height or span of the packing, i.e., the maximum height reached by the top of any square in the packing. As illustrated in Fig. 1, the packing must be orthogonal, i.e., the sides of the squares must be parallel to the sides or the base of the strip, and the squares must not overlap each other or the strip boundaries. The squares are provided in a list $L_n = (X_1, \cdots, X_n)$, where $0 < X_i \leqq w$, $1 \leqq i \leqq n$ denotes the dimensions of the ith square. In what follows X_i refers to the name of the ith square as well as to its size.

We let $\mathrm{ALG}(L_n)$ denote the height of the packing of list L_n produced by algorithm ALG. $\mathrm{OPT}(L_n)$ denotes the height of an optimal packing of L_n.

We adopt the normalization $w = 1$ and consider the situation where $X_1, \cdots, X_n$ are independent uniform random draws from $[0, 1]$. In § 2 we devise an efficient $n \log n$-time algorithm, called Algorithm A, and prove that for n large, $A(L_n) - \mathrm{OPT}(L_n) \leqq \frac{1}{4}$ with very high probability, i.e., with a probability that approaches 1 as $1 - O(e^{-cn})$ for some constant $c > 0$. Our next result, in § 2, is a very strong characterization of the height of optimal packings, and hence of A-packings, viz.,

* Received by the editors October 6, 1986; accepted for publication (in revised form) May 17, 1988.
† Mathematics Research Center, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

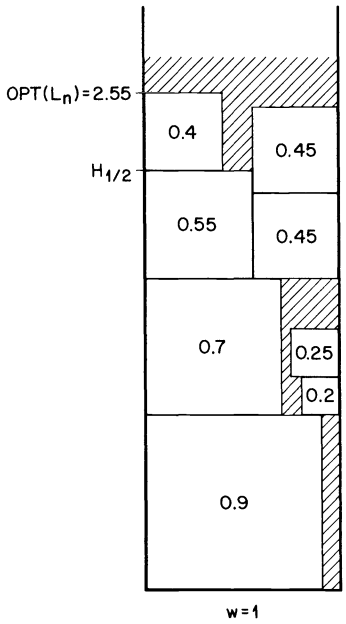


FIG. 1. *An optimal strip packing*; $L_n = (0.9, 0.7, 0.55, 0.45, 0.45, 0.4, 0.25, 0.2)$.

$\mathrm{OPT}(L_n) = 3n/8 + O(n^{1/3})$, and hence $A(L_n) = 3n/8 + O(n^{1/3})$, with very high probability. In addition, we prove the following average-case performance bounds:

$$E[A(L_n)] = E[\mathrm{OPT}(L_n)] + O(1),$$

$$E[\mathrm{OPT}(L_n)] = \frac{3n}{8} + \Theta(n^{1/3}).$$

Now consider the bin-packing version of square packing, and let OPT_B denote an optimal algorithm. Here, the strip of Fig. 1 is partitioned into unit squares called bins, $B_1, B_2, \cdots$, as shown in Fig. 2. Packings are as before but with the added requirement that squares must not overlap the boundaries between bins. The packing height is now measured as the index of the highest occupied bin, i.e., as the number of bins required by the packing of L_n. In § 3 we introduce an $n \log n$-time Algorithm B and prove that $B(L_n) = \mathrm{OPT}_B(L_n)$ with very high probability. In other words, with probability $1 - O(e^{-cn})$ for some $c > 0$, B is optimal. We also prove

$$E[\mathrm{OPT}_B(L_n)] = \frac{n}{2} + \Theta(1),$$

and similarly for $E[B(L_n)]$. Note that $n/2$ is the expected number of squares with sizes larger than $\frac{1}{2}$, and is therefore an obvious lower bound.

We conclude this section with a brief discussion of background results. Packing squares has long been a source of intriguing combinatorial problems. Numerous references to worst-case analysis can be found in [BCCL] and [CGJ]. However, we know of only two probabilistic studies, and these are briefly described below.

A probabilistic analysis of next-fit (NF) strip packing has been worked out by Hofri [Ho] for the general case of rectangles, where each dimension of each rectangle

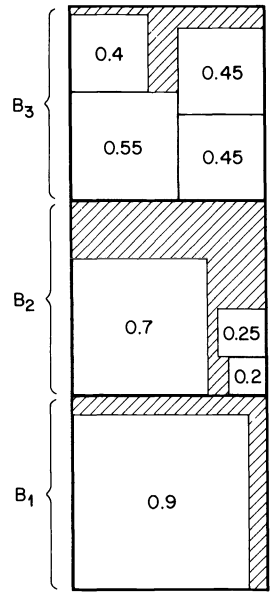


FIG. 2. *An optimal packing of L_n in three bins (see Fig. 1).*

is an independent uniform random draw from $[0, 1]$. According to NF, rectangles $X_1, X_2, \cdots$ are taken from L_n and placed left-justified along the bottom of the strip until a rectangle, say X_i, is encountered that will not fit in the remaining space. A baseline is then extended across the top of the tallest rectangle, and the process is repeated with rectangles from the list $(X_i, \cdots, X_n)$ placed along the new baseline. New baselines are created until all rectangles are packed.

An obvious shortcoming of NF is that a rectangle small enough to be packed on an earlier baseline is not so packed. For the special case of squares this leads to a performance substantially inferior to Algorithm A presented in § 2. Indeed, it is not difficult to show that $E[\mathrm{NF}(L_n)] - E[\mathrm{OPT}(L_n)]$ cannot be bounded by a constant independent of n (see [Ho]).

With the above model of rectangles, Karp, Luby, and Marchetti-Spaccamela [KLM] have devised an efficient algorithm for two-dimensional bin packing based on planar matching. Their results combined with recent results by Leighton and Shor [LS] show that this algorithm produces packings using $n/4 + \Theta(\sqrt{n}\log^{3/4} n)$ bins on the average. They also showed that *no* algorithm could pack L_n in fewer than $n/4 + \Omega(\sqrt{n})$ bins on the average.

2. The strip-packing problem. In this section we first derive a useful lower bound for optimal packings. Then we present Algorithm A and prove that, if the sizes of the squares in L_n satisfy a certain condition, then the height of the A-packing exceeds the lower bound by at most $\frac{1}{2}$. A probabilistic analysis then shows that with very high probability the above condition holds. Finally, we prove that the expected optimal packing height is $3n/8 + \Theta(n^{1/3})$.

In what follows we often simplify the presentation by ignoring the distinction between the notions of "square" and "square size." Thus, for example, when we say "a square in $[a, b]$" we mean the obvious "a square with a size in $[a, b]$."

The height of a packing cannot be less than the total area of the squares (since the strip width is 1), or less than the sum of the square sizes exceeding $\frac{1}{2}$:

$$H_{1/2} = H_{1/2}(L_n) = \sum_{X_i > 1/2} X_i.$$

The average total square area is

$$nEX_i^2 = n \int_0^1 x^2 \, dx = \frac{n}{3},$$

but $EH_{1/2} = n/2 \cdot 3/4 = 3n/8$. Thus,

$$\text{OPT}(L_n) \geqq H_{1/2} \tag{2.1}$$

is a better bound for our purposes.

Note that the above observations imply that no algorithm can waste less than $3n/8 - n/3 = n/24$ space on the average. In this sense, packings of squares drawn uniformly from [0, 1] are not particularly efficient. A more detailed explanation of this property is easy to find. Opposite each square X in $[1-a, 1]$, we can pack $\lfloor (1-a)/a \rfloor$ squares no larger than a without extending beyond the bottom or top of X. However, in our probabilistic model the expected numbers of squares in $[0, a]$ and in $[1-a, 1]$ are equal. Thus, for a moderately smaller than $\frac{1}{2}$ we can expect to have more space than we need for the efficient packing of squares in $[0, a]$, even when we allow for probable variations in the square sizes of L_n. Indeed, we prove later that if all squares in L_n with sizes in $[0, \frac{1}{3}]$ are removed, then with a probability that quickly approaches 1 as $n \to \infty$, the optimal packing height remains unchanged.

Thus, let us concentrate on squares in $(\frac{1}{3}, 1]$; because of their relatively large size it will be easier to characterize the packings of these squares. To obtain a more useful deterministic lower bound based only on the subset of squares in L_n with sizes in $(\frac{1}{3}, 1]$, it is helpful to define the following function. Let $\delta(y)$ denote the total height of squares in $[\frac{1}{2} - y, \frac{1}{2}]$ minus the total height of squares in $(\frac{1}{2}, \frac{1}{2} + y]$, i.e.,

$$\delta(y) = \sum_{1/2 - y \leqq X_i \leqq 1/2} X_i - \sum_{1/2 < X_i \leqq 1/2 + y} X_i, \qquad 0 < y \leqq \frac{1}{2}, \tag{2.2}$$

with $\delta(0) = 0$. We note immediately that if $\delta(y) > 0$ for any y in $(0, \frac{1}{6})$ then the packing of L_n must extend beyond the region occupied by squares in $(\frac{1}{2}, 1]$. This follows from the fact that at most one square in $(\frac{1}{3}, \frac{1}{2}]$ will fit abreast of a square in $(\frac{1}{2}, 1]$. We also note that the squares in $(\frac{1}{3}, \frac{1}{2}]$ not packed with squares in $(\frac{1}{2}, 1]$ can be packed at most two abreast.

To make use of these observations let

$$\Delta = \Delta(L_n) = \max_{0 \leqq y \leqq 1/2} \delta(y). \tag{2.3}$$

If $\Delta > 0$, then we let $X^* \leqq \frac{1}{2}$ denote a largest square size in L_n such that $\Delta = \delta(\frac{1}{2} - X^*)$, i.e.,

$$\Delta = \sum_{X^* \leqq X_i \leqq 1/2} X_i - \sum_{1/2 < X_i \leqq 1 - X^*} X_i; \tag{2.4}$$

but if $\Delta = 0$, then we define $X^* = \frac{1}{2}$, which may not be a square size in L_n.

We have the following deterministic bound.

THEOREM 1. *For any list L_n such that $X^* > \frac{1}{3}$, the optimal packing height satisfies*

$$\text{OPT}(L_n) \geqq H_{1/2} + \frac{\Delta}{2}. \tag{2.5}$$

Proof. If $\Delta = 0$ the result follows from (2.1), so suppose $\Delta > 0$. Consider an optimal strip packing of L_n. We may uniquely partition the packed strip into four disjoint collections R_1, R_2, R_3, R_4 of horizontal slices (or slabs) such that all squares hit by a horizontal cut in R_1 are in $[0, X^*)$; any horizontal cut in R_2 hits a square in $[X^*, \frac{1}{2}]$ but no square in $(\frac{1}{2}, 1]$; any horizontal cut in R_3 hits a square in $(\frac{1}{2}, 1-X^*]$; and any horizontal cut in R_4 hits a square in $(1-X^*, 1]$. We assign to each R_i the total vertical height, height(R_i), of the slices in its collection.

By definition,

$$\text{height}(R_3) + \text{height}(R_4) = H_{1/2} \tag{2.6}$$

and

$$\text{height}(R_3) = \sum_{1/2 < X_i \leqq 1-X^*} X_i. \tag{2.7}$$

It is also obvious from the definitions that the squares in $[X^*, \frac{1}{2}]$ can be packed only in the regions covered by R_2 and R_3. Since $X^* > \frac{1}{3}$, at most two such squares can fit side by side in R_2, and in R_3 a horizontal cut can hit at most 1. Thus,

$$2\,\text{height}(R_2) + \text{height}(R_3) \geqq \sum_{X^* \leqq X_i \leqq 1/2} X_i. \tag{2.8}$$

Subtracting (2.7) from (2.8), then using (2.4), yields

$$2\,\text{height}(R_2) \geqq \Delta.$$

Finally, this inequality and (2.6) yield

$$\begin{aligned} \text{OPT}(L_n) &\geqq \text{height}(R_2) + \text{height}(R_3) + \text{height}(R_4) \\ &\geqq H_{1/2} + \Delta/2. \end{aligned}$$

□

The development leading to Theorem 1 suggests the following algorithm. We say that a sequence of squares is packed *bottom-up* if, when each square is packed, it is placed at the lowest height where it will fit. It is convenient in describing and analyzing the algorithm to use the terms "big square" and "small square" when referring to squares greater than $\frac{1}{2}$ and less than or equal to $\frac{1}{2}$, respectively.

ALGORITHM A.

(1) Stack the big squares of L_n bottom-up in *decreasing* order of size along the left edge of the strip.
(2) Then pack the small squares bottom-up in *increasing* order of size along the right edge of the strip, placing each square as low as it will go, until all are packed. The algorithm then stops if at most one square extends above $H_{1/2}$; otherwise, step (3) is performed.
(3) Starting with the highest small square and working down, successively remove squares from the right edge of the strip and place them in a stack starting at $H_{1/2}$ along the left edge of the strip, until the heights of the two stacks agree to within the size of the top square in one of the two stacks.

Figure 3(a), (b) illustrates corresponding packings just after steps (2) and (3), respectively.

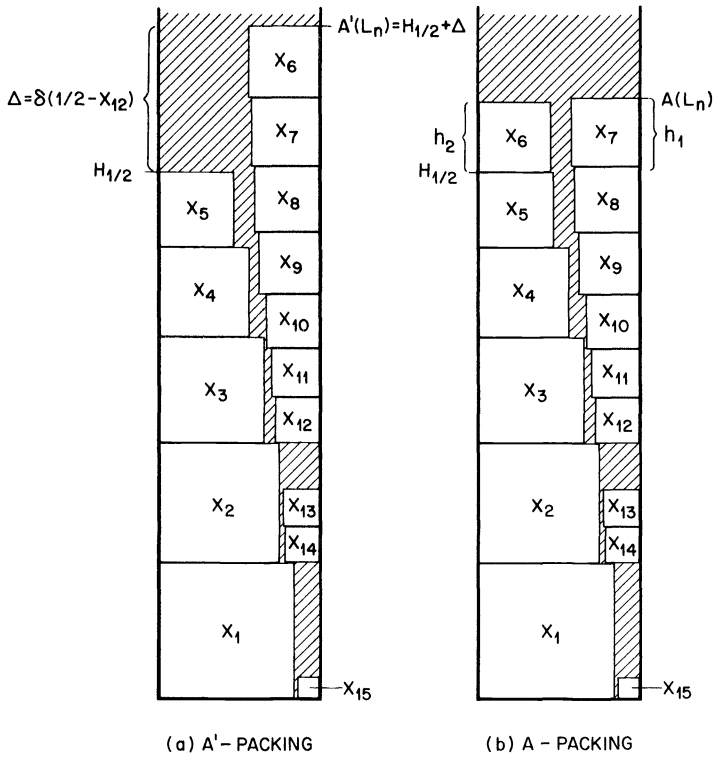


FIG. 3. *Examples for Theorem* 2.

The running time of Algorithm A is clearly dominated by the $O(n \log n)$ time required to sort the squares.

We let Algorithm A′ denote the first two steps of Algorithm A. It is clear that an A'-packing is never better and sometimes worse than an A-packing.

Our next result shows that Algorithm A is close to optimal when $X^* > \frac{1}{3}$.

THEOREM 2. *For all L_n we have*

$$A'(L_n) = H_{1/2} + \Delta \tag{2.9}$$

and

$$A(L_n) \leqq H_{1/2} + \frac{\Delta}{2} + \frac{1}{4}. \tag{2.10}$$

Remarks. (1) As illustrated in Fig. 3(a), the relation (2.9) gives a geometric interpretation of the quantity Δ.

(2) The bound (2.10) is tight. For example, consider any list L_n where n is odd and all squares are of size $\frac{1}{2}$. It is easy to see that $A(L_n) = \mathrm{OPT}(L_n) = (n+1)/4$. Also, $H_{1/2} = 0$ and $\Delta = n/2$, so $A(L_n) = \Delta/2 + \frac{1}{4} = (n+1)/4$. Similar examples for n even are obtained by adding a single big square to the above lists.

Proof. The main difficulty is (2.9); once (2.9) is proved, an easy analysis of step (3) of Algorithm A will then yield (2.10).

To prove (2.9) it is convenient to consider another algorithm, Algorithm A″, which packs the big squares as in step (1) and then packs the small squares in *decreasing* order, placing the largest small square along the right edge of the strip with its top

edge at height $H_{1/2}+\Delta$, and then successively placing each small square along the right edge of the strip with its top edge flush against the square above it (see Fig. 4, which corresponds to Fig. 3). We claim that Algorithm A″ actually produces a packing, i.e., that all of the small squares can be packed in this way without ever encountering a small square that does not fit. To establish the claim, we may suppose without loss of generality that the list L_n is in decreasing order:

$$X_1 \geqq X_2 \geqq \cdots \geqq X_k \geqq \tfrac{1}{2} > X_{k+1} \geqq X_{k+2} \geqq \cdots \geqq X_n.$$

When the small square X_{k+m} is to be packed, its bottom edge will be at height

$$h_m = H_{1/2}+\Delta-\sum_{j=1}^{m} X_{k+j}.$$

In order for X_{k+m} to fit, it is necessary and sufficient that the big square X_l placed opposite it during step (1) (the big square intersected by the line $y = h_m$) satisfy

$$X_l \leqq 1 - X_{k+m}.$$

Next, note that the set of big squares with sizes at most $1-X_{k+m}$ are placed by step (1) at vertical heights that occupy the interval $[h_m^*, H_{1/2}]$, where

$$h_m^* = H_{1/2} - \sum_{1/2 < X_i \leqq 1-X_{k+m}} X_i.$$

An equivalent necessary and sufficient condition for square X_{k+m} to fit is

$$h_m^* \leqq h_m. \tag{2.11}$$

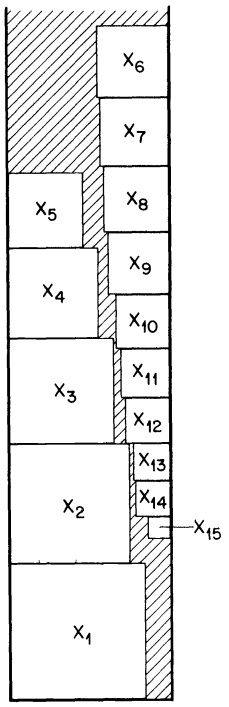


FIG. 4. *The A″-packing for Fig. 3.*

Now by definition of Δ we have

$$\begin{aligned}\Delta &\geqq \delta\left(\frac{1}{2}-X_{k+m}\right)\\ &= \sum_{X_{k+m}\leqq X_i\leqq 1/2} X_i - \sum_{1/2<X_i\leqq 1-X_{k+m}} X_i\\ &\geqq \sum_{j=1}^{m} X_{k+j} - \sum_{1/2<X_i\leqq 1-X_{k+m}} X_i\\ &= (H_{1/2}+\Delta-h_m)-(H_{1/2}-h_m^*)\\ &= h_m^*-h_m+\Delta.\end{aligned}$$

This inequality implies (2.11), which proves the claim.

It is clear that

$$A''(L_n) = H_{1/2}+\Delta.$$

The A'-packing is obtained by successively sliding the small squares packed along the right-hand side of the strip down as low as they will go, proceeding from the smallest to the largest (e.g., Fig. 3(a) is produced from Fig. 4). Hence

$$A'(L_n) \leqq H_{1/2}+\Delta.$$

To establish (2.9) it suffices to show that equality holds here. We do this by contradiction. Suppose equality did not hold; then the right side of the A''-packing in step (2) could be shifted down vertically by a positive amount α, and we would still have a packing. Then the lowest edge of each square X_{k+m} would be placed at height $\tilde{h}_m = h_m - \alpha$ in the new packing. However, by definition of Δ there is some value m_0 for which

$$\Delta = \delta\left(\frac{1}{2}-X_{k+m_0}\right).$$

Then we obtain, as in the proof above,

$$\begin{aligned}\Delta &= \delta(\tfrac{1}{2}-X_{k+m_0})\\ &= h_{m_0}^*-\tilde{h}_{m_0}+\Delta-\alpha,\end{aligned}$$

which shows that

$$\tilde{h}_{m_0} < h_{m_0}^*.$$

This violates (2.11), contradicting the assumption that we had a packing. So (2.9) is proved.

To prove (2.10), we first observe that if at most one square extends beyond $H_{1/2}$ in the A'-packing of L_n, then $A(L_n) = A'(L_n) = H_{1/2}+\Delta$ and $\Delta \leqq \frac{1}{2}$. But then

$$A(L_n) = H_{1/2}+\Delta \leqq H_{1/2}+\frac{\Delta}{2}+\frac{1}{4},$$

so it remains to consider cases where step (3) is performed. Let h_1 and h_2 be the heights of the higher and lower stacks, respectively, as measured from $H_{1/2}$ in the A-packing (see Fig. 3(b)). By (2.9) we have easily $h_1+h_2 = \Delta$, and by the definition of step (3), we have $h_1-h_2 \leqq \frac{1}{2}$. These two relations imply that $h_1 \leqq \Delta/2+\frac{1}{4}$, and hence that

$$A(L_n) = H_{1/2}+h_1 \leqq H_{1/2}+\frac{\Delta}{2}+\frac{1}{4}. \qquad \square$$

It is easy to verify that, if at most one square extends above $H_{1/2}$ in an A-packing, then $\mathrm{OPT}(L_n) = A(L_n) = A'(L_n) = H_{1/2}+\Delta$. A proof of optimality for this special case can be patterned after the proof of Theorem 1.

So far the analysis has been deterministic, with Theorems 1 and 2 combining to show that if $X^* > \frac{1}{3}$ then

$$\mathrm{OPT}(L_n) \leqq A(L_n) \leqq \mathrm{OPT}(L_n) + \tfrac{1}{4}.$$

We now begin a probabilistic analysis under the assumption that the squares are drawn independently from the uniform distribution on $[0, 1]$. Our objectives are estimates of the *probable* performance of the algorithms. As our first objective we prove Theorem 3.

THEOREM 3. *There is a constant $c > 0$ such that* $\Pr\{X^* \leqq 1/3\} = O(e^{-cn})$, *and consequently*

$$\Pr\{A(L_n) - \mathrm{OPT}(L_n) \leqq \tfrac{1}{4}\} = 1 - O(e^{-cn}).$$

Furthermore,

$$E[A(L_n)] = E[\mathrm{OPT}(L_n)] + O(1).$$

Our second objective is a proof of the following bound.

THEOREM 4. *There is a constant $c_1 > 0$ such that for all $x \geqq 1$,*

$$\Pr\{\Delta(L_n) > xn^{1/3}\} \leqq c_1 e^{-\sqrt{x}/3}.$$

Moreover,

$$E[\mathrm{OPT}(L_n)] = \frac{3n}{8} + \Theta(n^{1/3}).$$

Before getting into the proofs of these results we need a more detailed discussion of the function $\delta(y)$ and the tools needed for our probabilistic arguments.

We view $\delta(y)$ as a random process on $[0, \frac{1}{2}]$. To better understand this process, it is helpful to represent it as shown in Fig. 5. The points (square sizes) selected in

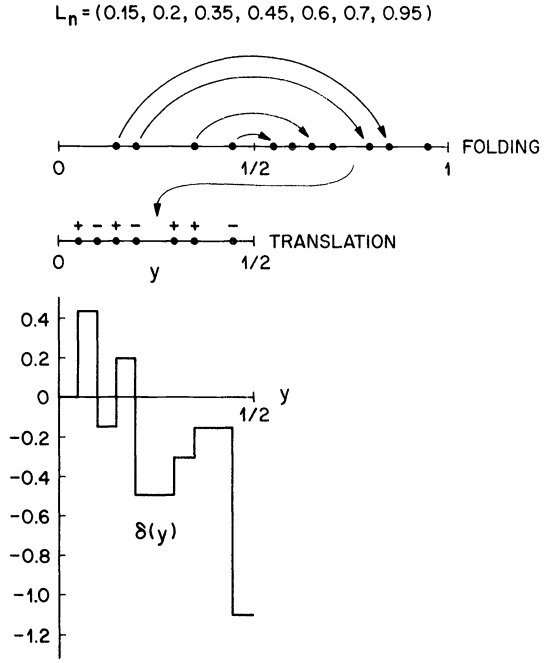


FIG. 5. *The function $\delta(y)$.*

$[0, \frac{1}{2})$ are reflected about the midpoint $\frac{1}{2}$ onto the interval $[\frac{1}{2}, 1]$. These "events" are labeled with a plus, while those falling originally in $(\frac{1}{2}, 1]$ are labeled minus. The pluses and minuses are then mapped by a simple translation onto $[0, \frac{1}{2}]$. Sample functions for $\{\delta(y), 0 < y \leqq \frac{1}{2}\}$ are step functions as illustrated in Fig. 5. A square $X \in [0, \frac{1}{2}]$ becomes a plus at $y = \frac{1}{2} - X$ and creates a positive step of size $X \leqq \frac{1}{2}$ in $\delta(y)$. A square $X \in (\frac{1}{2}, 1]$ becomes a minus at $y = X - \frac{1}{2}$ and creates a negative step of size $X \geqq \frac{1}{2}$ in $\delta(y)$. The event locations comprise n independent uniform random draws from the interval $0 \leqq y \leqq \frac{1}{2}$ in Fig. 5 (i.e., the mappings $y = \frac{1}{2} - X$ for $X \leqq \frac{1}{2}$ and $y = X - \frac{1}{2}$ for $X > \frac{1}{2}$ produce uniform random draws from $[0, \frac{1}{2}]$, since X is a uniform random draw from $[0, 1]$). The sign of each event is equally likely to be + or −, independent of its location.

In the analysis of $\delta(y)$ we need bounds on the tails of the distribution of a sum of bounded, independently and identically distributed random variables. We use the following bound of Bernstein (see [Be] for a proof).

LEMMA 1 (Bernstein). *Let S_n denote the sum of n independent samples of a bounded random variable X, and let $\sigma^2 = \operatorname{Var}[S_n] = n \operatorname{Var}[X]$ and $M = \max |X - E[X]|$. Then for any $t \geqq 0$*

$$\Pr\{S_n - E[S_n] > t\sigma\} < \exp\left\{ -t^2 \Big/ \left(2 + \frac{2}{3}\frac{Mt}{\sigma}\right)\right\}. \tag{2.12}$$

We will apply this lemma to several different random variables, each representing a weighted count of the independently and identically distributed random variables X_i on $[0, 1]$. For example, with y fixed, we define the random variable $N(y)$ counting the number of squares in $[\frac{1}{2} - y, \frac{1}{2} + y]$ by $N(y) = \sum_{i=1}^{n} Y_i$, where

$$Y_i = \begin{cases} 1 & \text{if } X_i \in [\frac{1}{2} - y, \frac{1}{2} + y], \\ 0 & \text{otherwise.} \end{cases}$$

Similarly, for a fixed y the process $\delta(y)$ is a random variable, which can be expressed as $\delta(y) = \sum_{i=1}^{n} Z_i$, where

$$Z_i = \begin{cases} X_i & \text{if } X_i \in [\frac{1}{2} - y, \frac{1}{2}], \\ -X_i & \text{if } X_i \in (\frac{1}{2}, \frac{1}{2} + y], \\ 0 & \text{otherwise.} \end{cases} \tag{2.13}$$

A routine calculation yields

$$E[Z_i] = -y^2, \qquad E[\delta(y)] = -y^2 n, \tag{2.14}$$

$$\operatorname{Var}[Z_i] = \frac{y}{2} + \frac{2}{3}y^3 - y^4, \qquad \operatorname{Var}[\delta(y)] = \left(\frac{y}{2} + \frac{2}{3}y^3 - y^4\right)n, \tag{2.15}$$

$$\max |Z_i - E[Z_i]| = \frac{1}{2} + y(1 - y) \leqq \frac{3}{4}. \tag{2.16}$$

As our last item of preparation we introduce a standard inequality to be used several times. In some probability space suppose the occurrence of event e implies that at least one of the events $e_1, \cdots, e_k$ must occur. Then by Boole's inequality,

$$\Pr\{e\} \leqq \sum_{i=1}^{k} \Pr\{e_i\}. \tag{2.17}$$

We are now in position to prove Theorems 3 and 4.

Proof of Theorem 3. To prove that $X^* > \frac{1}{3}$ with probability $1 - O(e^{-cn})$, for some $c > 0$, it is obviously sufficient to prove that $\delta(y) \leqq 0$ for all $\frac{1}{6} \leqq y \leqq \frac{1}{2}$, with probability $1 - O(e^{-cn})$. For this purpose we begin by considering those samples L_n that satisfy

$$\delta(\tfrac{1}{6}) < -\alpha n, \qquad N(\tfrac{1}{6}) \leqq \beta n \tag{2.18}$$

for any fixed choices of α, β satisfying $0 < \alpha < \frac{1}{36}$, $\frac{2}{3} < \beta < 1$. (We verify later that these inequalities hold jointly with very high probability.) For any such list we divide up the at least $n(1-\beta)$ events with $\frac{1}{6} \leqq y \leqq \frac{1}{2}$ into $[1/2\alpha]$ blocks containing either k or $k+1$ events, where

$$k = \left\lfloor \frac{n - N(\frac{1}{6})}{\lfloor 1/2\alpha \rfloor} \right\rfloor.$$

With no significant loss in generality, assume that $1/2\alpha$ is an integer and consider $n > 1/\alpha$. Then $0 \leqq N(\frac{1}{6}) \leqq \beta n$ implies that $2\alpha(1-\beta)n - 1 \leqq k \leqq 2\alpha n \leqq 3\alpha n - 1$.

Let δ_j denote the value of $\delta(y)$ just after the last event of block j, $1 \leqq j \leqq 1/2\alpha$, and let $\delta_0 = \delta(\frac{1}{6})$. The sizes of positive steps are at most $\frac{1}{3}$ and are decreasing as y varies from $\frac{1}{6}$ to $\frac{1}{2}$. Therefore, no block of at most $3\alpha n - 1$ events starting at $\delta_{j-1} < -\alpha n$ can include a zero crossing, where $\delta(y)$ would become positive. Thus, for $\delta(y)$ to become positive in $\frac{1}{6} \leqq y \leqq \frac{1}{2}$ there must be at least one $j \geqq 1$ such that $\delta_{j-1} < -\alpha n$ and $\delta_j \geqq -\alpha n$. But at any y in $[\frac{1}{6}, \frac{1}{2}]$ the magnitude $(\frac{1}{2}+y)$ of a negative jump is at least twice the magnitude $(\frac{1}{2}-y)$ of a positive jump. Hence for any $j \geqq 1$, the final value δ_j can be greater than the starting value δ_{j-1} only if the number of pluses, N_j^+, was greater than twice the number of minuses, N_j^-, in the jth block. Hence, if $\delta(\frac{1}{6}) < -\alpha n$ and $N_j^+ \leqq 2N_j^-$ for all $j \geqq 1$, then $\delta(y) \leqq 0$ for $\frac{1}{6} \leqq y \leqq \frac{1}{2}$.

If $p_{\alpha\beta}$ denotes the probability that an arbitrary L_n satisfies both inequalities in (2.18), then by the above argument and (2.17) we have

$$\begin{aligned} \Pr\{X^* \leqq \tfrac{1}{3}\} &\leqq \Pr\{\delta(y) \leqq 0, \tfrac{1}{6} \leqq y \leqq \tfrac{1}{2}\} \\ &\leqq (1 - p_{\alpha\beta}) + \sum_{j=1}^{1/2\alpha} \Pr\{N_j^+ > 2N_j^-\}, \end{aligned}$$

and

$$\Pr\left\{X^* \leqq \frac{1}{3}\right\} \leqq (1 - p_{\alpha\beta}) + \frac{1}{2\alpha} \max_j \Pr\{N_j^+ > 2N_j^-\}. \tag{2.19}$$

To bound the first term in (2.19) we use Lemma 1. Consider first the inequality $\delta(\frac{1}{6}) < -\alpha n$ in (2.18). Since $E[\delta(\frac{1}{6})] = -n/36$ (see (2.14)), we write

$$\Pr\left\{\delta\left(\frac{1}{6}\right) > -\alpha n\right\} = \Pr\left\{\delta\left(\frac{1}{6}\right) + \frac{n}{36} > \left(\frac{1}{36} - \alpha\right)n\right\},$$

so that, in the notation of Lemma 1 with $S_n = \delta(\frac{1}{6})$, we can put $t\sigma = (1/36 - \alpha)n$. Then by (2.15), t/σ is a positive constant and t^2 increases linearly with n for any $0 < \alpha < 1/36$. Since M is bounded as in (2.16), we conclude from Lemma 1 that for any $0 < \alpha < 1/36$, there exists a constant $c_1 > 0$ such that

$$\Pr\{\delta(\tfrac{1}{6}) > -\alpha n\} < e^{-c_1 n}. \tag{2.20}$$

For the second inequality in (2.18) the random variable $N(\frac{1}{6})$ is also governed by Lemma 1 with $E[N(\frac{1}{6})] = n/3$, $M \leqq 1$, and $\sigma^2 = \mathrm{Var}[N(\frac{1}{6})] = (2/9)n$. Thus, for any

$\frac{2}{3} < \beta < 1$, we have after substitution in (2.12)

$$\Pr\left\{N\left(\frac{1}{6}\right) > \beta n\right\} \leqq \Pr\left\{N\left(\frac{1}{6}\right) - \frac{n}{3} > \left(\beta - \frac{1}{3}\right)n\right\}$$
$$\leqq \Pr\left\{N\left(\frac{1}{6}\right) - \frac{n}{3} > \frac{n}{3}\right\}$$
$$\leqq e^{-n/6}.$$

It is easy to check that the c_1 in (2.20) must be less than $\frac{1}{6}$ for all $0 < \alpha < 1/36$. Consequently, for any α and β in $(0, 1/36)$ and $(\frac{2}{3}, 1)$, respectively, we have

$$1 - p_{\alpha\beta} = O(e^{-c_1 n}), \tag{2.21}$$

where $c_1 > 0$ is the constant determined by α in (2.20).

For the second term in (2.19), we observe that there will be more than twice as many pluses as minuses in a block if the number of pluses exceeds $\frac{2}{3}$ the total of k or $k+1$ signs. Our asymptotic results are independent of whether we choose k or $k+1$, so for simplicity we assume a total of exactly k signs. Then the distribution of the number of pluses is the binomial distribution for k tosses of a fair coin, i.e., $N_j^+ = \sum_{i=1}^{k} W_i$, where $W_i = 1$ if $X_i \geqq \frac{1}{2}$ and $W_i = 0$ otherwise. By Lemma 1 with $E[N_j^+] = k/2$, $M \leqq 1$, and $\sigma^2 = \mathrm{Var}[N_j^+] = k/4$, we have for all $j \geqq 1$

$$\Pr\{N_j^+ > 2N_j^-\} = \Pr\left\{N_j^+ - \frac{k}{2} > \frac{k}{6}\right\} \leqq e^{-k/22},$$

whereupon $k \geqq 2\alpha(1-\beta)n - 1$ implies that there exists a $c_2 > 0$ such that

$$\Pr\{N_j^+ > 2N_j^-\} \leqq e^{-c_2 n}, \qquad j \geqq 1. \tag{2.22}$$

Substituting (2.21) and (2.22) into (2.19), we obtain as desired

$$\Pr\{X^* \leqq \tfrac{1}{3}\} = O(e^{-cn}),$$

where $c = \min\{c_1, c_2\} > 0$.

The expected-height result is easy. We need only use the trivial bound $A(L_n) \leqq n$ in order to write

$$E[A(L_n) - \mathrm{OPT}(L_n)] \leqq \tfrac{1}{2}[1 - O(e^{-cn})] + O(n\, e^{-cn})$$
$$= O(1). \qquad \square$$

Proof of Theorem 4. To find an upper bound on $\Pr\{\Delta > xn^{1/3}\}$, we use an approach analogous to the proof of Theorem 3. For simplicity we assume that $n^{1/3}$ is an even integer; we leave to the interested reader the trivial modifications of the arguments below that account for values of n other than cubes of even integers. We partition the interval $0 \leqq y \leqq \frac{1}{2}$ in Fig. 5 into $\frac{1}{2}n^{1/3}$ subintervals of length $n^{-1/3}$. The initial value of $\delta(y)$ in the subinterval $[jn^{-1/3}, (j+1)n^{-1/3}]$, $0 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$, is denoted by

$$\delta_j = \delta(jn^{-1/3}) = \sum_{1/2 - jn^{-1/3} \leqq X_i \leqq 1/2} X_i - \sum_{1/2 < X_i \leqq 1/2 + jn^{-1/3}} X_i,$$

and the maximum value is denoted by

$$\Delta_j = \max\{\delta(y); jn^{-1/3} \leqq y \leqq (j+1)n^{-1/3}\}.$$

Clearly,

$$\Delta = \max\{\delta(y); 0 \leqq y \leqq \tfrac{1}{2}\} = \max\{\Delta_j; 0 \leqq j \leqq \tfrac{1}{2}n^{1/3} - 1\},$$

so by (2.17) we can write

$$\Pr\{\Delta > xn^{1/3}\} \leqq \sum_{j=0}^{n^{1/3}/2-1} \Pr\{\Delta_j > xn^{1/3}\}. \tag{2.23}$$

Our next objective is an upper bound on $\Pr\{\Delta_j > xn^{1/3}\}$ for any $x \geqq 1$. First, we define $N_j(y)$ as the number of pluses in $[jn^{-1/3}, y]$ less the number of minuses in $[jn^{-1/3}, y]$, and let

$$\hat{N}_j = \max\{N_j(y); jn^{-1/3} \leqq y \leqq (j+1)n^{-1/3}\}.$$

Next we observe that, for any fixed constant z_j, the maximum value satisfies $\Delta_j > xn^{1/3}$ only if either the initial value satisfies $\delta_j > (x - z_j)n^{1/3}$ or the value of the function has a net increase exceeding $z_j n^{1/3}$ in some initial subinterval, i.e., $\delta(y) - \delta_j > z_j n^{1/3}$ for some $jn^{-1/3} \leqq y \leqq (j+1)n^{-1/3}$. Since the sizes of all positive jumps in δ are at most the size of any negative jump, the latter condition can occur only if in some initial subinterval $[jn^{-1/3}, y]$, the number of pluses exceeds the number of minuses by more than $z_j n^{1/3}$, i.e., only if $\hat{N}_j > z_j n^{1/3}$. Therefore, by (2.17),

$$\Pr\{\Delta_j > xn^{1/3}\} \leqq \Pr\{\delta_j > (x - z_j)n^{1/3}\} + \Pr\{\hat{N}_j > z_j n^{1/3}\}, \tag{2.24}$$

for $0 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$. We will bound the right-hand side of this inequality, making the choice

$$z_j = \frac{x + j^2}{2}.$$

We start with the first of the probabilities on the right-hand side of (2.24). For $j = 0$ and any $x \geqq 0$ we have

$$\Pr\{\delta_0 > (x - z_0)n^{1/3}\} = \Pr\{\delta_0 > \tfrac{1}{2}xn^{1/3}\} = 0. \tag{2.25}$$

For the case $1 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$, $x \geqq 1$, we now prove the bound

$$\Pr\{\delta_j > (x - z_j)n^{1/3}\} < e^{-(x+j^2)/3j}. \tag{2.26}$$

We obtain (2.26) from Lemma 1 using the definition of $\delta(y)$, with $y = jn^{-1/3}$, as a sum of independently and identically distributed random variables Z_i, $1 \leqq i \leqq n$. By (2.14), (2.15), and $j \leqq \frac{1}{2}n^{1/3} - 1$, we have

$$E[\delta_j] = -j^2 n^{1/3},$$

$$\operatorname{Var}[\delta_j] = \tfrac{1}{2}jn^{2/3} + j^3[\tfrac{2}{3} - jn^{-1/3}] < \tfrac{2}{3}jn^{2/3}.$$

We apply Lemma 1 with $t\sigma = z_j n^{1/3} = ((x + j^2)/2)n^{1/3}$, $\sigma^2 < \frac{2}{3}jn^{2/3}$, and $M \leqq \frac{3}{4}$ (by (2.16)) to conclude that

$$\begin{aligned} \Pr\{\delta_j > (x - z_j)n^{1/3}\} &= \Pr\{\delta_j + j^2 n^{1/3} > z_j n^{1/3}\} \\ &< \exp\left\{ -\frac{(t\sigma)^2}{2\sigma^2 + \frac{2}{3}Mt\sigma} \right\} \\ &< \exp\left\{ -\frac{z_j^2 n^{2/3}}{\frac{4}{3}jn^{2/3} + \frac{1}{2}z_j n^{1/3}} \right\}. \end{aligned} \tag{2.27}$$

Since $1 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$ and $x \geqq 1$ imply that $z_j \geqq 1$ and $jn^{1/3} \geqq 4$, the magnitude of the exponent is bounded by

$$z_j^2 n^{2/3} \Big/ \left(\frac{4}{3}jn^{2/3} + \frac{z_j n^{1/3}}{2} \right) = \frac{z_j}{j}\left\{ 1 \Big/ \left(\frac{4}{3z_j} + \frac{1}{2jn^{1/3}} \right) \right\} \geqq \frac{2}{3}\frac{z_j}{j}.$$

Substituting this bound into (2.27) and then setting $z_j = (x + j^2)/2$ yields (2.26).

Next we turn to an estimate of the second probability on the right-hand side of (2.24). For given j and for $1 \leqq i \leqq n$ define the independently and identically distributed random variables

$$V_i = \begin{cases} +1 & \text{if } X_i \in [\tfrac{1}{2} - (j+1)n^{-1/3}, \tfrac{1}{2} - jn^{-1/3}], \\ -1 & \text{if } X_i \in [\tfrac{1}{2} + jn^{-1/3}, \tfrac{1}{2} + (j+1)n^{-1/3}], \\ 0 & \text{otherwise,} \end{cases} \tag{2.28}$$

and note that

$$N_j \equiv N_j((j+1)n^{-1/3}) = \sum_{i=1}^{n} V_i.$$

We claim that

$$\Pr\{\hat{N}_j > z_j n^{1/3}\} \leqq 2 \Pr\{N_j > z_j n^{1/3}\}. \tag{2.29}$$

To prove the claim we consider the conditional joint probability distribution of the V_i's given that exactly k of them are nonzero, i.e., $k = \sum_{i=1}^{n} |V_i|$. For each sample drawn from this conditional distribution, reorder the k nonzero V_i's in order of increasing value of the parameter

$$U_i = \begin{cases} \tfrac{1}{2} - X_i & \text{if } 0 < X_i \leqq \tfrac{1}{2}, \\ X_i - \tfrac{1}{2} & \text{if } \tfrac{1}{2} < X_i < 1, \end{cases}$$

to obtain an ordered sequence $\{W_l: 1 \leqq l \leqq k\}$ with each $W_l = \pm 1$. The joint probability distribution of the W_l's induced from that of the V_i's is easily seen to be that of a set of Bernoulli trials (coin flips). (Indeed the joint probability distribution of the X_i's is invariant under any subset of the reflections $X_i \to 1 - X_i$ for $1 \leqq i \leqq N$; each of these leaves all U_i unchanged but changes the sign of V_i, which shows that all sign patterns of $\{W_l: 1 \leqq l \leqq k\}$ are equally likely.) The quantities $\hat{N}_j$ and N_j are expressed in terms of the random variables W_l by

$$\hat{N}_j = \max_{1 \leqq i \leqq k} \sum_{l=1}^{i} W_l, \qquad N_j = \sum_{l=1}^{k} W_l.$$

A well-known result for Bernoulli trials [Fe1, Thm. 1, p. 88], proved using the reflection principle, is that for all w,

$$\Pr\left\{\max_{1 \leqq i \leqq k} \sum_{l=1}^{i} W_l > w\right\} \leqq 2 \Pr\left\{\sum_{l=1}^{k} W_l > w\right\}.$$

Choosing $w = z_j n^{1/3}$, this gives

$$\Pr\left\{\hat{N}_j > z_j n^{1/3} \,\middle|\, \sum_{i=1}^{n} |V_i| = k\right\} \leqq 2 \Pr\left\{N_j > z_j n^{1/3} \,\middle|\, \sum_{i=1}^{n} |V_i| = k\right\}.$$

Hence, removing the conditioning, we obtain (2.29) as claimed.

Now we bound $\Pr\{N_j > z_j n^{1/3}\}$ using Lemma 1 applied to the random variables V_i, $1 \leqq i \leqq n$. In this case $E[V_i] = 0$, $\sigma^2 = 2n^{2/3}$, and $M = 1$, so if we set $t\sigma = z_j n^{1/3}$ we obtain

$$\begin{aligned} \Pr\{N_j > z_j n^{1/3}\} &< \exp\left\{-\frac{(t\sigma)^2}{2\sigma^2 + \frac{2}{3}Mt\sigma}\right\} \\ &< \exp\left\{-\frac{z_j^2 n^{2/3}}{4n^{2/3} + \frac{2}{3}z_j n^{1/3}}\right\}. \end{aligned} \tag{2.30}$$

Since $0 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$, $x \geqq 1$, and hence $z_j \geqq \frac{1}{2}$, the magnitude of the exponent in (2.30) is bounded by

$$\frac{z_j^2 n^{2/3}}{4n^{2/3} + \frac{2}{3}z_j n^{1/3}} = z_j \Big/ \left(\frac{4}{z_j} + \frac{2}{3n^{1/3}}\right) \geqq \frac{2}{17} z_j = \frac{x + j^2}{17}.$$

Thus, (2.30) implies the simpler bound

$$\Pr\{N_j > z_j n^{1/3}\} < e^{-(x+j^2)/17} \tag{2.31}$$

for $x \geqq 1$, $0 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$. Combining (2.31) with (2.29), we obtain

$$\Pr\{\hat{N}_j > z_j n^{1/3}\} \leqq 2e^{-(x+j^2)/17}, \tag{2.32}$$

for $x \geqq 1$, $0 \leqq j \leqq \frac{1}{2}n^{1/3} - 1$.

Returning to (2.24), we substitute (2.25), (2.26), and (2.32) to obtain

$$\Pr\{\Delta_0 > xn^{1/3}\} \leqq 2e^{-x/17},$$

$$\Pr\{\Delta_j > xn^{1/3}\} \leqq 2e^{-(x+j^2)/17} + e^{-(x+j^2)/3j}, \qquad j \geqq 1,$$

and hence by (2.23)

$$\Pr\{\Delta > xn^{1/3}\} \leqq 2e^{-x/17} + \sum_{j=1}^{\infty} (2e^{-(x+j^2)/17} + e^{-(x+j^2)/3j}).$$

Clearly, $e^{-(x+j^2)/17} > e^{-(x+j^2)/3j}$ for $1 \leqq j \leqq 5$, whereas the opposite inequality holds for $j \geqq 6$. Hence,

$$\Pr\{\Delta > xn^{1/3}\} < 17e^{-x/17} + 3\sum_{j \geqq 6} e^{-(x+j^2)/3j}.$$

It is easy to verify that $(x+j^2)/3j$ is decreasing for $1 \leqq j \leqq \sqrt{x}$ and increasing for $j > \sqrt{x}$. Then using $(x+j^2)/3j \geqq 2\sqrt{x}/3$, $1 \leqq j \leqq \sqrt{x}$, and $(x+j^2)/3j \geqq j/3$, $j > \sqrt{x}$, we obtain

$$\begin{aligned}\Pr\{\Delta > xn^{1/3}\} &< 17e^{-x/17} + 3\sqrt{x}e^{-2\sqrt{x}/3} + 3\sum_{j > \sqrt{x}} e^{-j/3} \\ &< 17e^{-x/17} + 3\sqrt{x}e^{-2\sqrt{x}/3} + 11e^{-\sqrt{x}/3}.\end{aligned}$$

At this point it is easy to see that there exists a $c_1 > 0$ such that for all $x \geqq 1$,

$$\Pr\{\Delta > xn^{1/3}\} \leqq c_1 e^{-\sqrt{x}/3}. \tag{2.33}$$

It remains to prove the expected value result. Applying Theorems 1 and 2 and the fact that $\Pr\{X^* \leqq \frac{1}{3}\} = O(e^{-cn})$, we have

$$\frac{3n}{8} + \frac{1}{2}E[\Delta] \leqq E[\mathrm{OPT}(L_n)] \leqq \frac{3n}{8} + \frac{1}{2}E[\Delta] + O(1).$$

Hence, it suffices to prove

$$E[\Delta] = \Theta(n^{1/3}). \tag{2.34}$$

To prove the upper bound implied by (2.34) we write

$$E[\Delta] = n^{1/3} \int_0^{\infty} \Pr\{\Delta > xn^{1/3}\}\, dx.$$

Then $E[\Delta] = O(n^{1/3})$ follows directly from (2.33) and $\int_0^{\infty} e^{-\sqrt{x}/3}\, dx = O(1)$.

To prove the lower bound implied by (2.34) we show that there exists a constant $\gamma > 0$ such that

$$E[\Delta] \geqq \gamma n^{1/3} \tag{2.35}$$

for all n sufficiently large. Since $\Delta > \delta(y)$ for any fixed y, it suffices for each n to select a suitable value y_n and show that for all n sufficiently large,

$$\Pr\{\delta(y_n) > n^{1/3}\} > \gamma, \tag{2.36}$$

since (2.35) easily follows from this.

We choose $y_n = n^{-1/3}$, so that from (2.14) and (2.15)

$$E[\delta(y_n)] = -n^{1/3}, \qquad \operatorname{Var}[\delta(y_n)] = \frac{n^{2/3}}{2} + \frac{2}{3} - \frac{1}{n^{1/3}}.$$

The normalized sums

$$\hat{\delta}(y_n) = [\delta(y_n) - E[\delta(y_n)]]/\sqrt{\operatorname{Var}[\delta(y_n)]},$$

have mean zero and variance 1, and it is easy to see that for all $n \geqq 1$ if $\delta(y_n) \leqq n^{1/3}$ then $\hat{\delta}(y_n) \leqq 5$, and hence

$$\Pr\{\delta(y_n) > n^{1/3}\} \geqq \Pr\{\hat{\delta}(y_n) > 5\}. \tag{2.37}$$

Thus to prove (2.36) it suffices to show that

$$\Pr\{\hat{\delta}(y_n) \geqq 5\} \geqq \gamma$$

for a positive γ independent of n.

Now limit theorems on sums of independently and identically distributed random variables do not apply directly to $\{\hat{\delta}(y_n), n \geqq 1\}$. To get a corresponding limit theorem for this sequence we consider n as fixed and examine the random variables

$$\delta_m = \delta_{m,n} = \sum_{i=1}^{m} Z_i,$$

where

$$Z_i = \begin{cases} X_i & \text{if } X_i \in [\frac{1}{2} - n^{-1/3}, \frac{1}{2}], \\ -X_i & \text{if } X_i \in [\frac{1}{2}, \frac{1}{2} + n^{-1/3}], \\ 0 & \text{otherwise.} \end{cases} \tag{2.38}$$

The choice $m = n$ gives $\delta_{m,n} = \delta(y_n)$. There are classical limit theorems with error terms that apply to δ_m as m varies. One such result, the Berry-Esséen theorem [Fe2, p. 515], asserts that for any distribution of the Z_i's, which for some constant ρ satisfies

$$E[|Z_i - E[Z_i]|^3] < \rho, \qquad i \geqq 1,$$

we have for the normalized sums $\hat{\delta}_m = (\delta_m - E[\delta_m])/\sqrt{\operatorname{Var}[\delta_m]}$ that for all x and all m

$$|\Pr\{\hat{\delta}_m \leqq x\} - \Phi(x)| \leqq \frac{33}{4} \frac{\rho}{\sigma^3 \sqrt{m}}, \tag{2.39}$$

where $\Phi(x)$ is the normal distribution function with zero mean and unit variance, and where

$$\sigma^2 = \operatorname{Var}[Z_i - E[Z_i]] = \operatorname{Var}[Z_i].$$

For the random variables in (2.38) we have $E[Z_i] = n^{-2/3}$. Then $|Z_i - E[Z_i]| < \frac{1}{2} + n^{-1/3}$ when $Z_i = X_i$ or $Z_i = -X_i$, and $|Z_i - E[Z_i]| = n^{-2/3}$ when $Z_i = 0$. Hence,

$$\begin{aligned} E[|Z_i - E[Z_i]|^3] &\leqq [\tfrac{1}{2} + n^{-1/3}]^3 \cdot 2n^{-1/3} + n^{-2}[1 - 2n^{-1/3}] \\ &= O(n^{-1/3}). \end{aligned}$$

Using (2.15) we also obtain

$$\sigma^2 = \mathrm{Var}\,[Z_i - E[Z_i]] = \tfrac{1}{2}n^{-1/3} + O(n^{-1}).$$

Choosing $m = n$ in (2.39), we have

$$\hat{\delta}_{n,n} = \hat{\delta}(y_n) = \hat{\delta}(n^{-1/3}),$$

and

$$|\Pr\{\hat{\delta}(n^{-1/3}) \leqq x\} - \Phi(x)| = O\left(\frac{n^{-1/3}}{(n^{-1/3})^{3/2}n^{1/2}}\right)$$
$$= O(n^{-1/3}).$$

Thus,

$$\Pr\{\hat{\delta}(n^{-1/3}) > 5\} = 1 - \Phi(5) - O(n^{-1/3}),$$

so that for any $\gamma < 1 - \Phi(5)$, $\Pr\{\delta(n^{-1/3}) > n^{1/3}\} > \gamma$ for all n sufficiently large, and hence $E[\Delta] \geqq \gamma n^{1/3}$. □

3. Two-dimensional bin packing. The methods of the preceding section can be modified to prove similar results for packing squares into bins (unit squares). Since the analysis is rather simpler, we content ourselves here with stating the results obtained and briefly pointing out the basic observations used in their proofs. We use the subscript B to distinguish quantities in this section from the cognate quantities of § 2.

In the bin-packing problem it is the *number* of bins that is to be minimized. Consequently, the relevant quantities in the analysis turn out to be the numbers of squares within given ranges, rather than their cumulative heights. Specifically, the appropriate analogue $\delta_B(y)$ of the basic quantity $\delta(y)$, studied in § 2, is defined as

$$\delta_B(y) = \sum_{i=1}^{n} Y_i, \qquad 0 < y \leqq \frac{1}{2}, \tag{3.1}$$

where $Y_i = Y_i(y)$ is defined in terms of independently and identically distributed uniform random variables on $[0, 1]$ by

$$Y_i = \begin{cases} 1 & X_i \in [\frac{1}{2} - y, \frac{1}{2}], \\ -3 & X_i \in (\frac{1}{2}, \frac{1}{2} + y], \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

In other words, $\delta_B(y)$ counts the number of squares in $[\frac{1}{2} - y, \frac{1}{2}]$ minus three times the number in $(\frac{1}{2}, \frac{1}{2} + y]$. The factor of three comes from the fact that for all $0 \leqq y < \frac{1}{2}$ at least three squares of size $\frac{1}{2} - y$ can be packed into a bin along with a square of size $\frac{1}{2} + y$. As before, $\delta_B(0) = 0$, $\Delta_B = \max_{0 \leqq y \leqq 1/2} \delta_B(y)$, and X_B^* is a maximum square size such that $\Delta_B = \delta_B(\frac{1}{2} - X_B^*)$.

Our initial lower bound is now

$$\mathrm{OPT}_B(L_n) \geqq N_{1/2}, \tag{3.3}$$

where $N_{1/2}$ is the number of squares in L_n with sizes in $(\frac{1}{2}, 1]$, and has the expected value $EN_{1/2} = n/2$. The following more useful bound corresponds to Theorem 1 and is based on the simple observation that at most four of the squares in $(\frac{1}{3}, \frac{1}{2}]$ can be packed in a single bin.

THEOREM 5. *For any L_n such that $X_B^* > \frac{1}{3}$ there holds*

$$\mathrm{OPT}_B(L_n) \geqq N_{1/2} + \left\lceil \frac{\Delta_B}{4} \right\rceil.$$

Next, we define an algorithm similar to Algorithm A.

ALGORITHM B.

(1) Pack the squares $X_i \in (\frac{1}{2}, 1]$ bottom-up in *descending* order of size in bins $B_1, \cdots, B_{N_{1/2}}$, one per bin and against the left edge of the bin.

(2) Pack the remaining squares bottom-up in *ascending* order, placing each as far to the left as possible in the first (lowest indexed) bin where it will fit.

As in Theorem 2 it is easily verified that the squares in $[0, X_B^*)$ are all packed by Algorithm B in bins $B_1, \cdots, B_{N_{1/2}}$. Also, if $X_B^* > \frac{1}{3}$ then the squares in $[X_B^*, \frac{1}{2}]$ will be packed four per bin, except possibly for the last bin. We obtain Theorem 6.

THEOREM 6. *For any L_n there holds*

$$B(L_n) \leqq N_{1/2} + \left\lceil \frac{\Delta_B}{4} \right\rceil.$$

It follows from Theorems 5 and 6 that if $X_B^* > \frac{1}{3}$ then Algorithm B produces an optimal packing. The object of the probabilistic analysis is then to show that $X_B^* > \frac{1}{3}$ with very high probability. As before, this entails an analysis of $\delta_B(y)$, for which a calculation yields

$$E[\delta_B(y)] = -2yn, \tag{3.4}$$

$$\mathrm{Var}[\delta_B(y)] = (10y - 4y^2)n. \tag{3.5}$$

The following lemma applies a result in [Fe2, Example (c), p. 393] and provides the basis of the analysis.

LEMMA 2. *Let Z_i, $i = 1, 2, \cdots$ be independently and identically distributed random variables, each being $+1$ or -3 with equal probability. There exists a $C > 0$ such that*

$$\Pr\left\{ \max_{m \geqq 1} \sum_{i=1}^{m} Z_i > x \right\} \sim Ce^{-\alpha x} \quad \text{as } x \to \infty,$$

where $\alpha \approx 0.61$ is the unique root of $E[e^{\alpha Z_i}] = \frac{1}{2}[e^{\alpha} + e^{-3\alpha}] = 1$.

Let Z_i, $1 \leqq i \leqq n - N(\frac{1}{6})$, denote the jumps in $\delta_B(y)$ in the interval $\frac{1}{6} \leqq y \leqq \frac{1}{2}$, indexed in order of increasing y. Now $0 \leqq X_B^* \leqq \frac{1}{3}$ implies that $\delta_B(y) > 0$ for some $\frac{1}{6} < y \leqq \frac{1}{2}$, which in turn implies that either $\delta_B(\frac{1}{6}) > -n/6$ or

$$\max_{1 \leqq m \leqq n - N(1/6)} \sum_{i=1}^{m} Z_i > \frac{n}{6},$$

or both, whence

$$\Pr\left\{0 \leqq X_B^* \leqq \frac{1}{3}\right\} \leqq \Pr\left\{\delta\left(\frac{1}{6}\right) > -\frac{n}{6}\right\} + \Pr\left\{\max_{1 \leqq m \leqq n} \sum_{i=1}^{m} Z_i > \frac{n}{6}\right\}.$$

Using (3.4) and (3.5), we apply Lemma 1 to the first probability on the right, and then apply Lemma 2 to the second probability on the right to obtain Theorem 7.

THEOREM 7. *There is a constant $c > 0$ such that*

$$\Pr\{0 \leqq X_B^* \leqq \tfrac{1}{3}\} = O(e^{-cn}),$$

and hence

$$\Pr\{B(L_n) = \mathrm{OPT}_B(L_n)\} = 1 - O(e^{-cn}).$$

Finally, we can develop an analogue to Theorem 4 that describes the expected performance of an optimal algorithm. For this we again analyze the random variables $\delta_B(y)$ and Δ_B. In particular, let us consider a lower bound analysis such as that at the end of the proof of Theorem 4.

As before $E[\delta_B(y)] < 0$, and hence in order that $E[\Delta_B] > 0$ we must have for some y a positive probability of positive swings in $\delta_B(y)$ that are proportional to the absolute value of the mean. By the normal limit law, if the probability of such deviations is to remain bounded away from zero as $n \to \infty$, we need a standard deviation for each n that is proportional to the absolute value of the mean. In the strip-packing case this was obtained by the choice $y_n = n^{-1/3}$, where in absolute value the mean and standard deviation became $\Theta(n^{1/3})$ (see (2.14) and (2.15)). In the present case, an inspection of (3.4) and (3.5) shows that this requirement leads to $y_n = O(1/n)$. But for y_n proportional to $1/n$ the mean and standard deviation become constants. Thus, there is a constant $c > 0$ such that $E[\Delta_B] > c$ for all n sufficiently large.

Obtaining an upper bound is fairly easy. An application of Lemma 2 leads to an upper bound on the tails of Δ_B and shows that $E[\Delta_B]$ is bounded. The following result is obtained.

THEOREM 8. *For all n sufficiently large*

$$\Pr\{\Delta_B > x\} = O(e^{-\alpha x})$$

with α as defined in Lemma 2. Moreover,

$$E[\mathrm{OPT}_B(L_n)] = \frac{n}{2} + \Theta(1).$$

4. Final remarks. The performance of square-packing algorithms is sensitive to the probability distribution of the sizes of squares to be packed. How well Algorithms A and B perform for size distributions other than the uniform distribution on [0, 1] is an interesting question.

Roughly speaking, Algorithms A and B produce near optimum packings for any distribution that provides a sufficiently large number of squares greater than $\frac{1}{2}$, and allows us to effectively ignore small squares (e.g., squares smaller than $\frac{1}{3}$ in strip packing). As one example, Algorithms A and B perform well for distributions that are strictly positive on [0, 1] and symmetric about $\frac{1}{2}$. As a second example, consider a distribution uniform over $[0, \theta]$, for some θ. Here, we claim that, except for the adjustment of certain constants, our strip-packing analysis on a strip of width 1 applies to any θ large enough to ensure that the expected total height of squares greater than $\frac{1}{2}$ is greater than the expected total height of squares smaller than $\frac{1}{2}$. A simple calculation shows that this holds for $\theta > \sqrt{2}/2$ and that

$$E[H_{1/2}(L_n)] = \left(\frac{\theta^2 - \frac{1}{4}}{2\theta}\right) n$$

in this case. For the bin-packing case the corresponding requirement is that θ be large enough to ensure that the expected number of squares larger than $\frac{1}{2}$ is greater than one third the expected number of squares smaller than $\frac{1}{2}$, and a simple calculation shows that this holds for $\theta > \frac{2}{3}$. The analysis of previous sections can be adapted to cover such distributions.

The methods of this paper do not apply to probability distributions producing a preponderance of small squares. Indeed, in that situation the approximation algorithms analyzed in this paper will not perform well. It is an interesting problem to devise approximation algorithms that work well for such probability distributions.

Acknowledgments. We are indebted to George Lueker and Peter Shor for helpful comments and for many corrections to the original version of this paper.

REFERENCES

[BCCL] B. S. BAKER, A. R. CALDERBANK, E. G. COFFMAN, JR., AND J. C. LAGARIAS, *Approximation algorithms for maximizing the number of squares packed in a rectangle*, SIAM J. Algebraic Discrete Methods, 4 (1983), pp. 383-397.

[Be] G. BENNETT, *Probability inequalities for the sum of independent random variables*, Amer. Statist. Assoc. J., (1962), pp. 33-45.

[CGJ] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Approximation algorithms for bin-packing—an updated survey*, in Algorithm Design for Computer System Design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, Berlin, New York, 1984, pp. 49-106.

[Fe1] W. FELLER, *An Introduction to Probability Theory and Its Applications*, Vol. I, Third edition, John Wiley, New York, 1968.

[Fe2] ———, *An Introduction to Probability Theory and Its Applications*, Vol. II, John Wiley, New York, 1966.

[Ho] M. HOFRI, *Two dimensional packing: Expected performance of simple level algorithms*, Inform. and Control, 45 (1980), pp. 1-17.

[KLM] R. M. KARP, M. LUBY, AND A. MARCHETTI-SPACCAMELA, *Probabilistic analysis of multi-dimensional bin-packing problems*, in Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 289-298.

[LS] F. T. LEIGHTON AND P. W. SHOR, *Tight bounds for maximum upward-right matching*, Mathematics Department, Massachusetts Institute of Technology, Cambridge, MA, to appear.

SIAM J. COMPUT.
Vol. 18, No. 1, pp. 186-208, February 1989

012

THE KNOWLEDGE COMPLEXITY OF INTERACTIVE PROOF SYSTEMS*

SHAFI GOLDWASSER†, SILVIO MICALI†, AND CHARLES RACKOFF‡

Abstract. Usually, a proof of a theorem contains more knowledge than the mere fact that the theorem is true. For instance, to prove that a graph is Hamiltonian it suffices to exhibit a Hamiltonian tour in it; however, this seems to contain more knowledge than the single bit Hamiltonian/non-Hamiltonian.

In this paper a computational complexity theory of the "knowledge" contained in a proof is developed. Zero-knowledge proofs are defined as those proofs that convey no additional knowledge other than the correctness of the proposition in question. Examples of zero-knowledge proof systems are given for the languages of quadratic residuosity and quadratic nonresiduosity. These are the first examples of zero-knowledge proofs for languages not known to be efficiently recognizable.

Key words. cryptography, zero knowledge, interactive proofs, quadratic residues

AMS(MOS) subject classifications. 68Q15, 94A60

1. Introduction. It is often regarded that saying a language L is in NP (that is, acceptable in nondeterministic polynomial time) is equivalent to saying that there is a polynomial time "proof system" for L. The proof system we have in mind is one where on input x, a "prover" creates a string α, and the "verifier" then computes on x and α in time polynomial in the length of the binary representation of x to check that x is indeed in L. It is reasonable to ask if there is a more general, and perhaps more natural, notion of a polynomial time proof system. This paper proposes one such notion.

We will still allow the verifier only polynomial time and the prover arbitrary computing power, but will now allow both parties to flip unbiased coins. The result is a probabilistic version of NP, where a small probability of error is allowed. However, to obtain what appears to be the full generality of this idea, we must also allow the prover and verifier to *interact* (i.e., to talk back and forth) and to *keep secret* their coin tosses. We call these proof systems "interactive proof systems." This notion is formally defined in § 2, where we also define what it means for a language to have an interactive proof system.

It is far from clear how to use this power of interaction. Languages with nondeterministic polynomial time algorithms or with probabilistic polynomial time algorithms have proof systems with little or no interaction. We would therefore like examples of languages that appear to have neither nondeterministic nor probabilistic polynomial time algorithms, and yet have interactive proof systems. Although we do not present any such examples here, there are now examples in the literature. Using ideas from an initial version of this paper [GMR] Goldreich, Micali, and Wigderson [GMW] have shown that the "graph nonisomorphism" language has an interactive

* Received by the editors August 26, 1985; accepted for publication (in revised form) April 18, 1988. A preliminary version of this paper appeared in the Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 174–187.

Editor's Note. This paper was originally scheduled to appear in the February 1988 Special Issue on Cryptography (SIAM J. Comput., 17 (1988)).

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The work of these authors was supported by National Science Foundation grants DCR-84-13577 and DCR-85-09905.

‡ Computer Science Department, University of Toronto, Toronto, ONT M5S 1A4 Canada. The work of this author was supported by the Natural Sciences and Engineering Research Council of Canada under grant A3611.

proof system. Independently of this paper, Babai and Szemeredi [BS] show that certain matrix group problems have what they call "Arthur-Merlin" proof systems, which immediately implies that they have interactive proof systems. In fact, the notion of an Arthur-Merlin proof system consists of a restricted interactive proof system in which the prover sees the coin flips of the verifier. Nevertheless, Goldwasser and Sipser [GS] have shown that a language has an interactive proof system if and only if it has an Arthur-Merlin proof system.

It appears, however, that our notion of interactive proof systems generalizes in the right way to attack a novel problem. The main purpose of this current paper, in fact, is to use interactive proof systems to investigate a natural question; how much knowledge is transmitted to the verifier in an interactive proof system for L? For example, consider SAT, the NP-complete language of satisfiable sentences of the propositional calculus. In the obvious proof system, to prove $F \in \text{SAT}$, the prover gives a satisfying assignment I for the formula F, which the verifier then checks in polynomial time. This assignment gives the verifier much more knowledge than merely the fact that $F \in \text{SAT}$; it also gives a satisfying assignment. At the other extreme, every language that can be accepted in probabilistic polynomial time has a proof system in which the prover does nothing, and hence gives no knowledge to the verifier.

We say an interactive proof system for L is *zero-knowledge* if for each $x \in L$, the prover tells the verifier essentially nothing, other than that $x \in L$; this should be the case even if the verifier chooses not to follow the proof system but instead tries (in polynomial time) to trick the prover into revealing something. The notion of zero-knowledge is formally defined in § 3. This definition is an important contribution of this paper.

The main technical contributions of this paper are the proofs in §§ 5 and 6 that the languages QR and QNR (defined below) both have zero-knowledge interactive proof systems. These are the first zero-knowledge protocols demonstrated for languages not known to be recognizable in probabilistic polynomial time. To understand the languages QR and QNR, it helps to read the (brief) number theory background given in § 4. However, for now, let x, y be integers, $0 < y < x$, such that $\gcd(x, y) = 1$; we say that y is a *quadratic residue* mod x if $y = z^2 \bmod x$ for some z; if not, we say that y is a *quadratic nonresidue* mod x. We define

$$\text{QR} = \{(x, y) | y \text{ is a quadratic residue mod } x\}, \quad \text{and}$$

$$\text{QNR} = \{(x, y) | y \text{ is a quadratic nonresidue mod } x\}.$$

(Actually, QNR will be defined slightly differently in § 4.) Both QR and QNR are in NP, and thus possess an elementary proof system. For instance, to prove membership in QNR, the prover just sends x's factorization. But looking ahead to zero-knowledge proof systems, let us discuss a more interesting example of a proof system for QNR.

Example 1. Let us call the prover A and the verifier B. Say that the input is (x, y). Let $n = |x|$, where $|x|$ is the length of the binary representation of x. We will now describe (omitting some details) an interactive proof system for QNR. B begins by flipping coins to obtain random bits $b_1, b_2, \cdots, b_n$. B then flips additional coins to obtain a string α, from which B computes $z_1, z_2, \cdots, z_n$ such that each z_i is a random z, $0 < z < x$, $\gcd(x, z) = 1$. B then computes $w_1, w_2, \cdots, w_n$ as follows: for each i, if $b_i = 0$ then $w_i = z_i^2 \bmod x$; if $b_i = 1$ then $w_i = (z_i^2 y) \bmod x$. B then sends $w_1, w_2, \cdots, w_n$ to A. For each i, A computes (somehow) whether or not w_i is a quadratic residue mod x, and sends B a sequence of bits $c_1, c_2, \cdots, c_n$, where $c_i = 0$ if and only if w_i is a quadratic residue mod x. B checks if $b_i = c_i$ for every i, and if so is "convinced" that $(x, y) \in \text{QNR}$.

It is not hard to see that if $(x, y) \in \text{QNR}$ and both parties follow the protocol, then B will become "convinced." On the other hand, if y is a quadratic residue mod x, then so is each w_i, and every value for w_i is equally likely; since A does not see the sequence $\{b_i\}$, the probability that every $c_i = b_i$ (and hence that B will be convinced) is 2^{-n}. So this is an interactive proof system for QNR. Let us now address the question of how much knowledge A may release.

It is an interesting question how zero-knowledge should be defined. If the prover is trying to prove to the verifier that y is a quadratic residue mod x, then certainly the verifier should not be able to trick the prover into revealing a square root of y mod x, or the factorization of x, or any information which would help the verifier to compute these things much faster than before. In fact, the prover should not reveal anything which would help the verifier compute *anything* much faster than before. The way to state this formally seems to be that what the verifier sees in the protocol (even if he cheats) should be something which the verifier could have computed for himself, merely from the fact that $(x, y) \in \text{QNR}$. Of course, what the verifier sees in the protocol is really a probability distribution. Thus, zero-knowledge means that one can compute in polynomial time, from $(x, y) \in \text{QNR}$, without a prover, the same (or almost the same) probability distribution that the verifier would see with the prover. This is defined formally in § 3. Here, let us informally discuss whether the above interactive proof system for QNR is zero-knowledge.

Consider a pair $(x, y) \in \text{QNR}$, and say that A follows the protocol. Can B obtain any additional knowledge? For the moment, assume that B follows the protocol. B "sees" $[\{b_i\}, \alpha, \{w_i\}, \{c_i\}]$ distributed according to a particular distribution. Without any help from a prover, we can quickly generate a random string according to this distribution: just choose $\{b_i\}$ and α randomly, and then compute $\{w_i\}$ from them; then compute $c_i = b_i$ for each i.

Now what if B were to cheat? B could begin by setting $w_1 = 42$, and then behave correctly. Consider now the induced distribution on $[\{b_i\}, \alpha, \{w_i\}, \{c_i\}]$; in order to compute a random member of it (without help from a prover), we must compute whether or not 42 is a quadratic residue x, given x and a quadratic nonresidue y. At this time it is not known how to compute this in polynomial time, so this proof system may not be zero-knowledge.

A zero-knowledge proof system for QNR is given in § 6. The ideas of this proof system partially come from the secret exchanging protocol of Luby, Micali, and Rackoff [LMR] and are useful there as well. They have also proved useful in the oblivious transfer protocol of Fischer, Micali, Rackoff, and Witenberg [FMRW] and for the identification scheme of Feige, Fiat, and Shamir [FFS]. These ideas have also helped Goldreich, Micali, and Wigderson [GMW] to show that the languages of "graph isomorphism" and "graph nonisomorphism" have zero-knowledge interactive proof systems.

Although we find the idea of a zero-knowledge interactive proof system fascinating in itself, the main motivation for it and the main applications of it are in the area of cryptographic protocols. For example, in the secret exchanging protocol in [LMR], one person wishes to exchange a secret with another without giving away any additional knowledge. Ideas similar to those here must be used to even define this concept.

More generally, however, it often arises at some point in a protocol that A wishes to convince B of some fact. Say that we know that the protocol would be secure if at this point an angel or someone B trusted were to tell B (truthfully) if A is telling the truth. We want the notion of zero-knowledge to be such that an appropriate zero-knowledge interactive proof system could be inserted at this point instead of the trusted

party, and the whole protocol would remain secure. (Of course, A would have to possess some additional information enabling her to implement the part of the prover efficiently.) In particular instances, we can prove that such substitution works, but a general framework for discussing protocols must exist before the general theorem can even be stated. However, based on intuition and experience, the authors (and many others who have studied these ideas since their initial appearance) believe that the definition of zero-knowledge proposed here has the required properties.

The most important development since these results first appeared is the proof by Goldreich, Micali, and Wigderson [GMW] that, subject to a common complexity theory assumption, every language in NP has a zero-knowledge interactive proof system. These proof systems for NP languages appear to have applications in just about every protocol problem. It is almost certain that these results will vastly simplify distributed cryptographic protocol design in the future, as demonstrated by the powerful results of [GMW2].

2. Interactive proof systems. Intuitively, what should we require from an efficient theorem-proving procedure?

(1) That it should be possible to "prove" a true theorem.
(2) That it should not be possible to "prove" a false theorem.
(3) That communicating the "proof" should be *efficient.* Namely, regardless of how much time it takes to come up with the proof, its correctness should be efficiently verified.

The NP formalization of the concept of an efficient proof system captures one way of communicating a proof. In this section, we will generalize the NP proof system to capture a more general way of communicating a proof. The verifier will be a probabilistic polynomial time (in the length of the common input) machine that is able to exchange messages (strings) with the prover.

At the same time that we introduce probability into the proof system, we relax the classical notion of a "proof." Our verifier may erroneously be convinced of the truth of a proposition with a very small probability of error (less than n^{-k} for each positive constant k and all sufficiently large input-sizes n).

We proceed to formally define the new system.

2.1. Interactive Turing machines and protocols.

DEFINITION. An *interactive Turing machine* (ITM) is a Turing machine equipped with a read-only input tape, a work tape, a random tape, one read-only communication tape, and one write-only communication tape. The random tape contains an infinite sequence of random bits, and can be scanned only from left to right. We say that an interactive machine *flips a coin,* meaning that it reads the next bit in its own random tape.

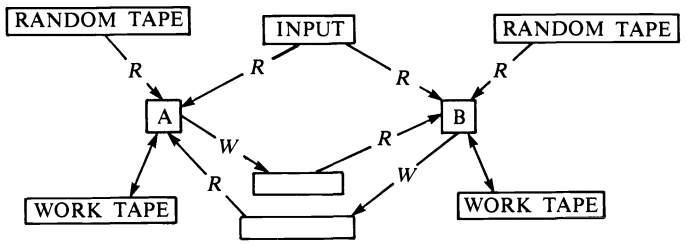


FIG. 1. *An interactive protocol. — denotes a read/write head, R a read-only head, W a write-only head.*

DEFINITION. An *interactive protocol* is an ordered pair of ITM's A and B such that A and B share the same input tape, B's write-only communication tape is A's read-only communication tape and vice versa. Machine A is not computationally bounded, while machine B's computation time is bounded by a polynomial in the length of the common input. The two machines take turns in being active, with B being active first. During an active stage machine $A(B)$ first performs some internal computation using its input tape, work tapes, communication tape and random tape; and, second, it writes a string (for $B(A)$) on its write-only communication tape. The ith message of $A(B)$ is the entire string that $A(B)$ writes on its communication tape during its ith active stage. As soon as machine $A(B)$ writes its message, it is deactivated and machine $B(A)$ becomes active, unless the protocol has been terminated. Either machine can terminate the computation of the protocol by not sending any message in an active stage. Machine B accepts (or rejects) the input by outputting *accept* (or *reject*) and terminating the protocol. The *computation time* of machine B is the sum of the B's computation time during its active stages, and it is this time that is bounded by a polynomial in the length of the input, denoted $|x|$.

2.2. Interactive proof systems.

DEFINITION. Let L be a language over $\{0, 1)^*$. Let (A, B) be an interactive protocol. We say that (A, B) is an *interactive proof system* for L if we have the following:

(1) For each k, for sufficiently large x in L given as input to (A, B), B halts and accepts with probability at least $1-|x|^{-k}$. (The probabilities here are taken over the coin tosses of A and B.)

(2) For each k, for sufficiently large x not in L, for any ITM A', on input x to (A', B), B accepts with probability at most $|x|^{-k}$. (The probabilities here are taken over the coin tosses of A' and B.)

Remark 1. The above probability of error can be decreased, say to smaller than $2^{-|x|}$, by the standard technique of repeating the protocol many times and choosing to accept by majority vote.

We now argue that this definition captures what we intuitively want from an efficient proof system. Condition (1) essentially says that, if $x \in L$, B will accept with overwhelming probability. Condition (2) says that, if x is not in L, there exists no strategy that succeeds with nonnegligible probability for convincing B to accept. In fact, B needs not to trust (or know the program of) the machine with which it is interacting. It is enough for B to trust the randomness of its own coin tosses.

Similar to the NP proof system, note that the definition of an interactive proof system for a language emphasizes the "yes-instances": when a string is in the language, B must be led to acceptance with high probability, but when a string is not in the language A is not required to convince B of the contrary.

A more general version of the above definition is where A is not a Turing machine, but an infinite state machine. However it has been shown by Feldman in [F] that this adds no extra power to the model. In fact, he shows that with respect to language recognition it is sufficient for A to be a deterministic PSPACE machine. The fact that A is probabilistic is of importance to the more subtle definition of zero-knowledge, which is given in the next section.

We define IP, *Interactive Polynomial time*, to be the class of languages for which there exists interactive proof systems.

The first examples of a language in IP but not known to be in NP have been exhibited by Babai and Szemeredi [BS]. Their examples are "matrix group nonmembership" and "matrix group order," where the matrix groups over finite fields are represented by a list of generator matrices. The other, more well-known example

of "graph nonisomorphism" is due to Goldreich, Micali, and Wigderson [GMW]. They have shown that the language of pairs of graphs that are nonisomorphic to each other is in IP.

2.3. Arthur–Merlin games. Babai independently conceived the notion of an "Arthur-Merlin Game," an interactive proof system in which Merlin plays the role of A and Arthur that of B. The interaction, though, is less "liberal" than in our model since Merlin sees all the coin tosses of Arthur. A message from Arthur to Merlin can only consist of a randomly selected string. In an interactive proof system, instead, the verifier is allowed, given a polynomial time computable function f, to secretly select a random string R and transmit only $f(R)$ to the prover (as in the interactive proof system for QNR of Example 1).

This restriction immediately implies that the languages recognized by an Arthur-Merlin game are a subset of those having an interactive proof system. Interestingly, Goldwasser and Sipser [GS] show that they are not a proper subset. However, there is value in having both definitions around. It is easier to design protocols using the interactive proof systems definition, and it is easier to prove complexity results using the Arthur-Merlin definition.

Though the ability to make secret random choices does not help us to recognize more languages, we believe that it is crucial for recognizing languages in zero-knowledge. We make precise this belief in the conjecture of § 3.7.

An Arthur-Merlin type of interactive proof system was already implicitly used in a paper by Blum [B1]. He showed an interactive protocol for recognizing the language of the Blum integers:

$$\mathrm{BL} = \{n \mid p^{\alpha} \text{ divides } n \text{ and } p^{\alpha+1} \text{ does not, where } p \equiv 3 \bmod 4 \text{ is prime and } \alpha \text{ is odd}\}.$$

The prover's goal was to demonstrate membership in BL without having to send n's prime factorization. In this proof system, the verifier talks only once and his message consists of sending the sequence of his coin tosses. Protocols of this type were also found by Goldwasser and Micali [GM1] to prove, without releasing the prime factorization membership in the languages:

$\mathrm{GM1} = \{n \mid n \text{ has exactly two distinct prime divisors}\}$ and

$\mathrm{GM2} = \{(x, n) \mid \gcd(x, \phi(n)) = 1\}$ where $\phi(n)$ is the number of positive integers smaller and relatively prime to n.

3. Zero-knowledge. Rather than giving the definition of zero-knowledge only for interactive proof systems, we will give a more general definition. We will define what it means for any interactive protocol (A, B) to be *zero-knowledge* for a language L, whether or not (A, B) is a proof system for L. Actually the definition will not depend on B at all; as we shall see, it says that for every polynomial time B', the distribution that B' "sees" on all its tapes, when interacting with A on input $x \in L$, is "indistinguishable" from a distribution that can be computed from x in polynomial time. We thus first focus on the notion of indistinguishability for random variables.

3.1. Indistinguishability of random variables. Throughout this paper, we will only consider families of random variables $U = \{U(x)\}$ where the parameter x is from a language L, a particular subset of $\{0, 1\}^*$, and all random variables take values in $\{0, 1\}^*$. Let $U = \{U(x)\}$ and $V = \{V(x)\}$ be two families of random variables. We want to express the fact that, when the length of x increases, $U(x)$ essentially becomes "replaceable" by $V(x)$. To do this, we consider the following framework.

A random sample is selected either from $U(x)$ or from $V(x)$ and it is handed to a "judge." After studying the sample, the judge will proclaim his verdict: 0 or 1. (We

may interpret 0 as the judge's decision that the sample came from $U(x)$; 1 as the decision that the sample came from $V(x)$.) It is then natural to say that $U(x)$ becomes "replaceable" by $V(x)$ for x long enough if, when x increases, the verdict of *any* judge becomes "meaningless," that is, essentially uncorrelated to the distribution from which the sample came.

There are two relevant parameters in this framework: the *size* of the sample and the amount of *time* the judge is given to produce his verdict. By bounding these two parameters in different ways we obtain different notions of indistinguishability for random variables. We focus on the three notions we believe to be the most important: equality, statistical indistinguishability, and computational indistinguishability. Roughly speaking, these notions correspond to the following restrictions on the relevant parameters. If the two families of random variables $\{U(x)\}$ and $\{V(x)\}$ are equal, then the judge's verdict will be meaningless even if he is given samples of arbitrary size and he can study them for an arbitrary amount of time. We will define the two families to be statistically indistinguishable if the judge's verdict becomes meaningless when he is given an infinite amount of time but only random, polynomial (in $|x|$) size samples to work on. We will define the two families to be computationally indistinguishable if the judge's verdict becomes meaningless when he is only given polynomial ($|x|$)-size samples and polynomial ($|x|$) time. Let us now proceed to formalize these notions.

DEFINITION (Statistical indistinguishability). Let $L \subset \{0,1\}^*$ be a language. Two families of random variables $\{U(x)\}$ and $\{V(x)\}$ are *statistically indistinguishable on* L if

$$\sum_{\alpha \in \{0,1\}^*} |\text{prob}\,(U(x)=\alpha) - \text{prob}\,(V(x)=\alpha)| < |x|^{-c}$$

for all constants $c>0$ and all sufficiently long $x \in L$.

Notice that, for U and V as above, if a "judge" is handed a polynomial (in $|x|$) size sample, having infinite computing power will not help him to decide whether it came from $U(x)$ or $V(x)$. His answer will be essentially useless, as he will say "1" with essentially the same probability in both cases.

Example 2. Let $U(x)$ assign equal probability to all strings of length $|x|$, and let $V(x)$ assign probability $2^{-|x|}$ to all strings of length $|x|$, except for $0^{|x|}$, which is given probability 0 and for $1^{-|x|}$, which is given probability $2^{-|x|+1}$. Then $\{U(x)\}$ and $\{V(x)\}$ are two families of random variables statistically indistinguishable on $\{0,1\}^*$.

To formalize the notion of computational indistinguishability we make use of nonuniformity (the reasons for this choice can be found in § 3.4). Thus, our "judge," rather than being a polynomial time Turing machine, will be a *poly-size family of circuits.* That is a family $C=\{C_x\}$ of Boolean circuits C_x with one Boolean output such that, for some constant $e>0$, all $C_x \in C$ have at most $|x|^e$ gates. In order to feed samples from our probability distributions to such circuits, we will consider only *poly-bounded* families of random variables, that is, families $U=\{U(x)\}$ such that, for some constant $d>0$, all random variable $U(x) \in U$ assigns positive probability only to strings whose lengths are exactly $|x|^d$. If $U=\{U(x)\}$ is a poly-bounded family of random variables and $C=\{C_x\}$ a poly-size family of circuits, we denote by $P(U, C, x)$ the probability that C_x outputs 1 on input a random string distributed according to $U(x)$. (Here we assume that strings assigned positive probability by $U(x)$ have lengths equal to the number of Boolean inputs of C_x.)

DEFINITION (Computational indistinguishability). Let $L \subset \{0,1\}^*$ be a language. Two poly-bounded families of random variables U and V are *computationally indistinguishable on* L if for all poly-size family of circuits C, for all constants $c>0$ and all

sufficiently long strings $x \in L$,

$$|P(U, C, x) - P(V, C, x)| < |x|^{-c}.$$

This notion of computational indistinguishability was already used by Goldwasser and Micali [GM] in the context of encryption and by Yao [Y] in the context of pseudorandom generation. It is trivial that if U and V are identical, then they are statistically indistinguishable. It is also not hard to see that if U and V are statistically indistinguishable, then they are computationally indistinguishable, as follows. Let C_x be a circuit and let S be the set of inputs on which C_x outputs 1. Since U and V are statistically indistinguishable, the value of $U(x)$ will be in S with almost exactly the same probability that the value of $V(x)$ will be. Hence $P(U, C, x)$ will be very close to $P(V, C, x)$.

Example 3. Consider a probabilistic encryption algorithm that is secure in the sense of Goldwasser and Micali [GM]; for n integer, let $U(1^n)$ and $V(1^n)$ be the random variables taking as values the possible encryptions of 0 and 1, respectively, on security parameter n. Then U and V are computationally indistinguishable on $L = \{1\}^*$.

We believe that the notion of computational indistinguishability for random variables achieves the right level of generality. Thus we will call *indistinguishable* any two families of random variables that are computationally indistinguishable.

Remark 2. Let us point out the robustness of the last definition. In this definition, we are handing our computationally bounded "judge" only samples of size 1. This, however, is not restrictive. We note that two families of random variables $\{U_x\}$ and $\{V_x\}$ are computationally indistinguishable (with respect to samples of size 1) if and only if when C_x is given a polynomial in $|x|$ number of input strings, each independently generated according to the distribution U_x, then the probability of accepting is close to the probability of accepting when V_x is used.

3.2. Approximability of random variables. We now formalize the notion that a random variable U is essentially easy to generate. That is, there exists an efficient algorithm that randomly outputs strings in a way that is indistinguishable from U.

DEFINITION. Let M be a probabilistic Turing machine that on input x halts with probability 1. We denote by $M(x)$ the random variable that, for each string α, takes on α with exactly the same probability that M on input x outputs α.

DEFINITION. Let $L \subset \{0, 1\}^*$ be a language and $U = \{U(x)\}$ a family of random variables. We say that U is *perfectly approximable on* L if there exists a probabilistic Turing machine M, running in expected polynomial time, such that for all $x \in L$, $M(x)$ is equal to $U(x)$. We say that U is *statistically* (*computationally*) *approximable on* L if there exists a probabilistic Turing machine M, running in expected polynomial time, such that the families of random variables $\{M(x)\}$ and $\{U(x)\}$ are statistically (computationally) indistinguishable on L.

In what follows, we will use *approximability* to mean computational approximability. We are now ready to define the notion of zero-knowledge.

3.3. Zero-knowledge protocols and proof systems. We first address the issue of a "cheating verifier," B', who is allowed not to follow the protocol.

DEFINITION. Let (A, B) be an interactive protocol. Let B' be an interactive Turing machine that has as input x and on an extra input tape H, where the length of H is bounded above by a polynomial in the length of x. (Figure 1 must be emended to allow B' this extra tape.) When B' interacts with A, A sees only x on its input tape, whereas B' sees (x, H). A good way to think of H is as some knowledge about x that

the cheating B' already possesses. Alternatively, H can be considered as the history of previous interactions that the cheating B' is trying to use to get knowledge from A; this is discussed in more detail in the next section. We assume that the total computation time of B' when interacting with A will be bounded above by a polynomial in the length of x.

For a run of the protocol on common input x and extra input H, we define the view of B' to be everything that B' sees. Namely, let σ (and ρ) be the strings contained in the random tapes of A (and B'). Say the computation of A and B', with these random choices, consist of n turns with B' going first, where a_i (and b_i) are the ith messages of A (and B'), respectively. Then, we say that $(\rho, b_1, a_1, \cdots, b_n, a_n)$ is the *view* of B' on inputs x and H, and let $\text{View}_{A,B'}(x, H)$ be the random variable whose value is this view. (Note that it would make no difference if we included in the view the material written by B' on its private tape, or excluded the strings that B' sends to A, since these bits can be efficiently computed from the other bits of the view.) For convenience, we consider each view to be a string from $\{0, 1\}^*$ of length exactly $|x|^c$ for some fixed $c > 0$.

DEFINITION. Let $L \subset \{0, 1\}^*$ be a language and (A, B) a protocol. Let B' be as above. We say that (A, B) is *perfectly* (*statistically*) (*computationally*) *zero-knowledge on L for* B' if the family of random variables $\text{View}_{A,B'}$ is perfectly (statistically) (computationally) approximable on

$$L' = \{(x, H) \mid x \in L \text{ and } |H| = |x|^c\}.$$

We say that (A, B) is *perfectly* (*statistically*) (*computationally*) *zero-knowledge on L* if it is perfectly (statistically) (computationally) zero-knowledge on L for all probabilistic polynomial time ITM B'.

Note that the definition of (A, B) being zero-knowledge (in some manner) for B' only depends on A and not at all on B. It might be less misleading to think of A as being zero-knowledge for B'. A similar issue arises in the definition of an interactive proof system in § 2.2. Part (2) of this definition depends only on B, and not at all on A.

Computational zero-knowledge is certainly the most general of the above notions, and we will refer to it simply as *zero-knowledge.* Zero-knowledge really captures any information, which could not have been obtained efficiently in polynomial time, about members of L. That is, if (A, B) is zero-knowledge, it is not possible, in probabilistic polynomial time, to extract any information about members of L by interacting with A, not even by "cheating."

DEFINITION. Let $L \subset \{0, 1\}^*$ be a language. We say that (A, B) is a *perfectly* (*statistically*) (*computationally*) *zero-knowledge proof system for L* if it is an interactive proof system for L and a perfectly (statistically) (computationally) zero-knowledge protocol on L.

We will refer to computationally zero-knowledge proof systems (the most general notion of the three) simply as *zero-knowledge proof systems.* This notion is totally adequate in the real world. That is, if (A, B) is a zero-knowledge proof system for L, it is not possible in polynomial time (not even for a "cheating" B') to interact with A and extract anything else besides proofs of membership in L.

3.4. Some remarks about the above definitions. First, let us stress that the coin tosses of B are an essential part of the notion of a view in the definition of zero-knowledge for B. Consider the language L of all composite integers and the following

protocol (A, B). On input an integer n, B randomly selects an integer x between 1 and n and relatively prime with n. It then sends A the number $a = x^2 \bmod n$. A responds by sending y, a randomly chosen square root of a mod n. Should the view consist only of the text of the interaction between A and B, then the above-mentioned protocol would be perfectly zero-knowledge on L. However, we define the view so as to contain also the coin tosses of B. Thus, if (x, a, y) is randomly selected in $\text{View}_{(A,B)}(n)$, then $\gcd(x+y, n)$ is not 1 or n with probability at least $\frac{1}{2}$. Thus, if factoring is not in probabilistic polynomial time, the above protocol is not zero-knowledge for B. (It should be noted, however, that in the definition of zero-knowledge (for *all* B'), it is *not* necessary to include the random bits of B' in the view; this is because we have included in the view the messages sent by B', and for every B' there is a B'', which is like B' except that it sends its random bits as part of its last message.)

Second, it should be explained why B' sees an additional string H. (The need for this was independently discovered by the authors of this paper, by Oren [O], and by Tompa and Woll [TW].) H may be thought about in a number of different ways; H may be some extra information that the verifier (cheating or not) happens to know. For example, a zero-knowledge protocol for graph isomorphism should remain zero-knowledge even if the verifier happens to know colorings for the graphs. It is also possible that the protocol will be inserted in the middle of another protocol, where the verifier has seen some history H. There is the fear that this H was generated perhaps by interacting with a machine of unlimited power; we want to rule out the possibility of the verifier then obtaining knowledge by using H when interacting with A. It is for a similar reason that the distinguishing circuits in the definition of computational indistinguishability are allowed to be nonuniform. We want to say that two families of random variables can be computationally distinguished if there are circuits that tell them apart, where the circuits may have wired in some information about x or some information obtained from the history of some protocol in which the protocol of interest is immersed. In other words, the view B' obtains on inputs (x, H) should look like the simulated distribution $M(x, H)$.

One test of a definition is that we should be able to prove those facts that intuition dictates *must* be true. One such fact is that the repetition of a zero-knowledge protocol a polynomial number of times is still zero-knowledge. We can prove this with our current definitions, but we cannot prove it if, for example, we do not give B' the extra string H. We can also prove that if B' wants to decide a special predicate $P(x)$ for $x \in L$, it does not help B' to engage in a zero-knowledge proof system for L. We can prove all the intuitively obvious "facts" we have tried to prove, but only time will tell if these definitions are the right ones, or if they need some further modifications. We feel (and hope) that these definitions capture exactly the intuitive ideas we have tried to capture.

Last, there is the peculiar fact that the machine M simulating the view is allowed to operate in *expected* polynomial time. This appears to be necessary for the zero-knowledge proof system for QNR given in § 6. To see why this is necessary in general, consider a pair (A, B), where B (on input of length n) sends n random bits α to A; if the predicate $P(\alpha)$ holds, then A sends a random β of length n such that $P(\beta)$ holds. Imagine that the predicate P is easily computable, but the number of strings of length n for which P holds is small—maybe a fraction n^{-10} or maybe n^{-20}—but we do not know exactly which; imagine that the only way we know to find a string for which P holds is to select random strings until one satisfying P is found. The only way we know how to simulate the view of B statistically closely (or even computationally indistinguishably) is to choose a random α; if $P(\alpha)$ holds, look through random n-bit

strings until a β is found such that $P(\beta)$ holds. This process is only expected polynomial time.

3.5. Examples of zero-knowledge languages. Trivially, all languages in BPP have perfect zero-knowledge proof systems. (A language is in BPP if there is a probabilistic, polynomial time machine which on each input computes membership in the language with small probability of error.)

The first nontrivial zero-knowledge proof systems (i.e., for recognizing languages not known to be in BPP) are the perfect zero-knowledge proof systems for the quadratic residuosity language QR given in § 5, and the statistically zero-knowledge proof system for QNR given in § 6.

Recently, Goldreich, Micali, and Wigderson have shown in [GMW] that the graph isomorphism language has a perfect zero-knowledge proof system, that the graph nonisomorphism language (though not known to belong to NP) has a statistically zero-knowledge proof system, and that all languages in NP possess computationally zero-knowledge proof systems if secure encryption schemes exist. In [BGGHKMR] it is proved that all languages in IP possess zero-knowledge proof systems.

Results by Boppana, Hastad, and Zachos [BHZ] and Fortnow [Fo] show that if an NP-complete language had a perfect or statistically zero-knowledge proof system, the polynomial time hierarchy would collapse. Thus it may not be surprising that the interactive proof systems in [GMW] for graph coloring were zero-knowledge only in a computational sense. A more immediate reason for their being computationally zero-knowledge is that they make use of probabilistic encryption [GM] (see Example 3). This may tempt us to interpret Fortnow's result as saying that encryption is crucial in any zero-knowledge proof system for NP-complete languages. (Further discussion on Fortnow's result can be found in § 7.)

3.6. A study of earlier proposals. Having reached the notion of a zero-knowledge proof system, let us now have a second look at the earlier, Arthur–Merlin type, proof systems of Blum [Bl] and Goldwasser and Micali [GM1] that we have already mentioned in § 2.3.

The Proof System for BL (defined in § 2.3). In his beautiful paper, assuming that integer factorization is computationally hard, Blum proposes a protocol for flipping a coin over the telephone. For "fairly" flipping a coin, Alice and Bob need an integer n whose prime factorization is known to Alice but not to Bob, and has a special property, namely, $n \in \mathrm{BL}$. Alice is sure that the coin flip is fair because she computes n by multiplying two randomly selected primes both congruent to 3 mod 4, and because she trusts that factoring is hard. Bob, after the coin has come up Head or Tail, checks that the flipping was fair by requesting n's factorization from Alice. Thus a different n should be selected for each coin flip. To make coin flipping more efficient, Blum proposed to test that $n \in \mathrm{BL}$, by means of a protocol that does not give away n's factorization in any obvious way. After slightly modifying it, Blum's protocol can be proved to be perfect zero-knowledge on BL. However, without this modification, it is not clear how much knowledge about n's factorization it releases.

The Proof Systems for GM1 *and* GM2 (defined in § 2.3). The cryptographic protocols of Goldwasser and Micali use the inefficient version of Blum's coin-flipping protocols, and thus assume that factoring integers is computationally difficult. Based on this assumption, they showed that their proof systems do not give away the prime factorization of an input n. That is, no cheating polynomial time verifier can, after participating in the protocol, compute n's factorization much faster than it could before.

More generally, their same protocols can actually be proved to be computationally zero-knowledge on, respectively, the languages GM1 and GM2. Thus, in particular, their protocols do not even give away whether or not, say, 3 is a quadratic residue mod n.

3.7. Interactive proof systems versus Arthur–Merlin games for zero-knowledge. We are now ready to formally express our belief that interactive proof systems are more appropriate than Arthur–Merlin games for recognizing languages in zero-knowledge.

CONJECTURE. There exist languages L that have perfect or statistical zero-knowledge proof systems, but do not have any Arthur–Merlin proof system that is perfect or zero-knowledge on L.

4. The quadratic residuosity problem. In this section we describe the necessary number-theoretic background and notation needed for the proofs in §§ 5 and 6.

Let $\mathbf{N}$ denote the natural numbers, $x \in \mathbf{N}$ and $\mathbf{Z}_x^* = \{y \mid 1 \leqq y < x, \gcd(x, y) = 1\}$. We can determine in time polynomial in $|x|$ and $|y|$ whether or not $y \in \mathbf{Z}_x^*$.

We say that y in $\mathbf{Z}_x^*$ is a *quadratic residue* mod x if there exists a w in $\mathbf{Z}_x^*$ such that $w^2 \equiv y \bmod x$. Otherwise, we call y in $\mathbf{Z}_x^*$ a *quadratic nonresidue* mod x.

FACT 1. Let $x \in \mathbf{N}$ and $y \in \mathbf{Z}_x^*$. Then, y is a quadratic residue mod x if and only if it is a quadratic residue mod all of the prime factors of x.

Define the quadratic residuosity predicate to be

$$Q_x(y) = \begin{cases} 0 & \text{if } y \text{ is a quadratic residue mod } x, \\ 1 & \text{otherwise.} \end{cases}$$

Then we have the following fact.

FACT 2. Let $x \in \mathbf{N}$ and $y \in \mathbf{Z}_x^*$. Given y and the prime factorization of x, $Q_x(y)$ can be computed in time polynomial in $|x|$.

Let $y \in \mathbf{Z}_x^*$ and the prime factorization of x be $\prod_{i=1}^{k} p_i^{\alpha_i}$. Then, the Jacobi symbol of y mod x is defined as

$$(y/x) = \prod_{i=1}^{k} (y/p_i)^{\alpha_i},$$

where $(y/p_i) = 1$ if y is a quadratic residue mod p_i, and -1 otherwise.

FACT 3. Given $x \in \mathbf{N}$ and $y \in \mathbf{Z}_x^*$, (y/x) can be computed in time polynomial in $|x|$.

The Jacobi symbol of y mod x gives some information about whether y is quadratic residue mod x or not. If $(y/x) = -1$, then y is a quadratic nonresidue mod x and $Q_x(y) = 1$. However, when $(y/x) = 1$, no efficient (probabilistic or deterministic polynomial time) solution is known for computing $Q_x(y)$ correctly with probability significantly better than $\frac{1}{2}$. This leads to the formulation of the quadratic residuosity problem.

DEFINITION. We define the *quadratic residuosity problem* as that of computing $Q_x(y)$ on inputs x and y, where y in Z_x^* and $(y/x) = 1$,

The current best algorithm for computing $Q_x(y)$, is to first factor x and then compute $Q_x(y)$. In fact, factoring integers and computing Q_x have been conjectured to be of the same time complexity. The difficulty of the quadratic residuosity problem has been used as a basis for the design of several cryptographic protocols [GM], [LMR], [Bl].

Define the following two languages:

$$\mathrm{QR} = \{(x, y) \mid x \in \mathbf{N}, y \in \mathbf{Z}_x^*, \text{ and } Q_x(y) = 0\},$$

$$\mathrm{QNR} = \{(x, y) \mid x \in \mathbf{N}, y \in \mathbf{Z}_x^*, (y/x) = 1, \text{ and } Q_x(y) = 1\},$$

where x and y are presented in binary.

Clearly, by Facts 1 and 2, both QR and QNR are in the intersection of CO-NP and NP. However, no probabilistic polynomial time algorithm is known that accepts these languages, and thus they are not trivially zero-knowledge. In § 5 we show a perfectly zero-knowledge proof system for QR, and in § 6 we show a statistically zero-knowledge proof system for QNR. The following facts will be useful in the zero-knowledge proofs of §§ 5 and 6.

FACT 4. Let $x \in \mathbf{N}$. Then, for all y such that $Q_x(y) = 0$, the number of solutions $w \in \mathbf{Z}_x^*$ to $w^2 \equiv y \bmod x$ is the same (independent of y).

FACT 5. Let $x \in \mathbf{N}$, $y, z \in \mathbf{Z}_x^*$. Then we have the following:

(a) If $Q_x(y) = Q_x(z) = 0$, then $Q_x(yz) = 0$.

(b) If $Q_x(y) \neq Q_x(z)$, then $Q_x(yz) = 1$.

FACT 6. Given x, y, the Euclidean gcd algorithm allows us to compute in polynomial time whether or not $y \in \mathbf{Z}_x^*$.

5. Zero-knowledge proofs of quadratic residuosity. Recall that $\mathrm{QR} = \{(x, y) \mid y$ is a quadratic residue mod $x\}$, where x and y are presented in binary.

We will first *informally* describe our zero-knowledge interactive proof system for QR, and then describe it with more rigor. Say that A and B are given (x, y), $|x| = m$; then the following is done m times:

- A sends B a random quadratic residue mod x, u.
- B sends A a random bit, *bit*.
- If $bit = 0$ then A sends B a random square root of u mod x, w; if $bit = 1$ then A sends B a random square root of (uy) mod x, w.
- B checks that either [$bit = 0$ and $w^2 \bmod x = u$] or [$bit = 1$ and $w^2 \bmod x = (uy) \bmod x$].

More formally, we assume, for convenience, that A starts the protocol.

A's PROTOCOL ON INPUT $(x, y) \in$ QR.

FOR $i = 1$ to m

Use random bits to generate u_i, a random quadratic residue mod x.

SEND u_i to B

GET a string β_i from B; let bit_i = the first bit of β_i (or 0 if β_i is empty). If $bit_i = 0$, use random bits and generate w_i, a random square root of u_i mod x; if $bit_i = 1$, use random bits and generate w_i, a random square root of $(u_i y)$ mod x.

SEND w_i to B

GET a string from B (this will merely indicate that B wishes to continue the protocol).

END FOR

SEND "terminate" (just a string to finish off the protocol) to B.

B's PROTOCOL ON INPUT (x, y).

See if $x \geqq 1$ and $y \in Z_x^*$; if not, halt.

FOR $i = 1$ to m

GET u_i from A. See if $u_i \in Z_x^*$; if not, halt.

Generate a random bit_i.

SEND bit_i to A.

GET w_i from A. See if $w_i \in Z_x^*$ and either [$bit_i = 0$ and $w_i^2 \bmod x = u_i$] or [$bit_i = 1$ and $w_i^2 \bmod x = (u_i y) \bmod x$]; if not, halt.

SEND "okay" (just a string to continue the protocol) to A.

END FOR

GET any string from A, and halt accepting.

This is clearly an interactive protocol.

CLAIM 1. *The above* (A, B) *protocol is an interactive proof system for* QR.

Proof. Say that B is interacting with an arbitrary A'. Say that $x \geqq 1$, $y \in Z_x^*$, and y is *not* a quadratic residue mod x. For each u_i that B receives from A' it cannot be the case that *both* u_i and $(u_i y)$ have square roots mod x. Since A' does not see any bit_i in advance, there is at most a $\frac{1}{2}$ probability that B will "okay" the ith pass. Hence, the probability that B will say "convinced" is at most $1/2^m$.

We will now show that the protocol is zero-knowledge for QR.

THEOREM 1. *The above* (A, B) *protocol is a perfectly zero-knowledge proof system for* QR.

Proof. Let B' be an arbitrary polynomial time ITM that interacts with A. Let $(x, y) \in \text{QR}$ be the common input to the pair (A, B'), $|x| = m$, and let H be the extra input to B'. For convenience we consider the view $\text{View}_{A,B'}((x, y), H)$ to consist of the random variables

$$R, U_1, \text{BIT}_1, W_1, U_2, \text{BIT}_2, W_2, \cdots, U_m, \text{BIT}_m, W_m,$$

where R is the string of random bits generated by B', U_i takes on the value u_i, BIT_i takes on the ith message of B', etc.

One way to describe the distribution of the view is as follows: R is assigned a random bit string r (of the appropriate length). Say that $R, U_1, \text{BIT}_1, W_1, \cdots, U_i, \text{BIT}_i, W_i$ is the random variable V_i. Assume that for some i, $1 \leqq i < m$, V_i has been given the value v_i; we will describe the experiment for giving values to $U_{i+1}, \text{BIT}_{i+1}, W_{i+1}$.

EXPERIMENT

Choose for U_{i+1} a random quadratic residue mod x, u_{i+1}. If B' were B, we would choose BIT_{i+1} to be the $(i+1)$st bit of r. However, all we can say is that BIT_{i+1} is assigned the value $bit_{i+1} = f(x, y, H, v_i, u_{i+1})$, where f is some $\{0, 1\}$-valued function computable in deterministic, polynomial time. If $bit_{i+1} = 0$, then W_{i+1} gets the value w_{i+1}, a random square root of $u_{i+1} \bmod x$; if $bit_{i+1} = 1$, then W_{i+1} gets the value w_{i+1}, a random square root mod x of $(u_{i+1} y) \bmod x$.

Having characterized the view with the above experiment, we will now describe a probabilistic Turing machine M that, given $(x, y) \in \text{QR}$ and a string H, runs in expected polynomial time, and such that its output distribution $M((x, y), H)$ is *exactly* the same as V_m above; that is, $M((x, y), H)$ is the same as $\text{View}_{A,B'}((x, y), H)$. M begins by choosing r equals a random bit string (of the appropriate length). Assume that v_i has been chosen for some i, $1 \leqq i < m$; M outputs $u_{i+1}, bit_{i+1}, w_{i+1}$ according to the following program:

```
DO FOREVER
    bit_{i+1} := a random member of {0, 1}
    w_{i+1} := a random member of Z_x^*
    IF bit_{i+1} = 0 THEN
        u_{i+1} := w_{i+1}^2 mod x
    ELSE
        u_{i+1} := (w_{i+1}^2 y^{-1}) mod x
    IF bit_{i+1} = f(x, y, H, v_i, u_{i+1}) THEN
        OUTPUT u_{i+1}, bit_{i+1}, w_{i+1} and HALT
END DO
```

Two things about M's program must be clarified. First, the way M chooses a random member of Z_x^* is by choosing random m-bit strings until one is found that is in $\mathbf{Z}_x^*$; this halts in expected polynomial time. Secondly, by "y^{-1}" we mean that unique member of Z_x^* which when multiplied by y mod x yields 1.

We now show that M halts in expected time polynomial in m, with exactly the right output distribution.

Let $R', \{U_i', \mathrm{BIT}_i', W_i'\}$ be the random variables corresponding to the output of M, and let V_i' be defined similarly to V_i. Certainly R' has exactly the same distribution as R. Let $1 \leqq i < m$ and assume that V_i' has exactly the same distribution as V_i, and assume that M gives a value to V_i' in expected time polynomial in m. Say that both V_i and V_i' have been given the value v_i; we wish to show that the above piece of program code halts in expected time polynomial in m, with the same output distribution as the above experiment, given that $V_i = V_i' = v_i$.

Consider the body of the DO loop up to but not including the last test. If $bit_{i+1} = 0$ at this point, then since every quadratic residue in Z_x^* has the same number of square roots (mod x), u_{i+1} is equally likely to be any quadratic residue, and w_{i+1} will be a random square root of u_{i+1}; if $bit_{i+1} = 1$ at this point, then u_{i+1} will also be a random quadratic residue in this case (since y is a quadratic residue), and w_{i+1} will be a random square root of $(u_{1+1}y)$ mod x. Therefore, the body of the DO loop has the following *effect* (even though the following code may not be efficiently executable):

EQUIVALENT DO BODY
$u_{i+1} :=$ a random quadratic residue mod x.
$bit_{i+1} :=$ a random bit
IF $bit_{i+1} = f(x, y, H, v_i, u_{i+1})$ THEN
IF $bit_{i+1} = 0$ THEN $w_{i+1} :=$ a random square root mod x of u_{i+1} FI
IF $bit_{i+1} = 1$ THEN $w_{i+1} :=$ a random square root mod x of $(u_{i+1}y)$ FI
HALT and output $(u_{i+1}, \mathrm{bit}_{i+1}, w_{i+1})$
FI

It is clear that the equivalent body halts (and outputs) with probability $\frac{1}{2}$, and therefore that the actual DO loop halts in expected polynomial time. Since for each value of u_{i+1} the equivalent body is equally likely to halt, U_{i+1}' gets assigned (by the DO loop) a random quadratic residue. BIT_{i+1}' will be assigned $f(x, y, H, v_i, u_{i+1})$. Lastly, we can see from the equivalent body that in the case where the DO loop halts, W_{i+1}' gets assigned a random square root of u_{i+1} or of $(u_{i+1}y)$, depending on bit_{i+1}, as required.

6. Zero-knowledge proofs of quadratic nonresiduousity. We define $\mathrm{QNR} = \{(x, y) \mid y \in \mathbf{Z}_n^*, (y/x) = 1, Q_x(y) = 1\}$, where x and y are presented in binary.

Let (A, B) be an interactive protocol given as input (x, y) such that $|x| = m$.

The basic idea of the protocol is that B generates at random elements w of two types: $w \equiv r^2$ mod x (type 1) and $w \equiv r^2 y$ mod x (type 2), and sends these elements to A. If $(x, y) \in \mathrm{QNR}$ then A can tell of which type w is by computing whether w is a quadratic residue (type 1) or not (type 2). If (x, y) is not a member of QNR, w is always a quadratic residue mod x and A cannot guess its type better than guessing at random. Thus, A will not be able to tell the types of the w's and B will not be convinced that $(x, y) \in \mathrm{QNR}$.

This idea is sufficient as a proof system but not as a zero-knowledge proof system. The danger is that B may not have followed the protocol and generated elements w in a manner differently than specified in the protocol. We get over this difficulty by complicating the protocol to force B to convince A that indeed B knows whether w

is of type 1 or type 2. He does this by convincing A that he knows either a square root of w or a square root of $wy^{-1} \bmod x$, without giving A any information (in the information-theoretic sense) of which one he really knows.

Our original protocol, which appeared in [GMR], was more complex to prove than the one presented here. The simplified protocol presented here was suggested by Cohen [Co].

As was done for the QR language, we will first informally describe an interactive protocol, which we claim is a statistical zero-knowledge interactive proof system for QNR, and then describe it with more rigor.

The following (A, B) protocol on input (x, y) should be repeated m times.

- B picks at random $r \in \mathbf{Z}_x^*$ and $bit \in \{0, 1\}$. If $bit = 0$, B sets $w = r^2 \bmod x$; otherwise B sets $w = r^2 y \bmod x$. B sends w to A.
  For $1 \leqq j \leqq m$, B picks random $r_{j1}, r_{j2} \in \mathbf{Z}_x^*$ and a random $bit_j \in \{0, 1\}$. B sets $a_j = r_{j1}^2 \bmod x$, and $b_j = y r_{j2}^2 \bmod x$. If $bit_j = 1$, B sends A the ordered pair, $\text{pair}_j = (a_j, b_j)$; else if $bit_j = 0$, B sends to A $\text{pair}_j = (b_j, a_j)$.
- A sends B an m-long random bit vector $i = i_1 i_2 \cdots i_m$.
- B sends A the sequence $v = v_1, v_2, \cdots, v_m$; if $i_j = 0$ then $v_j = (r_{j1}, r_{j2})$; if $i_j = 1$ then $v_j = r r_{j1} \bmod x$ (a square root of $wa_j \bmod x$) if $bit = 0$, and $v_j = y r r_{j2} \bmod x$ (a square root of $wb_j \bmod x$) if $bit = 1$.
  (The intuition behind this step is as follows: if $i_j = 0$, then B is convincing A that pair_j was chosen correctly; if $i_j = 1$, then B is convincing that *if* pair_j was chosen correctly, then w was chosen correctly.)
- A verifies that the sequence v was properly constructed. If not, A sends *terminate* to B and halts. Otherwise, A sets $answer = 0$ if w is a quadratic residue mod x and 1 otherwise. A sends *answer* to B.
- B checks whether $answer = bit$. If so B continues the protocol, otherwise B rejects and halts.

After m repetitions of this protocol, if B did not reject thus far, B accepts and halts.

More formally, we proceed to describe first the protocol for B and then the protocol for A. The protocol consists of B going through its first stage, followed by A's first stage, followed by B's second stage, followed by A's second stage, etc., until either A or B chooses to terminate the protocol.

Denote by $v = \{v; v_j\}$ the result of extending sequence v with element v_j.

B's PROTOCOL ON INPUT (x, y).
Check that $x \geqq 1$ and that $y \in \mathbf{Z}_x^*$ and that $(y/x) = 1$.
Set $m = |x|$.
Repeat Stages 1-3 m times.
Stage 1.[1]
    use random bits to pick $r \in \mathbf{Z}_x^*$ and $bit \in \{0, 1\}$.
    IF $bit = 0$ set $w \equiv r^2 \bmod x$, else set $w \equiv r^2 y \bmod x$ FI
    FOR $j = 1, 2, \cdots, m$
        choose random $r_{j1}, r_{j2} \in \mathbf{Z}_x^*$ and random $bit_j \in \{0, 1\}$.
        set $a_j \equiv r_{j1}^2 \bmod x$ and $b_j \equiv r_{j2}^2 y \bmod x$.

[1] The careful reader may observe that picking $r \in Z_x^*$ "exactly at random" can be done in expected polynomial time, while our B must by definition run in a fixed polynomial number of steps. Fortunately, B can pick $r \in Z_x^*$ "almost at random" in a fixed polynomial time. This will have a negligible effect on the result and we omit any further details on this point.

IF $bit_j = 0$ set $\text{pair}_j = (a_j, b_j)$ else set $\text{pair}_j = (b_j, a_j)$ FI
END FOR
SEND $(w, \text{pair}_j \text{ for } j = 1, \cdots, m)$ to A.

Stage 2.
GET from A and m-long bit vector $i = i_1 \cdots i_m$, where $i_j \in \{0, 1\}$.
Initialize the sequence v to the empty sequence.
FOR $j = 1, \cdots, m$
IF $i_j = 0$ then set $v_j = (r_{j1}, r_{j2})$ and set $v = \{v; v_j\}$ FI
IF $i_j = 1$ do one of the following:
IF $bit = 0$ then set $v_j = rr_{j1} \bmod x = \sqrt{wa_j} \bmod x$ and set $v = \{v; v_j\}$
otherwise set $v_j = yrr_{j2} \bmod x = \sqrt{wb_j} \bmod x$ and set $v = \{v; v_j\}$ FI
FI
END FOR
SEND v to A.

Stage 3.
GET $answer \in \{0, 1\}$ from A.
IF $answer \neq bit$ then reject and halt
otherwise go to stage 1 FI.

After m iterations of Stages 1-3, if the protocol has not halted by now, accept and halt.

A's PROTOCOL ON INPUT $(x, y) \in$ QNR.

Stage 1.
GET $(w, \text{pair}_j \text{ for } j = 1, \cdots, m)$ from B.
Pick at random $i = i_1 \cdots i_m$ where $i_j \in \{0, 1\}$.
SEND i to B.

Stage 2.
GET sequence v from B
for every $j = 1, \cdots, m$ check that, if $i_j = 0$, then v_j is a pair (s, t) such that $(s^2 \bmod x, t^2 y \bmod x)$ equals pair_j, possibly with the elements interchanged; and if $i_j = 1$, then $(v_j^2) w^{-1} \bmod x$ is a member of pair_j. If not, SEND *terminate* to B and halt.
(Assume that the above checks have succeeded.)
If w is a quadratic residue mod x, set $answer = 0$ and if w is a quadratic nonresidue mod x, set $answer = 1$.
SEND *answer* to B.
Go to Stage 1.

We first prove that (A, B) is an interactive proof system for QNR.

CLAIM 2. (A, B) *is an interactive proof system for* QNR.

Proof. Clearly (A, B) is an interactive protocol. If $(x, y) \in$ QNR and A and B follow the specification of the protocol, then for every execution of Stages 1-2 by B, w is a quadratic nonresidue mod x if and only if $bit = 1$. Thus, in A's Stage 2, A can always decide whether w is a quadratic residue mod x or not and send *answer* to B such that $answer = bit$ and B will always accept.

Suppose that (x, y) not in QNR (i.e., y is a quadratic residue mod x) and that B is interacting with an arbitrary prover A', in the kth iteration of Stages 1-3. Then we claim that even an A' with infinite computation power cannot distinguish an interaction with B where $bit = 0$ from an interaction with B where $bit = 1$. This is argued as follows. At Stage 1, A' gets the list $(w, \text{pair}_j \text{ for } j = 1, \cdots, m)$, where w is a *random* quadratic residue, and where pair_j simply consists of a pair of *random* quadratic residues. This gives absolutely no information about the value of *bit*.

Now consider Stage 2, where $i_j = 0$. A gets (r_{j1}, r_{j2}). Note that r_{j1} is just a random square root of a_j and r_{j2} is a random square root of $b_j/y \bmod x$. So all that A sees is the result of randomly choosing (or not) to reorder pair_j, and then taking a random square root of the first element and a random square root of the result of dividing the second element by y (all mod x, of course). This gives no information about *bit*.

Now consider the case where $i_j = 1$. If $bit = 0$ then A gets $rr_{j1} \bmod x$, which is a random square root of $wa_j \bmod x$. If $bit = 1$ then A gets $yrr_{j2} \bmod x$, which is a random square root of $wb_j \bmod x$. Since pair_j is a random reordering of (a_j, b_j), v_j is equally likely to be a random square root of w times the first element of pair_j as it is to be a random square root of w times the second element of pair_j, no matter what *bit* is.

Thus, from the information that A' receives in Stages 1 and 2, the value of *bit* is as likely to be 0 as it is to be 1 and the chance that A' predicts *bit* correctly is no greater than $\frac{1}{2}$. In m iterations through Stages 1-3, the probability that A' computed *answer* such that *answer* = *bit* is at most $1/2^m$. □

Proving that the (A, B) proof system is statistically zero-knowledge for QNR is much more complex.

THEOREM 2. *The above protocol* (A, B) *is a statistically zero-knowledge proof system for* QNR.

Proof. Let B' be an arbitrary probabilistic polynomial time interactive Turing machine interacting with A. Let $(x, y) \in \text{QNR}$ be input to (A, B'), let $m = |x|$, and let H be the extra input to B'.

For convenience, consider the random variable $\text{View}_{A,B'}((x, y), H)$ (B''s view of an iteration of the protocol) to consist of the random variables:

$$\text{RAN}, \quad \text{and}$$

$$\{W^k, \{\text{PAIR}_j^k : 1 \leqq j \leqq m\}, \{I_j^k : 1 \leqq j \leqq m\}, V^k, \text{ANSWER}^k \mid 1 \leqq k \leqq m\}.$$

RAN is the string of random bits generated by B'; W^k takes on the value of w in the kth iteration of the protocol; PAIR_j^k takes on the value of pair_j in the kth iteration of the protocol; I_j^k takes on the value of i_j in the kth iteration of the protocol; $V^k = \{V_j^k\}$ takes on the value of the sequence v in the kth iteration of the protocol; and ANSWER^k takes on the value of *answer* in the kth iteration of the protocol.

For simplicity (notational and otherwise) we concentrate on showing that a single iteration of the protocol is zero-knowledge. Doing the general case implies carrying along the view of the protocol so far as was done in the proof of Theorem 1. Thus, from here on we drop all superscripts and work with the random variables: RAN, W, $\{\text{PAIR}_j\}$, $\{I_j\}$, $V = \{V_j\}$, and ANSWER.

Note that in a good execution of the protocol (namely, if B's protocol is followed), we expect that $W = r^2 \bmod x$ or $W = r^2 y \bmod x$, where r is a substring of RAN; and that $\text{PAIR}_j = (r_{j1}^2 \bmod x, r_{j2}^2 y \bmod x)$ or $(r_{j2}^2 y \bmod x, r_{j1}^2 \bmod x)$, where r_{j1} and r_{j2} are substrings of RAN; and that for all $1 \leqq j \leqq m$, if $I_j = 0$ then V_j will equal (r_{j1}, r_{j2}); and if $I_j = 1$ then $V_j = rr_{j1} \bmod x$ or $yrr_{j2} \bmod x$.

However, since B' may not follow the protocol, all we can say about these random variables is that RAN is a random binary string; W (and PAIR_j) are assigned values w (and pair_j) computed by B' on inputs x, y, H and RAN; I is a random binary string of length m; and V_j is a value computed by B' on inputs x, y, H, RAN, I.

We will now describe a probabilistic Turing machine M that, given $(x, y) \in \text{QNR}$ and H, runs in expected polynomial time, and whose output distribution is statistically indistinguishable from $\text{View}_{A,B'}((x, y), H)$.

M starts by outputting a random string *ran* of the appropriate length and running B' on inputs (x, y, H), and random tape *ran* on it. B' goes through Stage 1 outputting w, pair_j, for $1 \leqq j \leqq m$.
Next, M chooses $i_1, \cdots, i_m$ at random in $\{0, 1\}$, sets $i = i_1 \cdots i_m$, and writes i on B'"s communication tape, activating B'"*s* Stage 2.
B' goes into Stage 2, writing on its communication tapes a sequence $v = \{v_j\}$. M outputs $w\{\text{pair}_j\}$, i, v.
Next M does the checking that A does in Stage 2.
If the check fails M outputs "*terminate*" and halts.

Let us assume that the check succeeds. Think of x, y, H, *ran* as being fixed, so that w and $\{\text{pair}_j\}$ are also fixed. The fact that the check succeeds means that A sending i to B' causes B' to send a v to A, which causes A to send a one-bit *answer* to B' (rather than *terminate*); let us call any such i' *special.* M has just computed that i is special, and now wants to compute the value of *answer* that A would send. This value is 0 if w is a quadratic residue mod x, and 1 otherwise. Since B' may not have computed w the way B would have, it is not obvious how to compute the quadratic residuosity of w, i.e., *answer.*

It turns out that finding one other special string $i' \neq i$ will allow M to determine if w is a quadratic residue, as follows:

> Say that $i_j = 0$ and $i'_j = 1$ and i, i' are special. Let v, v' be the sequences sent by B' after receiving i or i' (respectively); these can be computed in polynomial time by running B'. Since $i_j = 0$, $v_j = (s, t)$, where $(s^2 \bmod x, t^2 y \bmod x)$ equals pair_j, possibly with the elements reversed. Since $i'_j = 1$, $(v_j)^2 w^{-1} \bmod x \in \text{pair}_j$. If $(v_j)^2 w^{-1} \bmod x = s^2 \bmod x$, then w is a quadratic residue mod x; if $(v_j)^2 w^{-1} \bmod x = t^2 y \bmod x$ then w is a quadratic nonresidue mod x.

It therefore remains to find a special $i' \neq i \bmod x$. M uses the following algorithm.

ALGORITHM TO FIND A SPECIAL $i' \neq i$.

> Test 2^m random i' of length m (with replacement), halting when either a special $i' \neq i$ is found, or when 2^m strings have been tried. If no special $i' \neq i$ has been found, then test *all* m-bit strings (in order), looking for a special $i' \neq i$.

If a special $i' \neq i$ is found, then M calculates *answer* as explained above and outputs *answer.* If no such i' exists, then M outputs "?"; note that this will happen when i is the *only* special string.

In order to show that M operates in expected polynomial time, it is sufficient to show that M operates in expected polynomial time for each fixed value of (x, y, H, ran). Say that x, y, H, *ran* are fixed, and so w, pair are also fixed. Let k be the number of strings i that are special. If $k = 0$, then the ALGORITHM TO FIND A SPECIAL i' will not be invoked, and the running time is clearly polynomial in m. If $k = 1$, then with probability $(1/2^m)M$ will choose a special i; in that case the ALGORITHM will run for time $2^m m^c$ (for some c), so the expected running time is $(1/2^m)2^m m^c +$ a polynomial in m.

Assume that $k > 1$. M will choose a special i with probability $k/2^m$. To calculate an upper bound on the expected running time of the ALGORITHM, imagine that it was changed so that it tested random i', including i and with replacement, halting if and when a special $i' \neq i$ is found; the expected running time would be at least half that of the ALGORITHM. In effect, a coin is being tossed until "heads" comes up, where the probability of "heads" is exactly $(k-1)/2^m$. It is well known that the expected number of coin tosses is exactly $2^m/(k-1)$. Hence the expected time for the

ALGORITHM is $\leqq(2^m/(k-1))m^c$ (for some c). The total expected time is $\leqq(k/2^m)$-$(2^m/(k-1))m^c$ + a polynomial in m, which is polynomial in m.

Recall that $M((x, y), H)$ is the random variable denoting the distribution of M's output given x, y, H. It remains to show that M is statistically close to $\text{View}_{A,B'}$. Fix x, y, H. If *ran* is such that the number of special strings is not exactly 1, then any output string α beginning with *ran* is taken on by $\text{View}_{A,B'}((x, y), H)$ with exactly the same probability as by $M((x, y), H)$. Let S be the set of $\alpha = ran, w, \{\text{pair}_j\}, i, v, answer$, where i is the *unique* special string determined by *ran*. The probability that $\text{View}_{A,B'}((x, y), H)$ takes on a value in S is $\leqq 1/2^m$, since for each *ran* there is at most one i, which will be the unique special string. Similarly, the probability that $M((x, y), H)$ takes on a value in S is $\leqq 1/2^m$. Thus,

$$\sum_{\alpha} |\text{prob}\,(M((x, y), H) = \alpha) - \text{prob}\,(\text{View}_{A,B'}((x, y), H) = \alpha)|$$

$$= \sum_{\alpha \notin S} |\text{prob}\,(M((x, y), H) = \alpha) - \text{prob}\,(\text{View}_{A,B'}((x, y), H) = \alpha)|$$

$$+ \sum_{\alpha \in S} |\text{prob}\,(M((x, y), H) = \alpha) - \text{prob}\,(\text{View}_{A,B'}((x, y), H) = \alpha)|$$

$$\leqq 0 + \frac{1}{2^m} + \frac{1}{2^m} = \frac{2}{2^m}.$$

And for m iterations of the protocol, the difference is

$$\sum_{\alpha} |\text{prob}\,(M((x, y), H) = \alpha) - \text{prob}\,(\text{View}_{A,B'}((x, y), H) = \alpha)| \leqq \frac{2m}{2^m}.$$

This completes our proof. □

Remarks. In fact, the above protocol can be shown to be perfect zero-knowledge. We just have to change M so that when it discovers that i is the unique special string it *factors* x in time roughly 2^m, and then determines if w is a quadratic residue mod x in polynomial time. This does not change the expected running time by more than a polynomial factor, since when M decides to do this extra work, it has already spent time 2^m.

7. Related work.

7.1. Work related to interactive proof systems. In studying his Arthur-Merlin games, Babai [Ba] has focused on the number of *rounds*, i.e., the number of times the prover and the verifier alternate in sending messages. Babai denotes the set of all languages accepted by i *rounds* in an Arthur-Merlin proof system by AM[i], and AM[$f(n)$] denotes the set of languages accepted by an Arthur-Merlin proof system with $f(n)$ rounds. Here f is a nondecreasing function from natural numbers to natural numbers, and n the length of the input.

The elegant simplicity of Babai's definition allowed him to show that for every constant k, AM[k] collapses to AM[2]. This in turn is a subset of both Π_2^p and nonuniform NP.

We define IP[$f(n)$] as the class of languages having an interactive proof system with $f(n)$ rounds.

Goldwasser and Sipser [GS] show that, for all f, AM[$f(n)$] = IP[$f(n)$].

On the other hand, Aiello, Goldwasser, and Hastad [AGH] have shown that for any two nonconstant functions $g(n)$ and $f(n)$ such that $g(n) = o(f(n))$, there exists an oracle X such that (if we modify the definitions so that one is computing using the oracle X) IP[$g(n)$] is strictly contained in IP[$f(n)$]. This result is tight as Babai and Moran [BM] have shown that for all constants $c > 0$, IP[$f(n)$] = IP[$cf(n)$]. Namely, IP[$O(f(n))$] is well defined.

An interesting question is the following. Where does IP stand with respect to the polynomial time heirarchy? Boppana, Hastad, and Zachos [BHZ] have shown that if CO-NP has a constant-round interactive proof system, then the polynomial time hierarchy collapses. Thus, from the results of [GMW], [GS], and [BHZ], it follows that graph isomorphism is not NP-complete unless the polynomial time hierarchy collapses.

Other works related to the study of randomized and nondeterministic complexity classes appear in [P] and [ZF]. In Papadimitriou's *Games Against Nature*, the verifier is also a probabilistic polynomial time machine that flips coins and presents them to a prover capable of optimal moves. This is different from our model in that L is said to be accepted by a game against nature if $x \in L$ implies that the probability of the prover to win the game is greater than a $\frac{1}{2}$ rather than bounded away from a $\frac{1}{2}$.

Zachos and Furer [ZF], in a work investigating the robustness of probabilistic complexity classes, introduce a framework of probabilistic existential and universal quantifiers and prove several combinatorial lemmas about them. The AM and thus IP complexity classes can be formulated in terms of these special quantifiers.

7.2. Work related to knowledge complexity. Prior to our work, the theory of knowledge had received much attention in a model-theoretic framework (see [FHV] and [HM] for discussion). There are several essential differences between this framework and ours. In the latter, knowledge is defined with respect to a specific computational model with specific computational resources. In the former framework, there are no limitations on the computational power of the participants, i.e., they "know" all logical consequences of the information they possess. (For discussion of this aspect see, *Belief, Awareness, and Limited Reasoning* [FH].) As for another difference, in our model knowledge is defined with respect to an available public input and is gained by computing on this input. In their model-theoretic framework knowledge is gained by being told (or witnessing) that a certain event is true (e.g., the outcome of a coin flip is heads), rather than by computing.

Galil, Haber, and Yung [GHY] proposed the following extension of the concept of a zero-knowledge interactive proof systems. A language L is said to have a result-indistinguishable zero-knowledge proof system if there exists an interactive protocol (A, B) such that for every string $x \in \{0, 1\}^*$, A can convince B that $x \in L$ or x is not in L (whichever is the case) with high probability, such that no passive observer C can get any information of which is the case. They give a result-indistinguishable proof system for QR.

As previously mentioned, Goldreich, Micali, and Wigderson [GMW] have shown, subject to the existence of secure encryption schemes, that all languages in NP have computationally zero-knowledge proof systems. Subsequently, related notions of proof systems and zero knowledge were given by Brassard and Crepeau [BC] and Chaum [Ch]. They found that for any language L in NP, there is an interactive protocol that

(1) is zero-knowledge, and
(2) proves membership in L correctly (i.e., with probability approaching 1) only if factoring is computationally difficult and the prover is polynomial time.

Let us explicitly contrast their protocols with the ones in [GMW]. The latter ones

(1) correctly prove membership in L, and
(2) are zero-knowledge only if secure encryption schemes exist (which is true if factoring is difficult).

Finally, let us mention the recent result of Fortnow.

THEOREM [Fo]. *Assume a language L has an interactive proof system (A, B) that is statistically zero-knowledge with respect to B. Then L's complement has a constant-round interactive proof system.*

As a corollary, if SAT had statistically zero-knowledge proof systems, the polynomial time hierarchy would collapse.

Note that the hypothesis of Fortnow's theorem is much weaker than saying that (A, B) is a statistically zero-knowledge proof system on L, which would mean that, for all verifiers B', A is zero-knowledge on L for B'. Usually, it is defeating this latter quantifier "for all" that makes it hard to find a perfect or statistically zero-knowledge proof system. Thus Fortnow's result has the potential to be widely applicable.

Acknowledgments. We thank Mike Sipser, Steve Cook, and Mike Fischer who helped us focus on this research from its beginning. Without their enthusiastic encouragement we might not have completed this work.

Oded Goldreich and Ron Rivest, as usual, have been generous with comments and ideas. Thanks also go to Leonid Levin, Zvi Galil, and Dan Simon for having helped us in various ways.

Special thanks to Josh Cohen for simplifying our original zero-knowledge protocol for proving quadratic nonresiduosity. Thanks are also due to Manuel Blum for sharing with us so many beautiful ideas about cryptographic protocols.

Finally, we thank the anonymous referees, whose comments greatly improved this paper.

REFERENCES

[AGH] B. AIELLO, S. GOLDWASSER, AND J. HASTAD, *On the power of interaction*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 368–379.

[AH] B. AIELLO AND J. HASTAD, *Perfect zero-knowledge languages can be recognized in two rounds*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 439–448.

[BGGHKRM] M. BEN-OR, O. GOLDREICH, S. GOLDWASSER, J. HASTAD, J. KILIAN, P. ROGAWAY, AND S. MICALI, *Everything provable is provable in zero-knowledge*, in Proc. Crypto88, to appear.

[Bl] M. BLUM, *Coin flipping by telephone*, IEEE COMPCON 1982, pp. 133–137.

[BHZ] R. BOPPANA, J. HASTAD, AND S. ZACHOS, *Does co-NP have short interactive proofs?*, Inform. Process. Lett., 25 (1987) pp. 127–132.

[Ba] L. BABAI, *Trading group theory for randomness*, in Proc. 17th ACM Annual Symposium on Theory of Computation, 1975, pp. 421–429.

[BM] L. BABAI AND S. MORAN, *Arthur–Merlin games: A randomized proof system, and a hierarchy of complexity classes*, J. Comput. Sci. Systems; a previous version was entitled *Trading group theory for randomness*, in Proc. 17th Annual ACM Symposium on Theory of Computing, 1985, pp. 421–429.

[BS] L. BABAI AND E. SZEMEREDI, *On the complexity of matrix group problems*, in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, pp. 229–240.

[BC] G. BRASSARD AND C. CREPAU, *Non-transitive transfer of confidence: A perfect zero-knowledge interactive protocol for* SAT *and beyond*, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, October 1986.

[Ch] D. CHAUM, *Demonstrating the public predicate can be satisfied without revealing any information how*, in Proc. Crypto86.

[Co] J. COHEN (Benaloh), *Cryptographic capsules*, in Proc. Crypto86.

[CKS] A. CHANDRA, D. KOZEN, AND L. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[C] S. COOK, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium of Theory of Computation, 1971, pp. 151–158.

[CR] S. COOK AND R. RECKHOW, *The relative efficiency of propositional proof systems*, J. Symbolic Logic, 44 (1979).

[F] P. FELDMAN, private communication.

[FMRW] M. FISCHER, S. MICALI, C. RACKOFF, AND D. WITENBERG, *A secure protocol for the oblivious transfer*, unpublished manuscript, 1986.

[FFS] U. FEIGE, A. FIAT, AND A. SHAMIR, *Zero knowledge proofs of identity*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 210–217.

[Fo] L. FORTNOW, *The complexity of perfect zero-knowledge*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 204–209.

[FH] R. FAGIN AND J. HALPERN, *Belief, awareness, and limited reasoning*, in Proc. 9th International Joint Conference on Artificial Intelligence, 1985, pp. 491–501.

[FHV] R. FAGIN, J. HALPERN, AND M. VARDI, *A model theoretic analysis of knowledge*, in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, 1984, pp. 268–278.

[GM] S. GOLDWASSER AND S. MICALI, *Probabilistic encryption*, J. Comput. Science and Systems, 28 (1984), pp. 270–299.

[GM1] ———, *Proofs with untrusted oracles*, unpublished manuscript (submitted to STOC, 1984). Revised version: *The information content of proof systems*, unpublished manuscript (submitted to STOC, 1984).

[GMS] O. GOLDREICH, Y. MANSOUR, AND M. SIPSER, *Interactive proof systems: Provers that never fail and random selection*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 449–460.

[GMR] S. GOLDWASSER, S. MICALI, AND C. RACKOFF, *The knowledge complexity of interactive proof systems*, in Proc. 27th Annual Symposium on Foundations of Computer Science, 1985, pp. 291–304. Earlier version: *Knowledge complexity*, unpublished manuscript, (submitted to FOCS, 1984).

[GS] S. GOLDWASSER AND M. SIPSER, *Private coins versus public coins in interactive proof-systems*, in Proc. 18th Annual Symposium on Theory of Computing, 1986, pp. 59–68.

[GHY] Z. GALIL, S. HABER, AND M. YUNG, *A private interactive test of a Boolean predicate and minimum-knowledge public-key cryptosystems*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 360–371.

[GMW] O. GOLDREICH, S. MICALI, AND A. WIGDERSON, *Proofs that yield nothing but their validity and a methodology of cryptographic protocol design*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 174–187.

[GMW2] ———, *How to play any mental game*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 218–229.

[HM] J. HALPERN AND Y. MOSES, *Knowledge and common knowledge in a distributed environment*, in Proc. 3rd Principles of Distributed Computing Conference, 1984, pp. 50–61.

[LMR] M. LUBY, S. MICALI, AND C. RACKOFF, *How to simultaneously exchange a secret bit by flipping a symmetrically-biased coin*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, 1983, pp. 11–22.

[O] Y. OREN, *On the cunning power of cheating verifiers: some observations of zero-knowledge proofs*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 462–471.

[P] C. PAPADIMITRIOU, *Games against nature*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, 1983, pp. 446–450.

[TW] M. TOMPA AND H. WOLL, *Random self reducibility and zero knowledge interactive proofs of possession of information*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 472–482.

[ZF] S. ZACHOS AND M. FURER, *Probabilistic quantifiers vs. distrustful adversaries*, in Proc. Structure of Complexity Classes Conference, 1986.

[Y] A. YAO, *Theory and application of trapdoor functions*, in Proc. 23rd Annual IEEE Symposium on Foundations of Computer Science, November 1982, pp. 80–91.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 209-215, April 1989

A LOWER BOUND ON THE COMPLEXITY OF DIVISION IN FINITE EXTENSION FIELDS AND INVERSION IN QUADRATIC ALTERNATIVE ALGEBRAS*

THOMAS LICKTEIG†

Abstract. Let k be a field. The author continues the work of Lickteig [*SIAM J. Comput.*, 16 (1987), pp. 278-311] and extends it in two directions: (1) A lower bound on the Ostrowski complexity of division in a finite extension field $A \supset k$: $L_{\mathrm{DIV}} \geqq 3[A:k] + \log_2[B:k] - 2$, where $B \subset A$ is some simple extension of k of maximal degree. The proof combines the technique of adjoining intermediate results to the ground field [T. Lickteig, *op. cit.*] and lower bound criteria for approximative complexity recently obtained by Griesser [*Theoret. Comput. Sci.*, 46 (1986), pp. 329-338].

(2) An optimal lower bound for the complexity of inversion in a quadratic alternative algebra A of dimension greater than or equal to 2, $A \neq k \times k$: $L_{\mathrm{INV}} = 2 \dim A -$ index of the norm.

Key words. arithmetic in finite extension fields, Ostrowski complexity, substitution method, adjunction of intermediate results to the groundfield and approximative complexity

AMS(MOS) subject classifications. 68C20, 68C25

Throughout this paper k is assumed to be an infinite field. We use the same notation as in [8, § 2].

1. Finite extension fields. Let $A \supset k$ be a finite extension, $m = [A:k]$. Division in A is a k-rational mapping:

$$A \times A \to A, \qquad (x, y) \mapsto x^{-1}y,$$

with A being viewed as m-dimensional k-vector space. When we fix a k-basis $\omega_1, \cdots, \omega_m$ of A and use the basis $(\omega_1, 0), \cdots, (0, \omega_m)$ of $A \times A$, division is given by rational functions $q_1, \cdots, q_m \in k(X_1, \cdots, X_m, Y_1, \cdots, Y_m)$ such that $x^{-1}y = \sum_{i \leqq m} q_i(\boldsymbol{\alpha}, \boldsymbol{\beta})\omega_i$ if $x = \sum_{i \leqq m} \alpha_i \omega_i$, $y = \sum_{i \leqq m} \beta_i \omega_i$ with $\alpha_i, \beta_i \in k$. The complexity of division L_{DIV} is defined as the Ostrowski complexity $L(q_1, \cdots, q_m)$ (cf. [2]). As is easily verified, L_{DIV} does not depend on the choice of the bases, so it is well defined.

As k is assumed to be infinite, we may equivalently view $k(\mathbf{X}, \mathbf{Y})$ as functions on $A \times A$. Accordingly, in the sequel we allow ourselves to change to the function interpretation if it is more convenient. In this sense $X_1, \cdots, Y_m$ is the dual basis to $(\omega_1, 0), \cdots, (0, \omega_m)$ in $(A \times A)^*$.

We note that division always has a reduction to multiplication; also in the more general case of arbitrary associative algebras A: for a unit $\zeta \in A^x$ and $\eta \in A$, we have the expansion

$$(\zeta - \zeta x)^{-1} \cdot (\eta + \zeta(y - x\zeta^{-1}\eta)) = \zeta^{-1}\eta + \zeta^{-1}x\eta + y + xy + (\text{higher-order terms}).$$

Now, choosing (ζ, η) appropriately and applying Strassen's idea of computing the quadratic terms [9], we obtain for the left division $L_{\mathrm{LDIV}} \geqq L_{\mathrm{MULT}}$. Similarly for the right division in A, we obtain $L_{\mathrm{RDIV}} \geqq L_{\mathrm{MULT}}$. Both coincide if A possesses an anti-automorphism. So in case of finite field extensions $L_{\mathrm{DIV}} \geqq 2m - 1$ by Fiduccia and

* Received by the editors June 12, 1987; accepted for publication (in revised form) May 24, 1988. The research was done while the author was visiting the Institut für Angewandte Mathematik der Universität Zürich, Zürich, Switzerland. The hospitality and support of the institute, as well as the support of the Deutsche Forschungsgemeinschaft (Li 405/1-1), are gratefully acknowledged.

† Mathematisches Institut der Universität Tübingen, Auf der Morgenstelle 10, 74 Tübingen, Federal Republic of Germany.

Zalcstein [4], Winograd [10], and Alder and Strassen [1]; and for simple extensions this lower bound is the best we can get in this way.

To obtain a more explicit representation for the q_i, we recall the norm function $N_k^A: A \to k$, $N_k^A(x)$ being defined as the determinant of the regular representation R_x of $x \in A$ (e.g., [7]). Let S denote the maximal separable extension, $k \subset S \subset A$. By the transitivity of the norm $N_k^A = N_k^S \circ N_S^A$, where $N_S^A: x \mapsto x^\nu$, $\nu = [A:k]_i = [A:S]$. Let $\mu = \min\{\nu: \forall x \in A, x^\nu \in S\}$. By the theorem on primitive elements, we have

$$\begin{aligned} \mu &= \max\{[B:k]_i: B \subset A, B \supset k \text{ simple}\} \\ &= \max\{[B:k]/[S:k]: B \subset A, B \supset k \text{ simple}\}. \end{aligned} \tag{1}$$

The *principal norm* $N: A \to k$ is defined by $N(x) = N_k^S(x^\mu)$ for $x \in A$. Let $n: A \times A \to k$ be the composition $N \circ \pi_1$ of N with the first projection; so $n \in k[\mathbf{X}]$, and by (1) $\deg n = \max\{[B:k]: B \subset A, B \supset k \text{ simple}\}$. Let k^a denote an algebraic closure of k, and $\mathcal{J}$ the set of all embeddings $\iota: A \hookrightarrow k^a (|\mathcal{J}| = [A:k]_s)$. In $k^a[\mathbf{X}] (= k^a \otimes_k k[\mathbf{X}])$, n has the linear factorization

$$n = \prod_{\iota \in \mathcal{J}} \left(\sum_{j=1}^{m} \iota\omega_j X_j \right)^\mu, \tag{2}$$

while n is irreducible in $k[\mathbf{X}]$ (as $n(\alpha_1 X - \beta_1, \cdots, \alpha_m X - \beta_m) = \operatorname{Irr}(\iota\omega, k, X)$ if $1 = \sum \alpha_i \omega_i$ and $\omega = \sum \beta_i \omega_i$ has maximal degree over k ($\alpha_i, \beta_i \in k$, $\iota \in \mathcal{J}$)). It follows that the q_i can be written in a reduced quotient representation as

$$q_i = \frac{1}{n} \sum_{j=1}^{m} a_{ij} Y_j \qquad (i \leqq m), \tag{3}$$

with $a_{ij} \in k[\mathbf{X}]$, $\deg a_{ij} = \deg n - 1$.

For later purpose we remark that every nonzero $f \in A^*$ induces an isomorphy of A with A^*:

$$A \cong A^* \quad \text{by } \theta \mapsto R_\theta^*(f) = f \circ R_\theta \quad (f \in A^* - \{0\}) \tag{4}$$

($\theta \neq 0 \Rightarrow f(\theta A) = f(A) \neq 0$ implies injectivity).

As in [8] we now follow de Groote's line (initiated in [6]) of first finding the symmetries of the computation problem. The affine group $\operatorname{Aff}(A \times A, k)$ operates in the natural way on $k(\mathbf{X}, \mathbf{Y})$, and thus also on $k^a(\mathbf{X}, \mathbf{Y}) = (k^a \otimes_k k(\mathbf{X}, \mathbf{Y}))$. The symmetry group Γ of the division is defined as follows:

$$\Gamma = \{\alpha \in \operatorname{Aff}(A \times A, k): L(q_1, \cdots, q_m \bmod q_1^\alpha, \cdots, q_m^\alpha) = 0\}.$$

This means

$$kq_1 + \cdots + kq_m + \Lambda = kq_1^\alpha + \cdots + kq_m^\alpha + \Lambda, \tag{5}$$

where $\Lambda = kX_1 + \cdots + kY_m + k$. We have an embedding $\Delta(2, A) \hookrightarrow \Gamma$ of the group $\Delta(2, A)$ of triangular 2×2 matrices over A into Γ via $\left(\begin{smallmatrix} \zeta & \eta \\ 0 & \theta \end{smallmatrix}\right) \mapsto ((x, y) \mapsto (x\zeta, x\eta + y\theta))$ as the following diagram

$$\begin{array}{ccc} A^x \times A & \longrightarrow & A \\ {\scriptstyle \left(\begin{smallmatrix} \zeta & \eta \\ 0 & \theta \end{smallmatrix}\right)} \downarrow & & \downarrow {\scriptstyle R_{\zeta^{-1}\theta + \zeta^{-1}\eta}} \\ A^x \times A & \longrightarrow & A \end{array} \tag{6}$$

commutes; here the horizontal mappings indicate division.

Similarly, we have an embedding $\operatorname{Aut}(A/k) \hookrightarrow \Gamma$, $\alpha \mapsto (\alpha, \alpha)$ of the Galois group $\operatorname{Aut}(A/k)$ into Γ, for (6) with the vertical mappings replaced by (α, α) and α commutes.

THEOREM 1. *Let $A \supset k$ be a finite field extension. The symmetry group of division in A over k is*

$$\Gamma = \Delta(2, A) \cdot \mathrm{Aut}\,(A/k).$$

Proof. As $\Delta(2, A)$ and $\mathrm{Aut}\,(A/k)$ commute, their product is a subgroup of Γ. To prove the nontrivial inclusion, let $\alpha \in \Gamma$ be given. Let $X = \sum \omega_j X_j$, $Y = \sum \omega_j Y_j \in A(\mathbf{X}, \mathbf{Y})(= A \otimes_k k(\mathbf{X}, \mathbf{Y}))$. (As functions on $A \times A$, $X = \pi_1$ and $Y = \pi_2$.) Comparing denominators in (5), we get $n = \lambda n^\alpha$ with some $\lambda \in k^\times$ because n is irreducible in $k[\mathbf{X}]$. Extending the $\iota \in \mathcal{J}$ canonically, $\iota : A(\mathbf{X}) \hookrightarrow k^a(\mathbf{X})$, we get in $k^a[\mathbf{X}]$

$$\left(\prod_{\iota \in \mathcal{J}} \iota X\right)^\mu = \lambda \left(\prod_{\iota \in \mathcal{J}} \iota X^\alpha\right)^\mu,$$

by $(\iota X)^\alpha = \iota X^\alpha$. Thus there are ι, $\iota' \in \mathcal{J}$, $\theta' \in (k^a)^\times$ such that $\iota X = \theta'(\iota' X^\alpha)$. Using function interpretation on $A \times A$ and evaluating both sides at $(\alpha 1, 0)$, it follows that $\theta' \in \iota A$, say $\theta' = \iota\theta$. Thus, $\iota' X^\alpha = \iota(\theta^{-1} X) = \iota X^{\left(\begin{smallmatrix} \theta & 0 \\ 0 & 1 \end{smallmatrix}\right)}$, and hence we may assume without loss of generality $\iota' X^\alpha = \iota X$, by the action of $\Delta(2, A)$. But then $\iota A \subset \iota' A$; thus $\iota A = \iota' A$. So $\iota^{-1}\iota' \in \mathrm{Aut}\,(A/k)$. Thus, by the action of (the embedded copy of) $\mathrm{Aut}\,(A/k) \subset \Gamma$, we may change α such that $X^\alpha = X$. It suffices to prove $Y^\alpha \in AX + AY$, for then $\alpha \in \Delta(2, A)$. Equivalently we show $AY^\alpha + AX = AY + AX$. This is obtained by extending the ground field in (5) to A, multiplying (5) by $X = X^\alpha$, and intersecting with $A\Lambda$. To see this, it suffices to verify that

$$AY = (AXq_1 + \cdots + AXq_m) \cap A\Lambda;$$

the corresponding identity then also holds for Y^α. Let $\sum \xi_i X q_i \in A\Lambda$ with $\xi_i \in A$, $1 = \sum \alpha_i \omega_i$ with $\alpha_i \in k$. Substituting $Y_i \mapsto \alpha_i$ we obtain $\sum \xi_i X q_i(\mathbf{X}, \boldsymbol{\alpha}) \in A$, as the q_i are linear in $\mathbf{Y}$ (cf. (3)). Since the image of inversion is $A^\times$, $q_1(\mathbf{X}, \boldsymbol{\alpha}), \cdots, q_m(\mathbf{X}, \boldsymbol{\alpha}) \in k(\mathbf{X})$ are k-linearly independent, and, as A and $k(\mathbf{X})$ are linearly disjoint over k, they are also A-linearly independent. It follows that $\xi_1 = \xi\omega_1, \cdots, \xi_m = \xi\omega_m$ for some $\xi \in A$. Thus $\sum \xi_i X q_i = \xi Y$.

THEOREM 2. *Let $A \supset k$ be a finite extension field, and let $B \subset A$ be a simple extension of k of maximal degree. Then*

$$L_{\mathrm{DIV}} \geqq 3[A : k] + \log_2 [B : k] - 2.$$

Remark. For quadratic extensions the bound is 5, so the result of [8] $L_{\mathrm{DIV}} = 6$, which requires much more involved considerations, is not within reach of this bound.

Proof. The proof is done in three steps. For convenience we assume without loss of generality that $\omega_1 = 1$.

(i) $L(q_1) \leqq L(q_1, \cdots, q_m) - (m-1)$. To see this, we observe that any nontrivial linear combination of $q_1, \cdots, q_m$ can be written as $q_1^{\left(\begin{smallmatrix} 1 & 0 \\ 0 & \theta \end{smallmatrix}\right)}$ for some $\theta \in A^\times$ (use (4) and (6)). On the other hand, if $(p_1, \cdots, p_r)$ is a computation sequence for $\{q_1, \cdots, q_m\}$, an elimination shows that $(p_1, \cdots, p_{r-m+1})$ is a computation sequence for some nontrivial linear combination of them. Now use the fact that complexity is constant on Γ-orbits.

(ii) There exists $f \in k(\mathbf{X})$ such that $L((a_{11}/n) Y_1 + f) \leqq L(q_1) - (m-1)$: First we remark that for every $\xi = \sum \alpha_i \omega_i \in A^\times$, a_{11}/n is in the Γ-orbit of $q_1(\mathbf{X}, \boldsymbol{\alpha})$ (by $\left(\begin{smallmatrix} \zeta^{-1} & 0 \\ 0 & 1 \end{smallmatrix}\right) \in \Delta(2, A)$). There exists $q^{(m)} = q_1, q^{(m-1)}, \cdots, q^{(1)}$ with $L(q^{(s-1)}) < L(q^{(s)})$ of the form

$$q^{(s)} = \sum_{i \in I} \frac{1}{n} \left(a_{1i} + \sum_{j \notin I} \alpha_{ij} a_{1j}\right) Y_i + f,$$

where $I \subset \{1, \cdots, m\}, |I| = s, \alpha_{ij} \in k, f \in k(\mathbf{X})$. As follows from the remark, the coefficient of each Y_i is not contained in k, so $L(q^{(s)}) \neq 0$. Thus there exists a substitution of one of the Y_i of the kind

$$Y_i \mapsto \sum_{j \in I - \{i\}} \alpha_j Y_j + \tilde{f}$$

with $\alpha_j \in k$, $\tilde{f} \in k(\mathbf{X})$, reducing the complexity of $q^{(s)}$. This defines $q^{(s-1)}$ given $q^{(s)}$. By the remark, $q^{(1)}$ can be assumed to be of the form $(a_{11}/n) Y_1 + f$.

(iii) For all $f \in k(\mathbf{X})$ $L((a_{11}/n) Y_1 + f) \geqq m + \log \deg n$. The first member p not in $k(\mathbf{X})$ in an optimal computation sequence for $(a_{11}/n) Y_1 + f$ can be written as $p = (\cdots)^*/(Y_1 - h)$, or as $p = (Y_i - h)/g$ with $g, h \in k(\mathbf{X})$. Here we use the stronger technique from [8] of adjoining a certain intermediate result to the ground field and deleting an operation of arity zero. In our case we adjoin $t = (Y_1 - h)/g$ to k ($g = 1$ in the first case) and delete Y_1 in order to reduce the complexity, obtaining

$$\left(\frac{a_{11}}{n} g\right) t + \left(\frac{a_{11}}{n} h + f\right).$$

After multiplying the above by t^{-1}, it becomes degenerable into $a_{11} n^{-1} g$ (by $t^{-1} \mapsto 0$). So the complexity $L((a_{11}/n) Y_1 + f)$ is greater than the approximative complexity $\mathbf{L}(a_{11} n^{-1} g)$ (cf. Griesser [5], Bini [2], or the definition below). If n does not divide the reduced numerator of g, the subsequent lemma yields the desired result. Otherwise we apply the lemma to the intermediate result g, and use $L((a_{11}/n) Y_1 + f) > L(g)$.

Steps (i), (ii), and (iii) imply the assertion.

Let ε be a further indeterminate over k, $k_\varepsilon = k(\varepsilon)$, and let $\mathcal{O}_\varepsilon \subset k_\varepsilon(X_1, \cdots, X_m)$ be the domain of $\varepsilon \mapsto 0$ (a discrete valuation ring). The *approximative complexity* $\mathbf{L}(f_1, \cdots, f_s)$ of $f_1, \cdots, f_s \in k(\mathbf{X})$ is defined as the minimum $k_\varepsilon(\mathbf{X})$-complexity of some $F_1, \cdots, F_s \in \mathcal{O}_\varepsilon$ such that $F_{i(\varepsilon=0)} = f_i$ for $i \leqq s$ (cf. Griesser [5] and Bini [2]).

LEMMA 1. *Let $f = r/s$ with $r, s \in k[\mathbf{X}]$ be nonzero and reduced. Then $\mathbf{L}(f) \geqq m - 1 + \log \deg n$ if $n \mid r$, and $\mathbf{L}(f) \geqq m - 1 + \log (\deg n + 1)$ if $n \mid s$.*

For the proof we recall a result in [5]. There Griesser extends the substitution method to the approximative model. In terms of adjunction of intermediate results to the ground field [5], formulation and proof of his result become more transparent. Let us call $f \in k(\mathbf{X})$ K-linear if $f \in KX_1 + \cdots + KX_m + K$, $K \subset k(\mathbf{X})$ a subfield. We shall need Griesser's result in the following form.

THEOREM 3 [5]. *Let $f_1, \cdots, f_s \in k(\mathbf{X})$, not all k-linear.*

(i) *There exist a k-linear $l \in k(\mathbf{X})$ such that adjunction of l to k and deletion of some X_j reduces $\mathbf{L}(f_1, \cdots, f_s)$.*

(ii) *$r := \mathbf{L}(f_1, \cdots, f_s) \leqq m$ implies the existence of a tower of purely transcendental extensions $k = K_0 \subset K_1 \subset \cdots \subset K_r \subset k(\mathbf{X})$ such that*

$$\forall 1 \leqq i \leqq r \quad K_i = K_{i-1}(l_i), \qquad l_i \quad K_{i-1}\text{-linear},$$

$$f_1, \cdots, f_s \quad K_r\text{-linear}.$$

Proof. (i) The first member p in an optimal approximative computation sequence for $f_1, \cdots, f_s$ can be written as $p = (\cdots)^*/(\sum_1^m \beta_i X_i + \beta_0)$ with $\beta_i \in k_\varepsilon$. Without loss of generality $\nu_\varepsilon(\sum_1^m \beta_i X_i) = 0$, where ν_ε denotes the valuation associated with $\mathcal{O}_\varepsilon$ (shift a power of ε to the first operand). We use an idea from [3, p. 488] to make $\tilde{l} = \sum_1^m \beta_i X_i$ k-linear by means of a linear transform according to some $M \in k_\varepsilon^{m \times m}$ with $\nu_\varepsilon(M) \geqq 0$ and $M_{(\varepsilon=0)} = 1$ as follows. If $\nu_\varepsilon(\beta_j) = 0$, then

$$\tilde{l} = \beta_j X_j + \sum_{\substack{\nu_\varepsilon(\beta_i) = 0 \\ i \neq j}} \beta_i X_i + \sum_{\nu_\varepsilon(\beta_i) > 0} \beta_i X_i$$

is transformed to

$$l = \beta_j\left(\frac{\beta_j(0)}{\beta_j} X_j - \sum_{\nu_\varepsilon(\beta_i)>0} \frac{\beta_i}{\beta_j} X_i\right) + \sum_{\substack{\nu_\varepsilon(\beta_i)=0 \\ i\neq j}} \beta_i\left(\frac{\beta_i(0)}{\beta_i} X_i\right) + \sum_{\nu_\varepsilon(\beta_i)>0} \beta_i X_i = \tilde{l}_{(\varepsilon=0)}.$$

It is now clear that $\mathbf{L}(f_1, \cdots, f_s)$ decreases if we adjoin l to k and delete X_j.

(ii) Is proved by induction using (i).

Proof of Lemma 1. By Theorem 3(i) there is some $l = X_j + \sum_{i\neq j} \alpha_i X_i (\alpha_i \in k)$ such that adjunction of l to k and deletion of X_j reduces $\mathbf{L}(f)$. We consider the effect on n more closely. When we substitute X_j, the linear factors become $\sum_{i\neq j} (\iota\omega_i - \alpha_i \iota\omega_j) X_i + \iota\omega_j l \notin k^a(l)$. So, as n as element of $k[\mathbf{X}] = k[l][X_1, \cdots, X_{j-1}, X_{j+1}, \cdots, X_m]$ is irreducible, n as element of $k(l)[X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_m]$ is irreducible (by the Gauss lemma) and has the same degree. $m-1$ repetitions of this argument and a final application of the degree bound [5] yield the desired statements.

For inversion in a finite extension field we get the following corollary.

COROLLARY. *Let $A \supset k$ be a finite extension field, $B \subset A$ simple of maximal degree over k. Then*

$$L_{\mathrm{INV}} \geqq 2[A:k] + \log_2([B:k]+1) - 2.$$

Proof. Without loss of generality $\omega_1 = 1$. There is a nontrivial linear combination f of $a_{11}/n, \cdots, a_{m1}/n$ such that $L(a_{11}/n, \cdots, a_{m1}/n) \geqq L(f) + m - 1$. Now apply Lemma 1.

Remarks. (1) As we shall see in the next section (Example (6)), this lower bound is sharp for quadratic A ($\deg n = 2$).

(2) It is not clear whether the complexity of inversion (respectively, division) also has a linear upper bound. In certain cases, however, this is true. Assume there exists a tower $k = A_0 \subset A_1 \subset \cdots \subset A_s = A$ of subfields, with $A_i \supset k$ simple and the degrees $[A_i : A_{i-1}]$ bounded by some constant. Then, via computing the inverses of the norms $N_{A_{i-1}}^{A_i}$, we get a recurrence for inversion in A_i (over k) as

$$L_{\mathrm{INV}}(A_i) \leqq L_{\mathrm{INV}}(A_{i-1}) + C \cdot L_{\mathrm{MULT}}(A_{i-1})$$

for some constant C. Since multiplication in A_{i-1} (over k) has a linear upper bound [4], [10], this leads to a linear upper bound for the complexity of inversion in A (over k).

2. Quadratic alternative algebras. Let A be a finite-dimensional alternative algebra over k with identity element 1, i.e., we have the following weak associativity laws $x^2 y = x(xy)$ and $yx^2 = (yx)x$ for all $x, y \in A$. A is called *quadratic* if 1, x, x^2 are linearly dependent for all $x \in A$. In these algebras trace and norm t, $n : A \to k$ can be introduced such that for all $x \in A$

$$x^2 - t(x) + n(x) = 0;$$

t is linear, and n is a quadratic form (cf. [1, p. 37]). With the involution $x \mapsto \bar{x} = t(x) - x$, $t(x) = x + \bar{x}$, and $n(x) = x\bar{x}$. As is easily seen, $x \in A$ has a right inverse if and only if x has a left inverse if and only if right multiplication with x is regular if and only if left multiplication with x is regular if and only if $n(x) \neq 0$. Thus *the* inverse $x^{-1} = \bar{x}/n(x)$ is well defined if $n(x) \neq 0$. Let us define the index of n as $\mathrm{ind}(n) = \max\{\dim L : L \subset A$ a linear subspace, $n(L) = 0\}$. (For nondegenerate quadratic forms in char $k \neq 2$ this is the Witt index.) A has zero divisors if and only if $\mathrm{ind}(n) > 0$.

THEOREM 4. *Let A be a quadratic alternative algebra with identity element of dimension $\geqq 2$, $A \neq k \times k$. Then*

$$L_{\mathrm{INV}} = 2 \dim A - \mathrm{ind}\,(n).$$

Proof. Let $X_1, \cdots, X_m$ be indeterminates over k, $m = \dim A$. Via the choice of a basis we view n as an element of $k[\mathbf{X}]$. We have

$$L_{\mathrm{INV}} = L(X_1/n, \cdots, X_m/n),$$

as $x \mapsto \bar{x}$ is involutional, and therefore regular. Thus $L_{\mathrm{INV}} \leqq m + L(n) = 2m - \mathrm{ind}\,(n)$, by Strassen [9, § 4, Anwendung 7]. For the lower bound first we consider the case where n is reducible, $\mathrm{ind}\,(n) = m-1$. If $m=2$, then $A = k[T]/(T^2)$, and a direct calculation shows $L(X_2/X_1^2, 1/X_1^2) = 3$ ($n = X_1^2$, say). If $m \geqq 3$ and n is a square, say $n = X_1^2$, then two linearly independent linear combinations of the X_i/n have complexity $\leqq L_{\mathrm{INV}} - (m-3)$, and the assertion follows from $L(X_2/X_1^2, X_3/X_1^2) \geqq L(X_2/X_1^2, 1/X_1^2) = 3$ (replace X_3 by a multiple of X_1 in order to get this inequality). If n is not a square, say $n = X_1X_2$, then three linearly independent linear combinations of the X_i/n have complexity less than or equal to $L_{\mathrm{INV}} - (m-4)$. Here the result is obtained from $L(X_3/X_1X_2,\ X_4/X_1X_2,\ X_5/X_1X_2) \geqq L(X_3/X_1X_2,\ X_4/X_1X_2,\ 1/X_2 + \alpha/X_1) \geqq L(X_3/X_1X_2,\ 1/X_1,\ 1/X_2) > L(X_3/X_1^2,\ 1/X_1^2) = 3$, $\alpha \in k$ (here replace successively X_5, X_4, X_2 by multiples of $X_1 + \alpha X_2$, X_2, X_1, respectively, in order to get these inequalities). If $\mathrm{ind}\,(n) \leqq m-2$, a nontrivial linear combination f of the X_i/n has complexity less than or equal to $L_{\mathrm{INV}} - (m-1)$. Now, $m - \mathrm{ind}\,(n) - 2$ successive complexity decreasing adjunctions of linear forms to the ground field, and a final application of Lemma 3.2(ii) of [8] give $L_{\mathrm{INV}} - (m-1) \geqq L(f) \geqq m - \mathrm{ind}\,(n) - 2 + 3$, as desired.

Examples. (1) Null algebras $A = k[X_1, \cdots, X_{s-1}]/(\cdots, X_iX_j, \cdots)$, $L_{\mathrm{INV}} = s+1$.

(2) Quadratic extension fields A of k, $L_{\mathrm{INV}} = 4$.

(3) Quaternions $A = Q(\mu, \beta)$ (cf. [11, p. 30]). If A has no zero divisors, then n is anisotropic, so $L_{\mathrm{INV}} = 8$ (e.g., real quaternions $Q(-5/4, -1)$).

(4) 2×2 matrices $A = k^{2\times 2}$ ($= Q(0,1)$). As $\mathrm{ind}\,(\det) = 2$, $L_{\mathrm{INV}} = 6$. ($L_{\mathrm{INV}} = 5$ for the subalgebra of triangular matrices.)

(5) Cayley numbers $A = C(\mu, \beta, \gamma)$ [11]. $L_{\mathrm{INV}} = 16$ if A has no zero divisors (e.g., real Cayley numbers $C(-5/4, -1, -1)$).

(6) Char $k = 2$, a purely inseparable extension $A = k(\xi_1^{1/2}, \cdots, \xi_s^{1/2})$ with $[A:k] = 2^s$. Here $L_{\mathrm{INV}} = 2^{s+1}$.

Acknowledgment. This paper was written while the author visited Professor Volker Strassen's working group, University of Zurich. The inspiring and stimulating atmosphere is gratefully acknowledged.

REFERENCES

[1] A. ALDER AND V. STRASSEN, *On the algorithmic complexity of associative algebras*, Theoret. Comput. Sci., 15 (1981), pp. 201-211.

[2] D. BINI, *On commutativity and approximation*, Theoret. Comput. Sci., 28 (1984), pp. 135-150.

[2a] A. BORODIN AND I. MUNRO, *Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[3] D. COPPERSMITH AND S. WINOGRAD, *On the asymptotic complexity of matrix multiplication*, SIAM J. Comput., (1982), pp. 472-492.

[4] C. M. FIDUCCIA AND Y. ZALCSTEIN, *Algebras having linear multiplicative complexities*, J. Assoc. Comput. Mach., 24 (1977), pp. 311-331.

[5] B. GRIESSER, *Lower bounds for the approximative complexity*, Theoret. Comput. Sci., 46 (1986), pp. 329-338.

[6a] H. F. DE GROOTE, *On varieties of optimal algorithms for the computation of bilinear mappings*: I. *The isotropy group of a bilinear mapping*, Theoret. Comput. Sci., 7 (1978), pp. 1-24.

[6b] ———, II. *Optimal algorithms for* 2×2 *matrix multiplication*, Theoret. Comput. Sci., pp. 127-148.

[7] N. JACOBSON, *Lectures in abstract algebra*, III. *Theory of fields and galois theory*, in Graduate Texts in Mathematics 32, Springer-Verlag, Berlin, New York, 1964.

[8] T. LICKTEIG, *The computational complexity of division in quadratic extension fields*, SIAM J. Comput., 16 (1987), pp. 278-311.

[9] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 182-202.

[10] S. WINOGRAD, *Some bilinear forms whose multiplicative complexity depends on the field of constants*, Math. Systems Theory, 10 (1977), pp. 169-180.

[11] K. A. ZHEVLAKOV, A. M. SLIN'KO, I. P. SHESTAKOV, AND A. I. SHIRSHOV, *Rings That Are Nearly Associative*, Academic Press, New York, 1982.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 216-228, April 1989

ADAPTIVE BITONIC SORTING: AN OPTIMAL PARALLEL ALGORITHM FOR SHARED-MEMORY MACHINES*

GIANFRANCO BILARDI† AND ALEXANDRU NICOLAU‡

Abstract. A parallel algorithm, called *adaptive bitonic sorting*, that runs on a PRAC (parallel random access computer), a shared-memory multiprocessor where fetch and store conflicts are disallowed, is proposed. On a P processors PRAC, the algorithm presented here achieves optimal performance $TP = O(N \log N)$, for any computation time T in the range $\Omega(\log^2 N) \leqq T \leqq O(N \log N)$. Adaptive bitonic sorting also has a small constant factor, since it performs less than $2N \log N$ comparisons, and only a handful of operations per comparison.

Key words. sorting, parallel computation, shared-memory machines, bitonic sequence, time × processors optimality

AMS(MOS) subject classifications. 68Q20, 68Q25, 68Q10

1. Introduction.

Bitonic sorting [Ba68] is an interesting parallel algorithm based on a merge-sort scheme and an ingenious merging technique known as bitonic merging. Orginally proposed for a network of comparators, bitonic sorting has been considered for implementation on a variety of architectures such as the shuffle-exchange [St71], the binary cube [Pe77], the mesh [TK77], [NS79], the cube-connected cycles [PV81] and its pleated version [BP84], and on array processors [St78]. Various properties of bitonic networks have been investigated (e.g., [Kn73], [HS82], [Pr83], [Bl85], [Bl88]).

On an input of N elements, the bitonic merger performs $\Theta(N \log N)$ operations (comparisons and exchanges), coming to within a constant factor of a lower bound for merging networks of comparators that is due to Floyd [Kn73, p. 230]. However, in other models of parallel computation merging can be done with $O(N)$ operations [BH85], and therefore the bitonic algorithm is not optimal in these models.

In this paper we present procedures for merging and sorting which we propose to call *adaptive bitonic algorithms*. Our algorithms are based on bitonic sequences, as are Batcher's, but unlike Batcher's they perform a set of comparisons that is a function of the input values. As a result, our approach cannot be used in the context of a network of comparators, but will be shown to be more efficient than Batcher's, in terms of the number of operations performed, on a general purpose shared-memory machine. When necessary to avoid confusion, we shall refer to Batcher's algorithm as the nonadaptive or the network algorithm.

Adaptive bitonic merging [sorting] can be implemented on a PRAC (parallel random access computer) of P processors in time $T = O(N/P)[T = O((N \log N)/P)]$, for $1 \leqq P \leqq N/2^{\lfloor \log \log N \rfloor}$. The PRAC [LPV81] is a shared-memory multiprocessor of the EREW-PRAM variety [Sn85], where simultaneous access of the same memory location is disallowed. The product TP is optimal for both merging and sorting.

* Received by the editors October 17, 1986; accepted for publication (in revised form) April 19, 1988. A preliminary version of this paper was presented at the 20th Annual Conference on Information Systems and Sciences, Princeton, New Jersey, March 1986.

† Department of Computer Science, Cornell University, Ithaca, New York 14853. The work of this author was supported in part by National Science Foundation grant DCI-8602256.

‡ Department of Computer Science, Cornell University, Ithaca, New York 14853. The work of this author was supported in part by National Science Foundation grant DCR-8502884, and the Cornell National Science Foundation Supercomputing Center.

The merging algorithm also achieves the $T = \Omega(\log N)$ lower bound established in [Sn85] for merging on a PRAC. To our knowledge, ours is the first algorithm that attains the minimum time $T = O(\log N)$ with an optimal number of processors $P = O(N/\log N)$, on the PRAC model. An algorithm recently proposed in [AS85] achieves an optimal TP product, but for $T = \Omega(\log^2 N)$.

As for sorting on the PRAC, the question of asymptotic complexity was settled by the network of [AKS83], which is logspace uniform [L85], and therefore yields a PRAC algorithm with $TP = O(N \log N)$ for $T = \Omega(\log N)$. Very recently, the same asymptotic performance has been achieved by an adaptive algorithm proposed in [C86], building on the previous work of [P78] and [Kr83]. Our algorithm also achieves an optimal rate of growth for the TP measure, for $T = \Omega(\log^2 N)$, with a much smaller constant factor than in the AKS network, and probably smaller than the algorithm in [C86]: the latter algorithm performs less than $5N \log N$ comparisons ($2.5N \log N$ if concurrent reads are allowed). Furthermore, the number of other operations performed by that algorithm is significant [C87] and has not been precisely analyzed.

Adaptive bitonic sorting performs only about twice as many operations as the fastest sequential sorting algorithms, even for time $T = O(\log^2 N)$. It can obviously be implemented, with the same performance, on other shared-memory models with less restrictive memory access mechanisms. For results on the complexity of merging and sorting on some of these models see, for example, [BH85], [HH81], [Kr83], [SV81], [V75].

In the bitonic merging network, both the number of comparisons and the number of exchanges are $O(N \log N)$. Adaptive bitonic merging achieves a reduction of both numbers to $O(N)$, based on two properties established in the paper. First, there exists a subset of less than $2N$ of the comparisons performed by the network sufficient to determine the result of all the others. Second, there is a regularity in the pattern of exchanges that can be exploited by using a data structure, which we call *the bitonic tree*, whereby many element exchanges can be accomplished by a small number of subtree (i.e., pointers) exchanges. The properties of bitonic sequences exploited by our algorithm are discussed in § 2. The adaptive bitonic-merging algorithm is developed in § 3, where a sequential model is adopted for simplicity.

Parallel versions of merging and sorting are described in § 4. Here the main difficulty consists in avoiding a loss in time performance with respect to the network algorithm. In fact, in the PRAC implementation, adaptive bitonic merging emulates the kth stage of comparisons of the merging network in time $O(\log N - k)$, for $k = 0, 1, \cdots, \log N - 1$. If the stages are executed in sequence, the resulting time is $O(\log^2 N)$. However, a careful analysis of data-dependencies shows that $O(\log N)$ time can be achieved by a suitable overlapping of the stages.

The adaptive algorithms of §§ 3 and 4, as well as the network algorithms of [Ba68], assume that the length of the input sequence N is a power of two. The obvious strategy of padding the input sequence so that its length becomes a power of two leads to an increase of the complexity by a constant factor. In § 5 we show how this increase can be avoided by developing a variant of the algorithm that works for arbitrary N.

Besides attaining an optimal rate of growth, the performance of the algorithms presented here also exhibits very small constant factors. Adaptive bitonic sorting performs less than $2N \log N$ comparisons, independently of the number of processors, and appears to be attractive for practical implementation. Some indication of the practical behaviour is given in § 6.

2. Properties of bitonic sequences. Let $\mathbf{x} = (x_0, \cdots, x_{N-1})$ be a sequence of N (hereafter N is assumed even) elements from a totally ordered set. We introduce the

following operators on $\mathbf{x}$:

$$S_j\mathbf{x} \triangleq (x_{j \bmod N}, x_{(j+1) \bmod N}, \cdots, x_{(j+N-1) \bmod N}), \tag{1}$$

$$L\mathbf{x} \triangleq (\min\{x_0, x_{N/2}\}, \cdots, \min\{x_{N/2-1}, x_{N-1}\}), \tag{2}$$

$$U\mathbf{x} \triangleq (\max\{x_0, x_{N/2}\}, \cdots, \max\{x_{N/2-1}, x_{N-1}\}). \tag{3}$$

A sequence $\mathbf{x}$ is *bitonic* if, for some j, the sequence $S_j\mathbf{x} = (y_0, \cdots, y_{N-1})$ consists of a nondecreasing portion followed by a nonincreasing one. Bitonic merging is based on the fact that, if $\mathbf{x}$ is bitonic,

$$\text{sort}(\mathbf{x}) = (\text{sort}(L\mathbf{x}), \text{sort}(U\mathbf{x})). \tag{4}$$

This relation, due to [Ba68], leads to a recursive algorithm whose complexity is determined by that of computing $L\mathbf{x}$ and $U\mathbf{x}$. Theorem 1 below states four properties of L and U on bitonic operands. Property (P1) is a lemma for the others. Properties (P2) and (P3), obtained by Batcher, imply (4) above. Relation (P4) is crucial here since it provides the basis for a method to compute $L\mathbf{x}$ and $U\mathbf{x}$ that is more efficient than the direct application of definitions (2) and (3) above, which are used in the bitonic algorithm in [Ba68].

THEOREM 1. *If $\mathbf{x}$ is a bitonic sequence (of even length), then the following properties hold:*

(P1) *There is a shifted version $S_q\mathbf{x} = (z_0, \cdots, z_{N-1})$ of $\mathbf{x}$ such that*

$$L\mathbf{x} = S_{(-q \bmod N/2)}(z_0, \cdots, z_{N/2-1}), \tag{5}$$

$$U\mathbf{x} = S_{(-q \bmod N/2)}(z_{N/2}, \cdots, z_{N-1}). \tag{6}$$

(P2) *Each element of $U\mathbf{x}$ is no smaller than each element of $L\mathbf{x}$.*

(P3) *Sequences $L\mathbf{x}$ and $U\mathbf{x}$ are bitonic.*

(P4) *Let q be as in* (P1) *and let $t = q \bmod N/2$. Let $\mathbf{x} = (\mathbf{x}', \mathbf{x}'', \mathbf{x}''', \mathbf{x}'''')$ with $\mathbf{x}'$ and $\mathbf{x}'''$ of length t, and $\mathbf{x}''$ and $\mathbf{x}''''$ of length $N/2 - t$. If $q < N/2$ $(t = q)$, then*

$$(L\mathbf{x}, U\mathbf{x}) = (\mathbf{x}''', \mathbf{x}'', \mathbf{x}', \mathbf{x}''''). \tag{7}$$

If $q \geqq N/2$ $(t = q - N/2)$, then

$$(L\mathbf{x}, U\mathbf{x}) = (\mathbf{x}', \mathbf{x}'''', \mathbf{x}''', \mathbf{x}''). \tag{8}$$

Before proving Theorem 1, we show two relations among the operators L, U, and S_j.

LEMMA 1. *For any $\mathbf{x}$ and j,*

$$L\mathbf{x} = S_{(-j \bmod N/2)}LS_j\mathbf{x} \tag{9}$$

$$U\mathbf{x} = S_{(-j \bmod N/2)}US_j\mathbf{x}. \tag{10}$$

Proof. We prove only (9), the argument for (10) being similar. For $j = N/2$, (9) becomes $L\mathbf{x} = LS_{N/2}\mathbf{x}$, which can be trivially verified. Thus, index j in (9) can always be taken modulo $N/2$ and it is sufficient to consider $j < N/2$. In this case we have $-j \bmod N/2 = N/2 - j$, and

$$LS_j\mathbf{x} = (\min\{x_j, x_{j+N/2}\}, \cdots, \min\{x_{N/2-1}, x_{N-1}\}, \min\{x_0, x_{N/2}\}, \cdots, \min\{x_{j-1}, x_{j-1+N/2}\}),$$

or, in compact form, $LS_j\mathbf{x} = S_jL\mathbf{x}$, from which (9) follows by applying $S_{-j \bmod N/2}$ to both sides. □

Proof of Theorem 1. Let the *median* of a sequence $\mathbf{x} = (x_0, \cdots, x_{N-1})$ be defined as the minimum value m such that less than $N/2$ elements of $\mathbf{x}$ are greater than m. Let $\mathbf{x}$ be bitonic, and let $\mathbf{y} = S_j\mathbf{x} = (y_0, \cdots, y_{N-1})$ consist of a nondecreasing sequence followed by a nonincreasing sequence. In general, $\mathbf{y}$ is the concatenation of five (possibly empty) subsequences, respectively, containing N_1 elements smaller than m, N_2 elements equal to m, N_3 elements larger than m, N_4 elements equal to m, and N_5 elements smaller than m.

Obviously, $N_1 + \cdots + N_5 = N$. Also, by the definition of m, $N_3 < N/2$ and $N_2 + N_3 + N_4 \geqq N/2$. By simple arithmetic, $N_1 \leqq N/2 - N_5$ and there exists a k, with $N_1 \leqq k \leqq N/2 - N_5$, such that the sequence $(y_k, y_{k+1}, \cdots, y_{k+N/2-1})$ contains all the elements larger than m and none of those smaller than m.

We now consider the sequence $\mathbf{z} = S_{k+N/2}\mathbf{y} = S_q\mathbf{x}$, where $q = (j + k + N/2) \bmod N$. If we write $\mathbf{z} = (\mathbf{z}', \mathbf{z}'')$, with $\mathbf{z}'$ and $\mathbf{z}''$ sequences of $N/2$ elements, it is easy to see that $\mathbf{z}'' = (y_k, \cdots, y_{k+N/2-1})$.

Thus all the elements of $\mathbf{z}''$ are no smaller than the elements of $\mathbf{z}'$, which implies that $(L\mathbf{z}, U\mathbf{z}) = \mathbf{z}$. Lemma 1 applied to the latter relation yields $L\mathbf{x} = S_{(-q \bmod N/2)}\mathbf{z}'$ and $U\mathbf{x} = S_{(-q \bmod N/2)}\mathbf{z}''$, which are equivalent to (5) and (6) and establish (P1).

Property (P2) follows from (5), (6), and the fact that $\max\{z_0, \cdots, z_{N/2-1}\} \leqq \min\{z_{N/2}, \cdots, z_{N-1}\}$.

Sequences $\mathbf{z}'$ and $\mathbf{z}''$ are bitonic since they are subsequences of $\mathbf{z}$, which is bitonic. Then from (5) and (6), $L\mathbf{x}$ and $U\mathbf{x}$ are shifts of bitonic sequences, and (P3) is proved.

If $q < N/2$ and $\mathbf{x} = (\mathbf{x}', \mathbf{x}'', \mathbf{x}''', \mathbf{x}'''')$, as defined in (P4), then $S_q\mathbf{x} = (\mathbf{x}'', \mathbf{x}''', \mathbf{x}'''', \mathbf{x}')$. From (6) and (7), cyclically shifting each half of $S_q\mathbf{x}$ q positions to the right, we obtain $L\mathbf{x} = (\mathbf{x}''', \mathbf{x}'')$ and $U\mathbf{x} = (\mathbf{x}', \mathbf{x}'''')$. This proves (7). A similar argument yields (8), completing the proof of (P4). □

3. Adaptive bitonic merging with $O(N)$ comparisons. In this section we present a linear-time version of bitonic merging. For simplicity, we describe the algorithm in a sequential setting and we assume that N, the sum of the lengths of the sequences being merged, is a power of two. Parallelism and arbitrary input size are discussed in subsequent sections.

3.1. Analysis of comparisons. Let $\mathbf{x}$ be the bitonic sequence obtained by concatenating, in opposite order, two sorted sequences to be merged. The classical bitonic merging consists of the following steps:

(1) Compute $L\mathbf{x}$ and $U\mathbf{x}$ by $N/2$ (parallel) comparisons (according to definitions (2) and (3)).

(2) Recursively sort $L\mathbf{x}$ and $U\mathbf{x}$, in parallel.

The comparisons performed by the above algorithm are data-independent and hence can be hardwired in a network of comparator-exchangers, *the bitonic merger.* In the terminology of [Kn73], the bitonic merger has N (a power of two) lines numbered $0, 1, \cdots, N-1$ and $\log N$ stages each of $N/2$ comparator-exchangers. In the kth stage ($k = 0, 1, \cdots, \log N - 1$) lines i and j are connected by a comparator-exchanger if and only if the binary spellings of i and j differ exactly in the $(\log N - k)$th rightmost bit. Comparators output the smaller of the two inputs on the line with lower number.

Since, as is well known, N comparisons are sufficient to merge two sequences, the set C of the $(N \log N)/2$ comparisons executed by the bitonic merger obviously contains some redundancy. Less obviously, this redundancy can be almost eliminated by executing only a suitable subset of C, as shown in the next theorem. As it will become clear from the proof, this subset is a function of the input sequence $\mathbf{x}$.

THEOREM 2. *Let C be the set of the $(N \log N)/2$ comparisons executed by Batcher's merging network. Then, there is a subset $C' \subset C$ of size $|C'| = 2N - \log N - 2$ such that the results of the comparisons in C are uniquely determined by the results of the comparisons in C', as long as the input elements are distinct.*

Proof. Due to property (P4) established in Theorem 1, there is a decomposition $(\mathbf{x}', \mathbf{x}'', \mathbf{x}''', \mathbf{x}'''')$ of $\mathbf{x}$ such that $(L\mathbf{x}, U\mathbf{x})$ is obtained either by exchanging $\mathbf{x}'$ and $\mathbf{x}'''$ (7) or by exchanging $\mathbf{x}''$ and $\mathbf{x}''''$ (8). Due to (P2), if $x_{N/2-1} < x_{N-1}$ then (7) holds, else $(x_{N/2-1} > x_{N-1})$ (8) holds.

When (7) holds, $\mathbf{x}''$ contains x_i if and only if $x_i < x_{N/2+i}$. Thus the leftmost element of $\mathbf{x}''$ can be found by a binary search driven by comparisons of pairs of the form $(x_i, x_{N/2+i})$. Index i is first set to $N/4-1$, and then decremented if $x_i < x_{N/2+i}$ and incremented otherwise. The increment is initialized to $N/8$, and halved at each step. The search terminates after $\log N - 1$ steps, when the increment is zero. Therefore $\mathbf{x}''$, and thus $(L\mathbf{x}, U\mathbf{x})$, is determined by $\log N$ of the $N/2$ comparisons implicit in definitions (2) and (3). The case when (8) holds is completely symmetric.

Let C' be the set of comparisons resulting from recursively applying the above method to $L\mathbf{x}$ and $U\mathbf{x}$. Considering that at the kth level of recursion $(k = 0, 1, \cdots, \log N - 1)$ there are 2^k sequences each requiring $\log N - k$ comparisons, we can evaluate the cardinality of C' as follows:

$$M(N) \triangleq |C'| = \sum_{k=0}^{\log N - 1} 2^k (\log N - k) = 2N - \log N - 2. \qquad (11)$$

□

The assumption of distinct input elements is essential in Theorem 2. An arbitrary strategy for resolving ties may lead to incorrect behaviour. As an example, consider the bitonic sequences $\mathbf{u} = (0, 0, -1, 0, 0, 0, 0, 0)$ and $\mathbf{v} = (0, 0, 1, 0, 0, 0, 0, 0)$. It can be easily seen that if the case $x_i = x_{N/2+i}$ is treated as $x_i < x_{N/2+i}$, then $\mathbf{u}$ will be sorted properly, but $\mathbf{v}$ will not. The opposite will happen if the case $x_i = x_{N/2+i}$ is treated as $x_i > x_{N/2+i}$.

A simple and correct strategy consists of breaking ties (when comparing equal elements) according to the original position of the elements in the input sequence, which amounts to enforcing distinctness. The resulting algorithm clearly will be stable. In the remainder of the paper, distinct elements will be assumed.

3.2. The data structure. Although Theorem 2 gives a way to obtain all the information needed to merge two sequences with a linear number of comparisons, the problem remains of how to efficiently achieve the data rearrangement that in Batcher's network requires $\Omega(N \log N)$ exchanges in the worst case. We solve this problem with the adoption of a suitable data structure, which we call a bitonic tree.

A *bitonic tree* (see Fig. 1) is a binary tree where each node contains an element from a totally ordered set, and the sequence of elements encountered in the inorder traversal of the tree is bitonic. The bitonic tree is a simple generalization of the binary search-tree. In fact, an inorder traversal of the latter yields a monotonic (sorted) sequence.

Given a bitonic sequence of length N (a power of two), we adopt a representation in which the first $N-1$ elements are stored in a fully balanced bitonic tree of $\log N$ levels, and the last element is kept in a spare node. In this representation, the decomposition $\mathbf{x} = (\mathbf{x}', \mathbf{x}'', \mathbf{x}''', \mathbf{x}'''')$ considered in Theorem 1 corresponds to a decomposition of the bitonic tree into four forests (see Fig. 1). The roots of the trees in these forests form two parallel paths in the main subtrees of the bitonic tree. The exchange

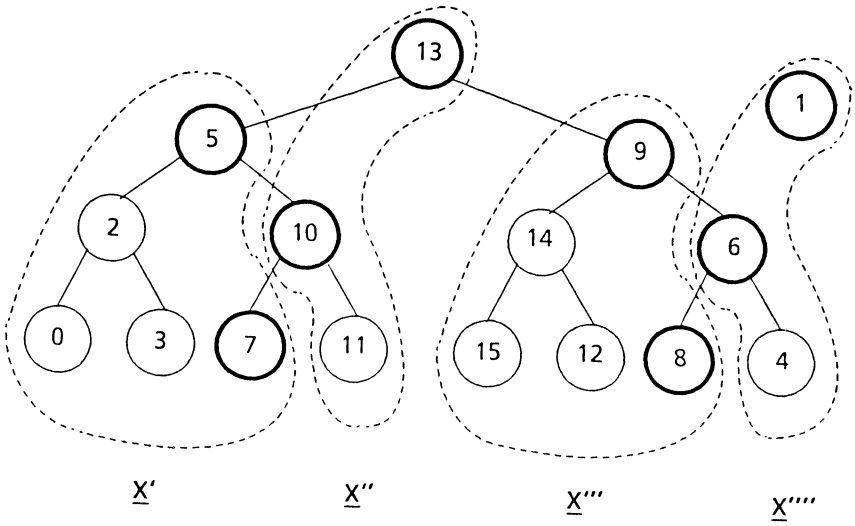


FIG. 1. *Bitonic tree. Bitonic tree for the sequence* $\mathbf{x} = (0, 2, 3, 5, 7, 10, 11, 13, 15, 14, 12, 9, 8, 6, 4, 1)$ *and the decomposition* $(\mathbf{x}, \mathbf{x}'', \mathbf{x}''', \mathbf{x}'''')$ *for the computation of* $(L\mathbf{x}, U\mathbf{x}) = (\mathbf{x}', \mathbf{x}'''', \mathbf{x}''', \mathbf{x}'')$. *Solid nodes are the ones examined by the binary search.*

of $\mathbf{x}'$ and $\mathbf{x}'''$ [$\mathbf{x}''$ and $\mathbf{x}''''$] required by (7) [(8)] to compute $(L\mathbf{x}, U\mathbf{x})$ can be accomplished with $O(\log N)$ exchanges of values and pointers in the bitonic tree.

The relation between the bitonic tree and the bitonic network can be viewed as a one-to-one correspondence between nodes and lines, the ith line being associated with the ith node encountered in the inorder traversal of the tree. However, a node and the corresponding line are guaranteed to hold the same value only at the beginning of merging and immediately after the completion of the computation of the sequences $(L\mathbf{x}, U\mathbf{x})$ at each level of the recursion.

3.3. The algorithm. Based on the preceding observations we give below a procedure bimerge(root, spare, dir) that sorts $\mathbf{x}$ by transforming the bitonic tree into a binary search-tree. Parameters root and spare are pointers to the root of the tree and to the spare node, respectively. Parameter dir is Boolean and represents the direction in which the sequence is to be sorted (dir = false for ascending, dir = true for descending).

Each node of the tree has three fields, value, left, and right, respectively storing an element of the sequence (or a pointer to it) and pointers to the left and right subtrees. The procedure bimerge given in Fig. 2 is written in pseudo-Pascal. The identifiers not explicitly defined are self-explanatory.

We now briefly comment on procedure bimerge. At the beginning, the root contains $x_{N/2-1}$ and the spare node contains x_{N-1}. After statement 1. of Fig. 2, rightexchange is false when (7) holds (i.e., $\mathbf{x}'$ and $\mathbf{x}'''$ are exchanged), and true when (8) holds (i.e., $\mathbf{x}''$ and $\mathbf{x}''''$ are exchanged). In the latter case, $x_{N/2-1}$ and x_{N-1} are exchanged (by statement 2.). After statement 3., *pl* and *pr* point to the nodes that contain $x_{N/4-1}$ and $x_{3N/4-1}$, respectively.

The binary search for the boundary between $\mathbf{x}'$ and $\mathbf{x}''$ (as well as between $\mathbf{x}''$ and $\mathbf{x}''''$) is performed by the while-loop (statements 4. ÷ 19.). At the end of the loop, an inorder traversal of the tree would yield $(L\mathbf{x}, U\mathbf{x})$. Sequences $L\mathbf{x}$ and $U\mathbf{x}$ are recursively

```
    procedure bimerge (root, spare, dir);
    begin
 1. rightexchange := (root↑value > spare↑value) XOR dir;
 2. if rightexchange then swap-value(root, spare);
 3. pl := root↑left; pr := root↑right;
 4. while (pl ≠ nil) do begin
 5.   elementexchange := (pl↑value > pr↑value) XOR dir;
 6.   if rightexchange then              /* X'' and X'''' exchange          */
 7.     if elementexchange then begin    /* swap values and right subtrees;
                                           search path goes left           */
 8.       swap-value(pl, pr);
 9.       swap-right(pl, pr);
10.       pl := pl↑left; pr := pr↑left
          end
11.     else begin                       /* search path goes right          */
12.       pl := pl↑right; pr := pr↑right
          end
13.   else                               /* X' and X''' exchange            */
14.     if elementexchange then begin    /* swap values and left subtrees;
                                           search path goes right          */
15.       swap-value(pl, pr);
16.       swap-left(pl, pr);
17.       pl := pl↑right; pr := pr↑right
          end
18.     else begin                       /* search path goes left           */
19.       pl := pl↑left; pr := pr↑left
          end
    end; /* while */
20. if (root↑left ≠ nil) then begin
21.   bimerge(root↑left, root, dir);
22.   bimerge(root↑right, spare, dir)
      end
    end; /* bimerge*/
```

FIG. 2. *Procedure Bimerge.*

sorted by the recursive calls in statements 21. and 22. We observe that $L\mathbf{x}$ and $U\mathbf{x}$ are represented, as is $\mathbf{x}$, by a bitonic tree and a spare node: $L\mathbf{x}$ by the left subtree and the root, $U\mathbf{x}$ by the right subtree and the original spare node. In general, the recursive calls of depth k (the first call being of depth zero) operate on sequences of length $N/2^k$. These sequences are represented by a subtree with root at level k (the root of the entire tree being at level zero) and a spare node. The spare node belongs to some level less than k or is the spare node of the original tree. More precisely, the ith subtree at level k, from left to right, is paired with the ith node encountered in the inorder traversal of the first k levels of the tree.

A simple analysis of bimerge shows that the total number of operations is of the same order as the number of comparisons. Thus, the algorithm runs in linear-time.

4. Parallel algorithms. In this section we present a parallel version of adaptive bitonic merging and a parallel sorting algorithm based on it. As a model of computation, we choose the PRAC of [LPV81]. The PRAC is a shared-memory multiprocessor machine. Any processor can access any common-memory location in constant time. However, simultaneous access (either read or write) of the same location is illegal and leads to termination error. For more details on the PRAC see [LPV81].

4.1. Merging. We call stage (k) of the merge $(k = 0, 1, \cdots, \log N - 1)$ the set of recursive calls of bimerge of depth k. There are 2^k such calls, each processing a different subsequence of $N/2^k$ elements. As already observed, this subsequence occupies a subtree with root at level k and a spare node in some level less than k. All the calls in stage (k) can be executed in parallel.

We call phase (0) of bimerge the execution of statements 1., 2., and 3. of Fig. 2. We call phase (i) the execution of the ith iteration of the while-loop. For a call in stage (k), i ranges from 1 to $\log N - k - 1$. A phase includes one comparison and a handful of tests and assignment statements. It can be executed in $O(1)$ time.

Since calls of depth k consist of $\log N - k$ phases, stage (k) takes $\log N - k$ parallel phases. The total time for executing stage $(0), \cdots,$ stage $(\log N - 1)$ in sequence is $O(\sum_{k=0}^{\log N - 1} (\log N - k)) = O(\log^2 N)$. We note a loss in performance with respect to the $O(\log N)$ time of Batcher's network. The loss is due to the difference in the computation of $(L\mathbf{x}, U\mathbf{x})$. For $\mathbf{x}$ of length N the $N/2$ comparisons of Batcher's network are data-independent and take only one time step, while the $\log N$ comparisons of our binary-search method are inherently sequential and take $\log N$ steps.

Little can be done to speed up the execution of a single stage without increasing the number of comparisons. However, a careful analysis of the data dependencies between comparisons of the various stages reveals that the execution of different stages can be partially overlapped, with considerable savings in running time.

We observe that, in stage (k), phase (0) operates on levels $0, 1, \cdots, k$ of the bitonic tree, and phase (i) operates on level $k + i$ $(i = 1, \cdots, \log N - k - 1)$. Thus, phase (0) of stage (k) can begin as soon as stage $(0), \cdots,$ stage $(k - 1)$ have processed the first k levels of the tree. In general, phase (i) of stage (k) can begin as soon as stage $(0), \cdots,$ stage (k) have processed the first $k + i$ levels of the tree. This condition is satisfied if a new stage is scheduled to begin every other phase step. The entire sequence of $\log N$ stages is completed in $2 \log N - 1$ phase steps, that is in $O(\log N)$ time. An example of the schedule of the phases of different stages is given in Table 1, for $N = 16$.

TABLE 1
Schedule for the overlapped execution of the stages of bimerge for $N = 16$.

Phase step	Stage (0)		Stage (1)		Stage (2)		Stage (3)	
	Phase	Tree levels	Phase	Tree levels	Phase	Tree levels	Phase	Tree levels
0	0	0	-	-	-	-	-	-
1	1	1	-	-	-	-	-	-
2	2	2	0	0, 1	-	-	-	-
3	3	3	1	2	-	-	-	-
4	-	-	2	3	0	0, 1, 2	-	-
5	-	-	-	-	1	3	-	-
6	-	-	-	-	-	-	0	0, 1, 2, 3

In the above schedule, the maximum number of calls simultaneously active is $N/2$, and hence $N/2$ processors are sufficient. We now consider a slightly different schedule that leads to a reduction of the number of processors without substantial degradation in time performance.

Let us assume a number of processors $P = 2^p$, with $P > 1$ and consider the following strategy:

(1) Stage $(0), \cdots,$ stage $(p-1)$ are scheduled to begin one every other step. The total number of calls in these stages is $P-1$ so that a different processor is available for each call. Stage (0) takes $\log N$ phase steps, after which a new stage terminates at each phase step. Thus, the first p stages take $t_1 = \log N + p - 1$ phase steps.
(2) For the remaining stages, one processor is assigned to each of the P subtrees corresponding to subsequences of length N/P. This will take $t_2 = 2N/P - \log(N/P) - 2$ phase steps (see (11)).

The total number of phase steps is $t = t_1 + t_2 = 2N/P - 2p - 3$.

If we choose $P = N/2^{\lfloor \log \log N \rfloor} \approx N/\log N$, we obtain $t = 2 \log N - 2 * 2^{\lfloor \log \log N \rfloor} - 2\lfloor \log \log N \rfloor - 2 < 4 \log N$. In conclusion, $T = O(t) = O(\log N)$, with $P = O(N/\log N)$ processors. The product TP is optimal, since N operations are necessary to merge. A similar result is obtained for $1 \leqq P \leqq N/2^{\lfloor \log \log N \rfloor}$, as summarized in the following theorem.

THEOREM 3. *Adaptive bitonic merge can be executed on a* PRAC *of P processors in time* $T = O(N/P)$, *for* $1 \leqq P \leqq N/2^{\lfloor \log \log N \rfloor}$.

It is of interest to estimate the overhead incurred to distribute the algorithm among P processors. Two aspects contribute to such overhead: (a) according to the schedule outlined above not all the processors are active all the time; and (b) some time will be spent in synchronizing the activities of different processors. Let us analyze each aspect in turn.

We can rewrite the expression for the number of phase steps t as $tP < 2N + 2pP$. Since $2N$ is essentially the number of phase steps for the sequential algorithm, the overhead due to idle processors is given by the term $2pP$, which for $P = N/2^{\lfloor \log \log N \rfloor}$ is about $2N$, and is smaller for smaller P's.

In the parallel version of bimerge, the parameters root, spare, and dir are computed by a processor different from the one that actually executes the call. Thus, the processor that computes the parameters must write them in the shared memory from which the processor assigned to the call will read them. A simple analysis shows that the total number of memory accesses for interprocessor communication is $O(P)$, so that the synchronization overhead contributes a lower-order term to the total number of operations of the merging algorithm.

4.2. Sorting. Essentially, using the classical sort-by-merge scheme (see Fig. 3) and standard manipulations, Theorem 2 leads to the following result.

```
    procedure bisort(root, spare, dir);
    begin
1.    if (root↑left = nil) then
          test-and-swap (root, spare, dir)   /* down at leaves */
2.    else begin
3.       bisort(root↑left, root, dir);
4.       bisort(root↑right, spare, ~dir);
5.       bimerge(root, spare, dir)
6.       end
    end;
```

FIG. 3. *Procedure Bisort.*

THEOREM 4. *Let S be the set of the $N \log N(\log N+1)/2$ comparisons executed by Batcher's sorting network. Then, there exists a subset $S' \subset S$ of size*

$$S(N) \triangleq |S'| = 2N \log N - 4N + \log N + 4 \tag{12}$$

such that the results of the comparisons in S are uniquely determined by the results of the comparisons in S'.

An efficient parallel version is provided by the following schedule. First, each processor sequentially sorts N/P elements, in $2(N/P)\log(N/P)$ phases. Then, for each $j = 1, 2, \cdots, \log P$, there will be a merging step where the P processors are partitioned in $P/2^j$ groups of size 2^j, with each group merging two sequences into one of length $(N/P)2^j$. The jth merging runs in less than $2(N/P)+2j$ phases. Thus, the total number of phases, t, for the entire sorting process is:

$$t < 2\left(\frac{N}{P}\right)\log\left(\frac{N}{P}\right) + \sum_{j=1}^{\log P}\left(2\left(\frac{N}{P}\right)+2j\right) = 2\left(\frac{N}{P}\right)\log N + (\log P+1)\log P.$$

Therefore,

$$tP < 2N \log N + P(\log P+1)\log P,$$

and we obtain the following result.

THEOREM 5. *Adaptive bitonic sorting can be implemented on a* PRAC *of P processors in time $T = O((N \log N)/P)$, for $1 \leqq P \leqq N/2^{\lfloor \log \log N \rfloor}$.*

We observe that $|S'|$ is within a factor of two of $\lceil \log N! \rceil$, which is a lower bound on the number of comparisons needed to sort N elements [Kn73]. It is remarkable that sorting can be done in $O(\log^2 N)$ time with so little redundance. An analysis similar to the one developed for merging shows that the total number of memory accesses for interprocessor communication (synchronization) is $O(P \log N)$, contributing a lower-order term to the total number of operations of our algorithm. The overhead due to idle processors is about $P \log^2 P$, which for $P = N/2^{\lfloor \log \log N \rfloor}$ is about $N \log N$, i.e., 50 percent of the sequential work. For smaller values of P, the overhead is obviously smaller.

5. Input sequence of arbitrary length. In the previous sections, it has been assumed that N, the number of elements to be sorted, is a power of two. The algorithms so derived can be used for any input sequence after adding enough dummy elements to it so that the length becomes a power of two. However, this strategy leads to a constant factor increase in time complexity, which is undesirable in practical applications.

In this section we modify our algorithm to handle arbitrary values of N, with a negligible increase in complexity with respect to the case where N is a power of two. The basic idea consists in simulating the actions that the power-of-two version of the algorithm would perform on the input sequence padded with dummies, while avoiding representing and processing most of the dummies.

5.1. Padding the input sequence. Let $n \triangleq \lceil \log N \rceil$, that is, let 2^n be the minimum power of two no less than N. Given a sequence $\mathbf{x} = (x_0, \cdots, x_{N-1})$ to be sorted, we augment it to obtain a sequence $\mathbf{z} = (d, \cdots, d, x_0, \cdots, x_{N-1})$ of length 2^n by inserting, at the beginning of $\mathbf{x}$, $D \triangleq 2^n - N$ dummy elements of value $d < x_i$, for all x_i's.

If $\mathbf{z}$ is stored in a tree according to the inorder traversal, the dummy elements occupy a left subforest of the tree. More precisely, let $D = \sum_{i=0}^{n-1} D_i 2^i$, with $D_i \in \{0, 1\}$. Let π be the path in the tree that starts at the root and whose ith edge goes left when $D_{n-i} = 0$, and goes right when $D_{n-i} = 1$, for $i = 1, 2, \cdots, n$. Then, the D dummy elements

occupy the nodes of π that are followed by a right edge and the left subtrees of such nodes. In other words, for each $D_i = 1$, there is a dummy subtree of height i, whose root is at level $n-i$ and is the left son of a node on π, also containing a dummy.

We now analyze the behaviour of our algorithm on a padded sequence $\mathbf{z}$ focusing in particular on the dummy subtrees.

For a given call to procedure bimerge, consider the "parallel" paths p' and p'' traced in the bitonic tree by the search for the boundary between $\mathbf{x}'$ and $\mathbf{x}''$ (see § 3.1). A given subtree T' can be modified by the call only if it is traversed by any of the two paths.

If p' and p'' originate at the root v' of T' or at a descendent of it, then only elements inside T' will be rearranged. In particular, if T' was originally a tree of dummies, it will remain such.

If p' and p'' originate at an ancestor of v', then only one path, say p', can traverse T', and v' is the first node of T' to be visited. During the traversal, v' will be compared with some node v'', root of a subtree T''. Paths p' and p'' continue in T' and T'', respectively. If T' is a tree of dummies, its elements form a consecutive run in the sorted output. Thus, the comparison of any node of p' with the corresponding node of p'' gives the same result as the comparisons of v' and v''. Therefore, subtrees T' and T'' will either be exchanged completely, or left in their positions.

5.2. Pruning the tree. The above discussion shows that dummy subtrees are left intact throughout the algorithm, and that they can be processed by examining only their root. This suggests a modification to the bitonic tree whereby a dummy subtree is represented by a single node (its root) with left and right pointers set to *nil*. We refer to the resulting data structure as the *pruned bitonic tree.*

The procedures bisort and bimerge have to be modified to work correctly on the pruned version of the tree. The necessary changes are simple, and are outlined below:

(1) The call bisort(root, spare, dir) is not executed whenever root is a node with dummy value and nil pointers.
(2) The call bimerge(root, spare, dir) is not executed whenever root is a node with dummy value and nil pointers.
(3) Whenever a node v', with dummy value and nil pointers is compared with another node v'', and v' and v'' are exchanged, their left and right pointers are also exchanged. In any case, the current call to bimerge is terminated after the comparison.

5.3. Analysis. Let us consider a dummy subtree of height i ($i \leqq n-2$). The savings in comparisons coming from each of the modifications to the algorithm described above are as follows:

(1) $(S(2^i)-1)$ for replacing a call to bisort with a single comparison between the root of the subtree and a spare node.
(2) $(M(2^i)-1)(n-i)$ for replacing each of the $(n-i)$ calls to bimerge with a single comparison between root and spare node.
(3) $(i-1)v_i$ for not traversing the $(i-1)$ levels below the root v' for each of the v_i times in which the dummy subtree should be traversed by a path originating above v'. In general, v_i is a function of the input sequence, but always satisfies the simple bound $v_i \leqq (n-i)(n-i+1)/2$, the latter being the number of calls to bimerge with an ancestor of v' as the root.

We can now estimate the total number of comparisons performed by the pruned-tree

sorting algorithm as

$$S(N) = S(2^n) - \text{savings} = S(2^n) - \sum_{i=0}^{n-2} d_i[S(2^i) - 1 + (M(2^i) - 1)(n-i) + (i-1)v_i]$$

and thus

$$S(N) \leqq 2N\lceil \log N \rceil - 4N + \text{lower order terms.} \tag{13}$$

To obtain the last expression we have (i) neglected the (negative) contribution of the term $(i-1)v_i$; (ii) used expressions (11) and (12) for $M(2^i)$ and $S(2^i)$, respectively; and (iii) applied some simple algebraic manipulation. The sublinear terms turn out to be of $O(\log^3 N)$. Comparing (13) with (12), we see that the complexity of the sorting algorithm for arbitrary N differs from the complexity for N a power of two only in sublinear terms. Summarizing the above discussion, we have Theorem 6.

THEOREM 6. *The pruned-tree version of adaptive bitonic sorting executes a number of comparisons*:

$$S(N) = 2N\lceil \log N \rceil - 4N + O(\log^3 N). \tag{14}$$

6. Conclusions. We have presented a parallel sorting algorithm with optimal $TP = O(N \log N)$ complexity for $\Omega(\log^2 N) \leqq T \leqq O(N \log N)$. To explore the practical potential of our algorithm we must examine the constant factors. The small comparison count ($<2N \log N$) is encouraging, but we need to consider the total number of operations. To this end, we observe that, when running on P processors, adaptive bitonic sorting executes all the operations that it would execute on one processor plus a small number ($O(P \log N)$) of memory references due to interprocessor communication. Therefore, an accurate estimate of the operation count can be obtained by considering the performance of uniprocessor implementations. In general, the operation count is a lower bound of the PT measure on the PRAC, which is met only if processors actually perform useful operations at each timestep. This is nearly the case for adaptive bitonic sorting since, even for $T = O(\log^2 N)$, on average a processor is idle for less than one third of the total running time.

We have coded the sequential pruned-tree version of our sorting algorithm in C under Berkeley Unix 4.2 on a VAX 780, and a Gould 9080. The only optimization we have performed is the straightforward removal of recursion. As a term of comparison, we have chosen "quicker-sort," the Unix system sort, which is a carefully tuned version of quicksort. On sequences of length up to 2^{19}, the running time of our algorithm has consistently been below 2.5 times the running time of quicker-sort. This performance is remarkable for an algorithm that, with a small synchronization overhead and high processor utilization, can run in $O(\log^2 N)$ parallel time.

The combination of relative simplicity, optimal operation count, and small overhead makes adaptive bitonic sorting appealing for practical implementation.

Acknowledgment. We are grateful to the referees for their careful reading of the paper and for their constructive comments.

REFERENCES

[AKS83] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(\log N)$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, April 1983, pp. 1–9.

[AS85] S. G. AKL AND N. SANTORO, *Optimal parallel merging without memory conflicts*, in Proc. 1st International Conference on Supercomputing Systems, St. Petersburg, FL, December 1985, pp. 205–208.

[Ba68] K. E. BATCHER, *Sorting networks and their applications*, in Proc. AFIPS Springer Joint Computer Conference, Vol. 32, AFIPS Press, Arlington, VA, April 1968, pp. 307–314.

[BH85] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130–145.

[Bl85] G. BILARDI, *A family of merging networks derived from the bitonic merger*, in Proc. 23rd Annual Allerton Conference on Communication, Control and Computing, Monticello, IL, October 1985, pp. 261–267.

[Bl88] ———, *Merging and sorting networks with the topology of the omega network*, IEEE Trans. Comput., to appear.

[BP84] G. BILARDI AND F. P. PREPARATA, *An architecture for bitonic sorting with optimal* VLSI *performance*, IEEE Trans. Comput., 33 (1984), pp. 646–651.

[C86] R. COLE, *Parallel merge sort*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, October 1986, pp. 511–516.

[C87] ———, *Parallel merge sort*, Tech. Report 278 Computer Science Department, New York University, NY, March 1987.

[HH81] R. HAGGKVIST AND P. HELL, *Parallel sorting with constant time for comparisons*, SIAM J. Comput., 10 (1981), pp. 465–472.

[HS82] Z. HONG AND R. SEDGEWICK, *Notes on merging networks*, in Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, May 1982, pp. 296–302.

[Kn73] D. E. KNUTH, *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Kr83] C. P. KRUSKAL, *Searching, merging, and sorting in parallel computation*, IEEE Trans. Comput., 32 (1983), pp. 942–946.

[L85] T. F. LEIGHTON, *Theory of parallel computation and* VLSI, Class Notes, Massachusetts Institute of Technology, Cambridge, MA, 1985.

[LPV81] G. LEV, N. PIPPINGER, AND L. G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., 30 (1981), pp. 93–100.

[NS79] D. NASSIMI AND S. SAHNI, *Bitonic sort on a mesh-connected parallel computer*, IEEE Trans. Comput., 28 (1979), pp. 2–7.

[P78] F. P. PREPARATA, *New parallel-sorting schemes*, IEEE Trans. Comput., 27 (1978), pp. 669–673.

[Pe77] M. C. PEASE, *The indirect binary n-cube microprocessor array*, IEEE Trans. Comput., 26 (1977), pp. 458–473.

[Pr83] Y. PERL, *The bitonic and odd-even networks are more than merging*, Tech. Report DCS-TR-123, Department of Computer Science, Rutgers University, New Brunswick, NJ, February 1983.

[PV81] E. P. PREPARATA AND J. VUILLEMIN, *The cube-connected-cycles: a versatile network for parallel computation*, Comm. ACM, 24 (1981), pp. 300–309.

[Sn85] M. SNIR, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688–708.

[St71] H. S. STONE, *Parallel processing with the perfect shuffle*, IEEE Trans. Comput., 20 (1971), pp. 153–161.

[St78] ———, *Sorting on* STAR, IEEE Trans. Software Engrg., 4 (1978), pp. 138–146.

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

[TK77] C. D. THOMPSON AND H. T. KUNG, *Sorting on a mesh-connected computer*, Comm. ACM, 20 (1977), pp. 263–271.

[V75] L. G. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 229-243, April 1989

003

THE TOKEN DISTRIBUTION PROBLEM*

DAVID PELEG† AND ELI UPFAL†

Abstract. A solution to the following fundamental communication problem is presented. Suppose that n tokens are arbitrarily distributed among n processors with no processor having more than K tokens. The problem is to specify a bounded-degree network topology and an algorithm that can distribute the tokens uniformly among the processors.

The first result is a tight $\Theta(K+\log n)$ bound on the complexity of this problem. It is also shown that an approximate version of this problem can be solved deterministically in $O(K+\log n)$ on any expander graph with sufficiently large expansion factor.

In the second part of this work, it is shown how to extend the solution for the approximate distribution problem to an optimal probabilistic algorithm for the exact distribution problem on a similar class of expander graphs. Note that communication through an expander graph is a necessary condition for an $O(K+\log n)$ solution of the problem.

These results have direct applications to the efficient implementation of many-to-one and one-to-many communication requests, as well as to the solution of load-balancing problems in distributed systems.

Key words. parallel and distributed computation, communication, expander graphs

AMS(MOS) subject classifications. 68Q25, 68R10, 68M10

1. Introduction. Information exchange between processors is essential for any efficient parallel computation. In most applications, data transfer between individual processors has to be done through a relatively sparse communication network. Thus, the problem of routing simultaneously many packets on bounded-degree interconnection networks is fundamental to the theory of parallel and distributed computation.

The problem of packet routing is best formulated in terms of communication requests. An (m, K_1, K_2)-*communication request* is a set of up to m pairs. Each pair specifies a source and a destination for one packet in the network, and no processor appears as a source (respectively, destination) of more than K_1 (respectively, K_2) packets. The (m, K_1, K_2)-*routing problem* is the problem of routing simultaneously an (m, K_1, K_2)-communication request.

Most previous research has focused on the problem of routing a permutation or an $(n, 1, 1)$-communication request (one-to-one communication) on an n-vertex graph. An $O(\log n)$ deterministic solution for this problem is known for certain bounded-degree networks enabling sorting in $O(\log n)$ steps (henceforth, *S-networks*), based on the AKS sorting network [AKS], [L1]. $O(\log n)$ probabilistic algorithms have been analyzed for more practical networks like the butterfly [U] and the d-way shuffle [Ale]. These results match the $\Omega(\log n)$ lower bound for the implementation of an $(n, 1, 1)$-communication request on any n-vertex bounded-degree network.

However, permutation routing is a very restrictive form of communication, and the assumption that individual processors can coordinate their activity in a distributed computation so that only one message is addressed to each processor at any given time can rarely be justified. Furthermore, the algorithms developed for $(n, 1, 1)$-routing

* Received by the editors July 23, 1986; accepted for publication (in revised form) April 19, 1988. A preliminary version of this work appeared in the Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, October 1986, pp. 418-427.

† Computer Science Department K53, IBM Almaden Research Center, San Jose, California 95120. Present address, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel.

do not "scale up" to give an optimal solution for communication requests with higher parameters. (A naive iterative application of a permutation algorithm for an (n, K, K)-communication request takes $O(K \log n)$ steps.) Thus, a new approach is needed for the more general problem.

In the first part of this work we give an optimal solution for the general communication problem on bounded-degree networks.

THEOREM 1. *Any solution of the (n, K_1, K_2)-routing problem on an n-processor bounded-degree network requires $\Omega(K_1 + K_2 + \log n)$ parallel steps.*

THEOREM 2. *There exist an (explicitly constructible) n-processor bounded-degree network and a deterministic algorithm that solves any (n, K_1, K_2)-communication request on this network in $O(K_1 + K_2 + \log n)$ parallel steps.*

The core of our solution is an optimal algorithm for the (n, K)-*token distribution problem*, which is stated as follows: n tokens are initially distributed among n processors, with no more than K at each site. Redistribute the tokens through a bounded-degree network so that each processor will have exactly one token.

This problem is thus very similar to the $(n, K, 1)$-routing problem, except that the destinations of the tokens are immaterial and not fixed in advance. It is trivial to see that any algorithm for the (n, K)-distribution problem, combined with any permutation-routing algorithm, implies also an algorithm for the $(n, K, 1)$-routing problem, with the same time complexity (up to an additive factor of $O(\log n)$). It is slightly harder to show that the same applies to the $(n, 1, K)$-, and hence also to the (n, K, K)-, routing problems.

This problem was studied to some extent on networks of unbounded degree. Leighton [L2] has derived an $O(K + \log n)$ algorithm for (n, K)-token distribution on an n-processor hypercube. The best lower bound known for this problem on the hypercube is $\Omega(K/\log K)$[K]. In this paper we give an optimal solution for the problem on bounded-degree networks.

THEOREM 3. *The complexity of the (n, K)-token distribution problem on bounded-degree communication networks is $\Omega(K + \log n)$.*

THEOREM 4. *There exist an (explicitly constructible) n-processor bounded-degree network and a deterministic algorithm that solves the (n, K)-token distribution problem on this network in $O(K + \log n)$ steps.*

In addition to its importance in the context of communication, the token distribution problem has direct applications to the problem of load balancing in distributed systems. Consider each token as an independent process waiting for execution. Then a token distribution algorithm provides an efficient distributed method for balancing the queue lengths of the individual processors. This and other applications motivate the study of a more general distribution problem, in which the number of tokens is larger than the number of processors in the system and the goal is to uniformly distribute the tokens among the processors. Our method is extended to give an optimal solution for this problem as well [PU].

The token distribution algorithm works in two phases. First an *approximate* token distribution problem is solved deterministically in $O(K + \log n)$ steps, leaving no more than $O(1)$ tokens in each processor. Then, an *exact* distribution is achieved using an $O(\log n)$ routing or sorting algorithm.

It is easy to verify that both these problems can be solved in $O(K + \log n)$ time only on expander graphs. (An (α, β)-*expander* is a graph with the property that for every set U subject to $|U| \leqq \beta n$, $|\Gamma(U) - U| \geqq \alpha |U|$, where $\Gamma(U)$ denotes the set of neighbors of nodes in U.) The rapid progress in the explicit construction of strong, low-degree expander graphs [LPS] suggests that expanders are very likely to become

a feasible pattern for future communication networks. These new construction methods should motivate more research on algorithms that exploit the special properties of expander graphs.

Since these problems appear in a variety of applications, our goal is to provide an optimal solution for a large class of communication networks, preferably for all networks for which such a solution exists. Indeed, the solution to the approximate token distribution problem differs from other known algorithms for network routing in that it is not tailored for one particular network; rather, it solves the problem on any d-regular expander graph with a sufficient expansion factor (which is still small enough so that there exist such explicitly constructed networks).

The solution to the approximate problem involves solving a subproblem which is of general interest by itself. This is the problem of constructing a directed subgraph in a network (e.g., a spanning tree), based on some local property P, such that the subgraph is centered around the nodes enjoying P, and the other nodes appear only in the periphery (i.e., as leaves). This can be done distributedly, when initially every node knows only whether he/she itself has P.

In the second part of this work (§ 5) we attempt to achieve a similarly wide result for the *exact* K token distribution problem. We do not know of any deterministic algorithm that can sort or route n elements in $O(\log n)$ steps on every expander. However, we present a probabilistic algorithm that solves the $(n, 1, 1)$-routing problem, i.e., executes any $(n, 1, 1)$-communication request in $O(\log n)$ steps on any d-regular expander. The algorithm is a variant of Valiant's two-phase routing algorithm [V]. Note that the routing algorithms mentioned above were devised for special networks, and their analyses depended strongly on the network topology. Our analysis of the new algorithm uses a completely different approach. Since we analyze the algorithm for a whole class of graphs, characterized by some abstract properties, the analysis can make use only of these properties of the graphs rather than their particular design. In our case we use information about the eigenvalues of the adjacency matrix of an expander graph to analyze the convergence of a Markov chain, which is the core of the algorithm.

THEOREM 5. *Let G be an n-vertex d-regular graph, and let $\lambda(G)$ denote the second largest eigenvalue in absolute value of its adjacency matrix. If $d - \lambda(G)$ is bounded away from zero, then the probabilistic routing algorithm implements any $(n, 1, 1)$-communication request on G in $O(\log n)$ parallel steps.*

A recent result of Alon [Al1], [Al2] guarantees that the above condition on $\lambda(G)$ holds for any d regular expander graph. Thus we prove Theorem 6.

THEOREM 6. *Exact (n, K)-token distribution can be achieved by a probabilistic algorithm in $O(K + \log n)$ steps on any sufficiently strong regular expander graph.*

Theorem 4 has further implications, beyond the context of the token distribution problem. To mention a few, using Leighton's sorting algorithm [L1] we get Corollary 7.

COROLLARY 7. *There exists a probabilistic algorithm for sorting n elements on any n-processor regular expander graph in $O(\log^2 n)$ steps.*

And in general, using a recent result of Alt et al. [AHMP].

COROLLARY 8. *There exists a probabilistic algorithm that can simulate the execution of any n-processor T-step* PRAM (*parallel random access machine*) *program in $O(T \log^2 n)$ steps on any n-vertex regular expander.*

Finally, in § 6 we study the token distribution problem on general given networks. We first give a specific lower bound for any network and any initial distribution. This bound can be as bad as $\Omega(n)$ for some particular graphs and distributions. We then

give a general deterministic algorithm that works on any network and for every initial distribution in $O(n)$ steps.

2. Notation. We first define precisely the underlying model. Throughout, our communication network is modeled by an undirected graph $G=(V, E)$, where $|V|=n$ and the maximal degree is d. The system is assumed to be synchronous, and we use the number of rounds (clock ticks) as our time measure. A node can send at most one message along any adjacent edge in one timestep, where by *message* we mean either a token (i.e., a packet) or a string of $O(\log n)$ bits. Thus a node may send up to d messages in any time unit. (This does not affect the time complexity of our problem, since we consider only graphs of bounded degree.) We assume that the processors can perform an unlimited amount of computation in every timestep. However, in our upper bounds local computations are minimal (i.e., logarithmic in n).

For every node $v \in V$ we denote the number of tokens in v in any given moment by $t(v)$, where $\sum_v t(v)=n$. We assume that the tokens are numbered. However, this is not necessary, as the tokens can be numbered initially in $O(\log n)$ time.

3. Reducing routing to distribution. In this section we sketch how an $O(K+\log n)$ algorithm for token distribution (to be described in the rest of the paper) implies an $O(K_1+K_2+\log n)$ solution for the (n, K_1, K_2)-routing problem on an n-vertex network.

For every processor v_i, let x_i denote the number of tokens whose destination is $v_i (x_i \leqq K_2)$.

A central component of the simulation is a step in which we count the number of tokens to be sent to each processor. For this step we assume that we have preconstructed on the graph a tree of bounded degree and depth $O(\log n)$ whose leaves are precisely all the nodes of the graph (so each node appears once as a leaf and possibly several more times as an internal node). It is easy to prove that such a tree exists for any bounded-degree graph whose diameter is $O(\log n)$, and hence in particular for any expander. The tree has to be constructed once, and be known to all the processors. We also assume, for the purposes of the simulation, that the nodes of the graph are numbered in an order corresponding to some depth-first traversal of the tree.

The routing problem is solved in six steps:

(1) Run the token distribution algorithm.
/* At the end of this step each processor stores exactly one token. */

(2) Sort the tokens according to their destinations.
/* At this point, the x_1 tokens with destinations v_1 are located in processors $v_1, \cdots, v_{x_1}$, the x_2 tokens with destinations v_2 are located in processors $v_{x_1+1}, \cdots, v_{x_1+x_2}$, and so on. */

(3) Count for each processor v_i the number x_i of tokens to be sent to it. This can easily done by the following three substeps:

(3.1) Perform the actual counting, using the tree structure described earlier, by shipping upwards partial counting information. (See [Ul].)
/* At the end, the numbers x_i are held at different nodes in various levels of the tree, with at most d numbers in any "virtual" tree node, and hence at most $O(d \log n)$ numbers in each "real" graph node. */

(3.2) Distribute the numbers in the graph, using the token distribution algorithm.

(3.3) Permute these values, sending x_i to v_i.
/* At the end of this step, each processor v_i knows the number of tokens it should expect. */

(4) Each processor v_i produces x_i dummy tokens and supplies them with the labels "(i, j)," $1 \leqq j \leqq x_i$.
(5) Run the token distribution algorithm on the dummy tokens and then sort them lexicographically according to their labels (with i being more significant than j). During this step the paths used by the tokens are recorded by the intermediate processors. (That is, in every time unit, every processor records the label and entry/exit edge of each token going through it.)
/* After this step every node has two tokens: a real one, destined for some node v_i, and a corresponding dummy one that originated at v_i. */
(6) Send the real tokens to their destinations, each traversing the path "marked" by its corresponding dummy token, in reverse order.

Analysis. The sorting and permutation of steps (2), (3.3), and (5) require the network to be an S-network and take time $O(\log n)$. Step (3.1) can be done in time $O(\log n)$ also; step (4) takes time $O(1)$; and step (6) takes the same amount of time as step (5). Assuming that we have an $O(K+\log n)$ token distribution algorithm, step (1) takes time $O(K_1+\log n)$; steps (5), (6) take time $O(K_2+\log n)$; and step (3.2) takes time $O(\log n)$ (since it starts with $K = O(\log n)$), so overall, the above reduction implies Theorem 3.1.

THEOREM 3.1. *There exist an (explicitly constructible) n-processor bounded-degree network and a (deterministic) algorithm that solves any (n, K_1, K_2)-communication request in $O(K_1+K_2+\log n)$ parallel steps on that network.*

To achieve this run-time we need a network that satisfies the requirements of the token distribution algorithm and is also an S-network.

4. The token distribution problem. In this section we give the lower bound for token distribution, and then describe a deterministic algorithm for the approximate problem on a wide class of expander networks. We then extend this algorithm to an algorithm for the exact problem.

THEOREM 4.1. *The approximate and exact (n, K)-token distribution problems require $\Omega(K+\log n)$ steps on any bounded-degree network. Further, $O(K+\log n)$ run time is achievable only on expander graphs.*

Proof. Since the degree of the graph is bounded, $\Omega(K)$ steps are needed in order to take K tokens out of a processor. Also, to send rs tokens out of any set of size s in $O(r)$ steps the set needs to have $O(s)$ edges connecting them to the rest of the network. Thus, the graph must be an expander.

To prove the $\Omega(\log n)$ bound, observe that in any bounded-degree network there are three vertices that are at least $\delta \log n$ apart from each other for some $\delta = \delta(d) \geqq 0$. Let u_1, u_2, and v be three such vertices. Fix an initial distribution of tokens as follows. Give one token to each processor in the graph other than u_1, u_2, and v. Give two tokens to v and the remaining one to either u_1 or u_2. Let $N(u_1, \frac{1}{2}\delta \log n)$ be the set of vertices that are within distance $\frac{1}{2}\delta \log n$ or less from u_1 and let $F(u_1, \frac{1}{2}\delta \log n)$ be the set of vertices that are in distance $\frac{1}{2}\delta \log n$ or more from u_1. At the termination of the algorithm, the total flow from the set $F(u_1, \frac{1}{2}\delta \log n) - N(u_1, \frac{1}{2}\delta \log n)$ to the set $N(u_1, \frac{1}{2}\delta \log n) - F(u_1, \frac{1}{2}\delta \log n)$ is one if u_1 did not get a token initially and zero otherwise. The set $N(u_1, \frac{1}{2}\delta \log n) \cap F(u_1, \frac{1}{2}\delta \log n)$ cannot get information on the initial number of tokens in u_1 in less than $\frac{1}{2}\delta \log n$ steps; thus the algorithm execution must take $\Omega(\log n)$ steps. □

4.1. Approximate token distribution. Our algorithm employs parallelism on two levels. The algorithm consists of $O(\log K)$ phases, each of which reduces the maximum number of tokens at any site in the network by a constant factor. In the execution of

each phase, parallelism is employed in the n processors in the usual sense. However, executing the $O(\log K)$ iterations successively does not yield the desired run-time. To speed up the algorithm we introduce a second level of parallelism by executing all of the phases in parallel. We manage to do so despite the fact that the input of one phase is the output of the previous one.

Our solution works on graphs of the following type. A *sqrt expander* is an expander with the additional property that there exists $0 \leqq i \leqq d$ subject to $\alpha > (d+i^2)/(i+1)$. For such expanders we define $l = \min\{i \mid \alpha > (d+i^2)/(i+1)\}$ as the *spanning factor* of the *sqrt* expander. Note that this requirement holds approximately when $\alpha > 2\sqrt{d}$, and it is possible to explicitly construct *sqrt* expanders with the above property and with relatively low degree. For instance, the construction in [LPS] yields expanders with second eigenvalue $\lambda_2 \leqq 2\sqrt{d-1}$, and using Tanner's equation [T, Thm. 2.1] we get $\alpha > d/4$ for sufficiently small β.

We first describe the algorithm assuming the phases are carried out sequentially. Our goal is to reach a situation in which $t(v) \leqq c$ for every v, for some fixed constant c (independent of n and K). In particular, for a *sqrt* (α, β)-expander we choose $c = \lceil 2/\beta \rceil$. In the beginning of phase j ($j = 0, 1, \cdots$) the current maximum is no larger than K_j. The *convergence property* relating the phases is $K_j = \gamma^j K$, or alternatively $K_{j+1} = \gamma K_j$, where $\gamma = (2l+1)/(2l+2)(0 < \gamma < 1)$ and l is the spanning factor of the underlying expander.

As the phases are identical, we shall describe one phase j in general. Consider a set U_j of "heavily loaded" nodes (i.e., all nodes with more tokens than a certain threshold). During the phase the nodes first distributedly construct a directed acyclic graph in which *all* of the heavy nodes participate as internal nodes, and the leaves are taken from among the lightly loaded nodes. Then, the internal nodes ship tokens to their children in the dag for a prescribed number of steps. The construction of the dag guarantees that the heavy nodes keep losing tokens in every shipping step (it is most convenient to visualize the particular case of a binary forest). More precisely, let $U_j = \{v \mid t(v) \geqq K_j/2\}$. The phase j consists of two parts.

Part (1) *Dag construction.* Distributedly construct a *dag* $G' = (V', E')$ with the following properties:

(1) $U_j \subseteq V' \subseteq V$.

(2) $E' \subseteq E$ (looking at the underlying undirected edges).

(3) For every $v \in V'$, indegree $(v) \leqq l$.

(4) For every $v \in U_j$, outdegree $(v) = l+1$.

(Again l is the spanning factor of the expander. Note that when $l = 1$ indeed we get a binary forest.)

Part (2) *Token flow.* Perform $K_j/2(l+1)$ "flow" steps, in which on every edge $(u, v) \in E'$, u sends a token to v.

Before going into the details of the construction, let us note that the above properties of the dag guarantee the promised convergence property.

LEMMA 4.2. *Upon completion of phase j, for every v,* $t(v) \leqq K_{j+1} (= \gamma K_j)$.

Proof. Clearly we only need to consider nodes in V'. If $v \in U_j$ then in every flow step it receives at most l tokens and sends $l+1$, so it loses a token per step, and hence the new bound. If $v \in V' - U_j$ then it begins with less than $K_j/2$ tokens, and gets at most l in every flow step, so it will also end with no more than γK_j tokens. □

Construction of the dag. The dag is constructed "bottom up" while keeping the nodes of U_j that did not choose children yet in a set W. Throughout, each node knows whether or not it is in W. Initially $W = U_j$ and in every following step some of the

nodes in W choose children and leave W, and are ready to be chosen as children of other members of W. Each step of the construction of the dag is of the following form:

(1) Every node $v \in W$ informs all its neighbors that it belongs to W.
(2) A node $v \notin W$ that received such messages from at most l neighbors returns to these neighbors the message "ready" (meaning intuitively: "I'm ready to be chosen as a child and consequently to receive tokens").
(3) A node $v \in W$ that has received "ready" messages from at least $l+1$ neighbors, chooses exactly $l+1$ of them as children and so informs them. (The edge set E' will contain directed edges from v to these nodes.) For the rest of the construction, v removes itself from W.
These steps are repeated as long as $W \neq \emptyset$.

It is easy to see that nodes participating in the dag obey the degree requirements in the definition. Specifically, we have the following lemma.

LEMMA 4.3. *For every* $v \in V'$, indegree $(v) \leqq l$, *and if* $v \in U_j$ *then also* outdegree $(v) = l+1$.

Proof. A node v receives parents in the dag only if and when $v \notin W$. Consider the first iteration i during which a node $v \notin W_i$ sends "ready" messages to (some of) its neighbors. This happens if there are at most l neighbors of v in W_i. The set W shrinks with every iteration, so clearly all possible parents of v come from among its (at most l) possible parents in W_i.

The second condition holds trivially. □

For the analysis, denote the sequence of sets W in the successive iterations by $U_j = W_0 \supseteq W_1 \supseteq \cdots \supseteq W_m$. We have to prove that the construction process stops with $W_m = \emptyset$ and further, that $m \in O(\log |U_j|)$. The proof can be sketched along the following lines.

For a set $U \subseteq V$ we denote by $\Gamma(U)$ the set of neighbors of nodes in U. Let $\Gamma_i = \Gamma(W_i) - W_i$, $X_i = \{v \mid v \in \Gamma_i, v \text{ has at most } l \text{ neighbors in } W_i\}$, $Y_i = \{v \mid v \in W_i, v \text{ has at least } l+1 \text{ neighbors in } X_i\}$, and $\delta = 1 - ((l+1)\alpha - d - l^2)/l(d-l)$.

LEMMA 4.4. $0 < \delta < 1$.

Proof. It suffices to show that $0 < ((l+1)\alpha - d - l^2)/l(d-l) < 1$, and this follows from the choice of the expander to satisfy $(d+l^2)/(l+1) < \alpha < d$. □

LEMMA 4.5. $|\Gamma_i| \geqq \alpha |W_i|$, *where* α *is the expansion constant of the graph.*

Proof. The definition of U_j implies $|U_j| \leqq 2n/K$. $W_i \subseteq U_j$, and $K > c = \lceil 2/\beta \rceil$. Put together, $|W_i| \leqq |U_j| \leqq 2n/\lceil 2/\beta \rceil \leqq \beta n$, so the expansion requirement holds for W_i. □

LEMMA 4.6. $|X_i| \geqq (((l+1)\alpha - d)/l)|W_i|$.

Proof. Denote by e the number of edges between W_i and Γ_i. Clearly $e \leqq d|W_i|$. On the other hand, from every node in X_i there is at least one edge to W_i, and from every node in $\Gamma_i - X_i$ there are at least $l+1$ such edges. Hence,

$$1 \cdot |X_i| + (l+1)|\Gamma_i - X_i| \leqq e.$$

Combining the two inequalities we get $(l+1)|\Gamma_i| - l|X_i| \leqq d|W_i|$. Together with the previous lemma we have $l|X_i| \geqq (l+1)\alpha|W_i| - d|W_i|$. □

LEMMA 4.7. $|Y_i| \geqq (1-\delta)|W_i|$.

Proof. Every node in Y_i has at most d neighbors in X_i, and every node in $W_i - Y_i$ has at most l neighbors in X_i. Therefore

$$l|W_i - Y_i| + d|Y_i| \geqq |X_i|,$$

and the result follows from the previous lemma. □

COROLLARY 4.8. $|W_{i+1}| \leqq \delta |W_i|$.

Proof. The proof follows from the fact that $W_{i+1} = W_i - Y_i$, together with the previous lemma. □

COROLLARY 4.9. *The dag can be constructed correctly in* $O(\log n)$ *steps.* □

Summarizing, the algorithm requires $O(\log K)$ phases, where phase j takes time $O(K_j + \log n)$ and $K_j = \gamma^j K$. Therefore, the K_j components sum up to $O(K)$, and the entire time is in $O(K + \log n \log K)$, which, for $K \leqq n$, is contained in $O(K + \log n \log\log n)$. To reach optimality it remains to eliminate the consecutive execution of the phases.

Parallelization of the phases. It is clear that as long as $K_j \geqq O(\log n)$, the cost of a single phase is dominated by the "token flow" steps. The cost of the dag construction becomes significant only from that point on. Therefore we may execute the first few phases sequentially, and parallelize only when $K_j < 2(l+1)\log n$.

The idea for parallelizing the phases is based on the observation that we can "separate" the phases, i.e., construct all the dags first and perform all the token transitions afterwards. Furthermore, the communication required for the construction of the dags is only $O(1)$ bits per step, while the model allows us to use $O(\log n)$ bit messages. We also observe that to begin building the dag for the $j+1$st phase it is not necessary to have completed the jth dag, but merely to know U_{j+1}. This leads to the idea of trying to find a fast way of identifying the sets U_j in advance, and then constructing the dags in parallel. An obvious obstacle to this idea is the fact that the exact composition of U_{j+1} is determined only upon completion of the construction of the jth dag. However, it is possible to maintain approximations $\hat{t}(v)$ for the number of tokens in every node, upper bounding the real number $t(v)$, and to identify a larger set $\hat{U}_{j+1}$ that is guaranteed to contain U_{j+1} and that can be constructed in $O(\log\log n)$ steps (from knowing $\hat{U}_j$).

It should be clear that when using $\hat{U}_j$ instead of U_j we will still have $K_j = \gamma^j K$ as the bounds on the number of tokens at the beginning of phase j. The worst that might happen is that nodes in $\hat{U}_j - U_j$ will discover this fact during the actual flow steps, and may cease sending tokens if they do not have any, which can only result in some nodes having fewer tokens than expected.

The sets $\hat{U}_j$ are constructed *serially* as follows. For $j = 0$, $\hat{U}_0 = U_0 = \{v \mid t(v) \geqq K/2\}$. For $j \geqq 0$, we start phase j from $\hat{U}_j$ and apply the dag construction for $m = 2\log_{1/\delta}\log n$ iterations. The remaining set of candidates (nodes from $\hat{U}_j$ who still need to choose children) was denoted earlier by W_m. Let $\hat{W}_j = W_m$ and $Z_j = \hat{W}_j \cup \Gamma(\hat{W}_j)$. Denote by T_j the set of all nodes $v \notin Z_j$ that entered the dag during the jth iteration (as parents, children, or both). At this point each node v in the graph is required to determine its value of $\hat{t}(v)$ at the end of the jth phase. For nodes $v \notin (Z_j \cup T_j)$ this value cannot possibly change during this phase. Nodes in T_j ask their parents in the dag for the number of tokens they should expect to receive during the flow steps of phase j (note that, unlike the situation in the serial execution, a parent may send its sons fewer than $K_j/2(l+1)$ tokens during phase j). Since the dag is of depth $m = -2\log_\delta \log n$, this information can be collected in $O(\log\log n)$ steps using $O(\log\log n)$-bit messages, and then the new value of $\hat{t}(v)$ can be computed directly, for every $v \in T_j$. Finally, nodes in Z_j *cannot* yet estimate the number of tokens they will have at the end of the phase. Therefore they exaggerate and take this number to be the maximum possible, K_{j+1}. This can be viewed as a commitment on behalf of these nodes to produce "dummy" tokens and distribute them if they do not have enough. (This assumption serves to simplify the analysis, but the commitments clearly do not have to be honored

during the actual flow steps, which accounts for the difference between $t(v)$ and $\hat{t}(v)$.) Now we take $\hat{U}_{j+1} = \{v \mid \hat{t}(v) \geqq K_{j+1}/2\}$.

This process can be repeated as long as $\hat{U}_j \leqq \beta n$. Clearly $U_j \subseteq \hat{U}_j$. The crux of the analysis is in showing that the sets $\hat{U}_j$ do not grow too fast. This requires us to bound the number of dummy tokens added along the process.

LEMMA 4.10. *The number of phases is* $O(\log \log n)$.

Proof. Since $\hat{U}_j \geqq U_j$, the number of phases is bounded above by the number of phases in the serial algorithm. □

LEMMA 4.11. *In every phase* j, $|Z_j| \leqq (d+1)n/\log^2 n$.

Proof. Recall that $\hat{W}_j$ is the set W_m obtained after m iterations of the dag construction in phase j. Hence $|\hat{W}_j| \leqq \delta^m |W_0| \leqq \delta^{-2 \log_\delta \log n} n = n/\log^2 n$, and $|Z_j| \leqq |\hat{W}_j| + |\Gamma(\hat{W}_j)| \leqq (d+1)n/\log^2 n$. □

LEMMA 4.12. *In every phase* j, $\sum_v \hat{t}(v) \leqq 2n$.

Proof. The number of dummy tokens added by nodes in Z_j in any single phase is bounded by $|Z_j| K_j \leqq (d+1)n/\log n$. Hence by Lemma 4.10, the total number of tokens added to the system is in $o(n)$. □

LEMMA 4.13. $|\hat{U}_j| \leqq 4n/K\gamma^j$.

Proof. By the definition of $\hat{U}_j$ and by Lemma 4.12,

$$2n \geqq \sum_{v \in \hat{U}_j} \hat{t}(v) \geqq |\hat{U}_j| \frac{K\gamma^j}{2},$$

and the result follows immediately. □

As noted before, the sets $\hat{U}_j$ are usable as long as they are no larger than βn. By the last lemma this holds while $4n/K\gamma^j \leqq \beta n$, or in other words while

$$j \leqq J = \left\lfloor \log\left(\frac{4}{\beta K}\right) \Big/ \log \gamma \right\rfloor.$$

This number of phases lowers K down to $K' \leqq \lceil 4/\beta \rceil$, which is still constant, although twice as large as that achieved in the serial construction. Therefore the general process proceeds as follows.

(1) For each of the first $J+1$ phases ($0 \leqq j \leqq J$), start from the set $\hat{U}_j$, perform the first m steps of the dag construction, identify the sets $\hat{W}_j$ and Z_j, and determine the new value of $\hat{t}(v)$ for every v and $\hat{U}_{j+1}$. These phases are executed serially, in time $O((\log \log n)^2)$.

(2) Complete the construction of all the dags of the first $J+1$ phases *in parallel*, based on the sets $\hat{W}_j$, in time $O(\log n)$.

(3) Perform the actual flow of tokens on the constructed dags, phase by phase (for J phases).
The total cost is therefore $O(K + \log n)$.

THEOREM 4.14. *The approximate* $V(n, K)$*-token distribution problem can be solved deterministically in time* $O(K + \log n)$ *on sqrt expanders.* □

4.2. Exact token distribution. Our algorithm for the exact (n, K)-token distribution problem starts with solving the approximate problem. Assuming that the tokens are numbered, the remaining problem can be viewed as an $(n, c, 1)$-routing problem, where c is a constant independent of n and K. In this section we will show that the $(n, c, 1)$-routing problem can be solved deterministically, in $O(\log n)$ steps on any n-vertex S-network. More significantly, the result of the next section shows that the same run-time can be achieved probabilistically on any regular $(\alpha, \frac{1}{2})$-expander graph ($\alpha > 0$). Thus, we have the following theorem.

THEOREM 4.15. *Exact (n, K)-token distribution can be achieved in time $O(K + \log n)$:*

(1) *Deterministically on any network combined of a sqrt-expander + an S-network.*
(2) *Probabilistically on any network combined of a sqrt-expander + a regular $(\alpha, \frac{1}{2})$-expander $(\alpha > 0)$.* □

THE $O(\log n)$ DETERMINISTIC $(n, c, 1)$-ROUTING ALGORITHM. Initially all the tokens are labeled *I*. When a token reaches its final location its label is changed to *F*.

Repeat the following six steps for c times:

(1) A processor that stores some tokens with label I changes the label of one of these tokens to R. A processor that does not store such tokens generates a dummy token with label R and value $-\infty$.
(2) Sort the n tokens with label R according to their values.
(3) Discard the dummy tokens.
(4) A processor that stores a token with label F generates a dummy token with label R and value $-\infty$. A processor that does not store a token with label F or R generates a dummy token with label R and value $+\infty$.
(5) Sort the n tokens with label R.
(6) Discard the dummy tokens and change all the labels R to F.

Analysis. Since initially there are no more than c tokens in each processor, after c iterations of the algorithm all the tokens are *processed*, i.e., given label R, moved to a new location and given label F. Note that each token moves during exactly one iteration, and does not move afterward.

Assume that x tokens have been processed during the first $i-1$ iterations, and assume also that at the beginning of the ith step these tokens are stored in processors $1, \cdots, x$, one in each processor. (These inductive assumptions certainly hold for $i = 1$ and $x = 0$.) Let y be the number of tokens processed during the ith iteration. After the first sorting phase (step (2)), these y tokens are stored in processors $n-y+1, \cdots, n$, one in each processor. Clearly, $n - y \geqq x$ and the second sorting phase (step (5)) involves x dummy tokens with value $-\infty$ and $n-y-x$ dummy tokens with value $+\infty$. This implies that the y "real" tokens processed in the ith iteration end up in places $x+1, \cdots, x+y$. Thus at the end of the last iteration all the tokens are evenly distributed among the processors.

Each iteration involves two sorting phases; thus the total run-time is $O(c \log n)$, which is $O(\log n)$ when c is a constant.

5. Routing on regular expanders. In this section we give a probabilistic algorithm that solves the (qn, q, q)-routing problem on any d-regular n-vertex $(\alpha, \frac{1}{2})$-expander graph in $O(q \log n)$ parallel steps. As mentioned in § 4, augmenting this result with the approximate distribution algorithm we get a probabilistic algorithm that solves the (n, K)-distribution problem in optimal time on any graph that is both a *sqrt*-expander and an $(\alpha, \frac{1}{2})$-expander for some $\alpha > 0$.

The communication protocol we use is a variant of the protocol used in [U] for the butterfly network, based on Valiant's two-phase randomized routing scheme [V]. The process of routing a packet to its destination has two phases. Each phase consists of L ($L = O(q \log n)$) transitions. In the first phase every packet X performs L random transitions. If X is in processor u and e is an edge going out of u, then X chooses e for its next transition with probability $1/d$.

The first phase defines the following sets of probabilities for every pair of vertices $\{u, v\}$ and for every $1 \leqq j \leqq L$. $P^j_{u,v}$ is the probability that a packet initially located at

u enters processor v in its jth random transition, and $P^j_{u,v}(e)$ is the probability that a packet initially located at u enters v in its jth random transition through edge e. Note that these probabilities do not depend on the destination of the packet.

In the second phase every packet is sent from its current location to its final destination in L transitions. Its route is defined as follows: If a packet is on its way to processor u, and is located in processor v after executing $L-j$ transitions of this phase, then it chooses an edge e for its next transition with probability $P^j_{u,v}(e)/P^j_{u,v}$. We show in the analysis that for every pair (u, v), $P^L_{u,v}>0$. Thus, the routes are properly defined and on the Lth transition a packet always reaches its destination. Note that these probabilities must be computed only once, when the network is constructed, and this computation can be done very efficiently.

We still have to specify the queue policy of processors. In each step every packet has a *priority* associated with it, which is the number of transitions the packet has already performed. In the first phase the priority numbers are $1, \cdots, L$ and in the second phase $L+1, \cdots, 2L$. The packet with the smallest priority number is the first to be transmitted. Thus, priorities are used to speed up slower packets at the expense of faster ones.

THEOREM 5.1. *There are constants $S=S(d)$ and $R=R(d)$ such that the routing algorithm executes any (qn, q, q)-communication request on any d-regular $(\alpha, \frac{1}{2})$-expander in $Sq \log n$ steps with probability $1-e^{-R \log n}$.*

Proof. Let G be a d-regular $(\alpha, \frac{1}{2})$-expander and let $A=(a_{i,j})$ be its adjacency matrix. A random walk of a packet on the graph defines a Markov process in which a state of a packet at a given time is its location in the graph. Let $B=(b_{i,j})$ be its stochastic transition matrix. An estimate of the convergence rate of this process to its stationary state is the crux of the analysis. Since G is d-regular, $B=A/d$. To analyze the Markov process we need bounds on the eigenvalues of B, which we obtain from the following algebraic characterization of expanders.

For any matrix M we denote its eigenvalues by $\lambda_1(M) \geqq \lambda_2(M) \geqq \cdots$.

THEOREM 5.2 (Alon [Al1]). *Let G be a d-regular $(\alpha, \frac{1}{2})$-expander with an adjacency matrix A. Then $d-\lambda_2(A)$ is bounded away from zero.* □

By observing that the proof of Theorem 5.2 holds even if G is a multigraph we can slightly extend the result of the theorem. Let G^2 be the graph defined by the matrix A^2. Since G is an expander, so is G^2. Thus, $\lambda_2(A^2)$ is bounded away from d^2. However, $\lambda_2(A^2)=\max_{i \neq 1} \lambda_i(A)$; thus $\max_{i \neq 1} |\lambda(A)|$ is bounded away from d.

COROLLARY 5.3. *Let G be a d-regular $(\alpha, \frac{1}{2})$-expander with an adjacency matrix A. Then $d-|\lambda_i(A)|$ is bounded away from zero for any $i \neq 1$.*

Let $Q^j_{u,v}$ be the probability that a packet located at u will reach v in exactly j random transitions.

LEMMA 5.4. *There is a constant $\gamma<1$ such that for every pair (u, v), $Q^j_{u,v} \leqq 1/n+\gamma^j$.*

Proof. The matrix B is real and symmetric and thus has real nonnegative eigenvalues $\lambda_1(B), \cdots, \lambda_n(B)$ with corresponding orthogonal eigenvectors $\bar{a}_1, \cdots, \bar{a}_n$. Since $B=A/d$, $\lambda_i(B)=\lambda_i(A)/d$. It is easy to check that $\lambda_1(A)=d$, and from Corollary 5.3 we get that $\lambda_2(A)$ is bounded away from d; thus $\lambda_1(B)=1$ and $\lambda_2(B)$ is bounded away from 1. It is also easy to check that a possible choice for $\bar{a}_1$ is the vector I/n, where I is the unit vector.

To compute $Q^j_{u,v}$ we start with a vector $\bar{b}$ having 1 in the entry corresponding to the processor u and zero elsewhere and look at the entry corresponding to v in the vector $\bar{b}B^j$. Let $\sum_{i=1}^n \eta_i \bar{a}_i$ be the orthogonal representation of $\bar{b}$. Then $\bar{b}B^j=\sum_{i=1}^n (\lambda_i(B))^j \eta_i \bar{a}_i$. It is easy to verify that η_1 must be 1 since $\sum_{i=2}^n (\lambda_i(B))^j \eta_i \bar{a}_i$ converges to zero. Therefore, we get $Q^j_{u,v} \leqq 1/n+(\lambda_2(B))^j$. □

LEMMA 5.5. *Let* $L = 3(\log 1/\gamma)^{-1} \log n$. *Then for every pair* u, v, $1/n - 1/n^2 \leqq Q^L_{u,v} \leqq 1/n + 1/n^3$.

Proof. From Lemma 5.4 we immediately get the second inequality, since $Q^L_{u,v} \leqq 1/n + \gamma^L = 1/n + 1/n^3$. Now fix a pair of nodes u and v. Clearly $\sum_{w \in V} Q^L_{u,w} = 1$, and by the second inequality, $\sum_{w \neq v} Q^L_{u,w} \leqq (n-1)(1/n + 1/n^3)$. Put together, we get $Q^L_{u,v} \geqq 1/n - 1/n^2$. □

LEMMA 5.6 *For any* d-*regular* $(\alpha, \frac{1}{2})$-*expander* G *there exists a bounded-degree* d'-*regular* $(\alpha', \frac{1}{2})$-*expander* G' *for some* $\alpha' > 0$, (*obtained by a suitable iterative construction*) *with an adjacency matrix* A' *such that* $\lambda_2(A'/d') < \frac{1}{2}$ *and a communication step on* G' *can be simulated by* $O(1)$ *communication steps on* G.

Proof. Define A and B as above with respect to G and let k be the minimum integer such that $\lambda(B)^k < \frac{1}{2}$. Define G' as follows: G' has the same vertex set as G, u and v are connected by an edge in G' if and only if their distance in G is k. Clearly G' is a d^k-regular (multi-)graph, and if A' is the adjacency matrix of G' then $\lambda_2(A') = (\lambda_2(A))^k$. Thus, $\lambda_2(A'/d^k) = (\lambda_2(A)/d)^k = (\lambda(B))^k < \frac{1}{2}$, and clearly a step on G' can be simulated by kd^k steps of G. □

Due to Lemma 5.6 we can assume without loss of generality that $\gamma = \lambda_2(B) < \frac{1}{2}$.

LEMMA 5.7. *There are positive constants* S' *and* R' *such that the execution of the first phase terminates in* $S'q \log n$ *steps with probability* $1 - e^{-R'q \log n}$.

Proof. For the purposes of the proof we consider the following modified algorithm. No processor ever transmits a message of priority i before it has transmitted all messages of priority less than i that it will ever transmit. Such an execution clearly takes at least as long as an execution of the original algorithm.

The analysis is based on the familiar technique of *critical delay sequences* [U]. This technique has so far been applied only to highly structured networks such as the butterfly [P], [RV]. In contrast, expanders are of much more general, unspecified and seemingly irregular structure, which makes the task harder. A *delay sequence* is a sequence of processors $v_0, \cdots, v_L$, such that for any $i < L$ either $v_i = v_{i+1}$ or v_i is one of the d processors connected to v_{i+1}. A delay sequence is *critical* if v_L was one of the last processors to transmit a message in the execution of the algorithm, and for each $0 \leqq i < L$, v_i was one of the last processors to transmit messages with priority i from among the d processors connected to v_{i+1} and v_{i+1} itself.

For a given critical delay sequence D, define two random variables:

f^D_i is the number of messages with priority i leaving v_i, and

t^D_i is the time when all messages of priority j leaving v_i, for $j \leqq i$, have been transmitted.

It follows from the construction of the critical delay sequence that if the algorithm terminates after t steps, then $t \leqq t^D_L$. Moreover, at time t^D_i, v_i has finished transmitting all messages with priority less than or equal to i that pass through it and v_{i+1} has received all messages that it should transmit with priority $i+1$. Thus, at time t^D_i each of the f^D_{i+1} messages that v_{i+1} should transmit is either in its queue or has already been transmitted. Therefore, the time required to transmit the f^D_{i+1} messages from the head of v_{i+1}'s queue is an upper bound for $t^D_{i+1} - t^D_i$. Hence $t^D_{i+1} - t^D_i \leqq f^D_{i+1}$.

Denoting $F^D = \sum_{0 \leqq i \leqq L} f^D_i$ and fixing $t^D_0 = 0$, we get $t \leqq \sum_{1 \leqq i \leqq L} (t^D_i - t^D_{i-1}) \leqq \sum_{0 \leqq i \leqq L} f^D_i = F^D$.

Hence we are interested in bounding the probability of an execution that has a critical delay sequence D of length $F^D \geqq cq \log n$ for some $c > 0$.

In order to perform this analysis, we present each f^D_i as the sum of two random variables g^D_i and h^D_i.

g_i^D is the number of messages in f_i^D not appearing in $\cup_{0\leq j\leq i-1} f_j^D$, and h_i^D is $f_i^D - g_i^D$.

Let $F_1^D = \sum_i g_i^D$ and $F_2^D = \sum_i h_i^D$. Then $F^D = F_1^D + F_2^D$.

It is easy to estimate F_1^D. $E(g_i^D) \leq q$ and F_1^D is bounded by the number of successes in qnL independent Poisson trials. Thus, by Hoeffding theorem [H], $Pr\{F_1^D \geq \eta\} \leq B(\eta, qnL, 1/n)$, and using Chernoff [C] bound, we prove that $Pr\{F_1^D \geq s_1' q \log n\} \leq e^{-r_1' q \log n}$ [U].

The tricky part is to bound F_2^D, the estimate for the number of transitions the $F_1^D (= O(q \log n))$ packets performed inside the delay sequence, since a packet might return for an *unpredictable* number of times to the sequence after leaving it (as opposed to the situation on the butterfly). Here we rely again on the expansion properties of the graph. Let Q be the probability that a packet, currently in processor v_i in the delay sequence, will ever return to the sequence. Then, by Lemma 5.4,

$$Q = \sum_{j=1}^{L} Q^j_{v_i, v_{i+j}} \leq \sum_{j=1}^{L} \left(\frac{1}{n} + \gamma^j\right) < 1 \qquad \left(\text{recall } \gamma < \frac{1}{2}\right).$$

Thus, the number of transitions a packet performs on the delay sequence is bounded by a geometrically distributed random variable. The routes of distinct packets are probabilistically independent. Thus, using Chernoff bound again we prove that

$$Pr\{F_2^D \geq s_2' q \log n\} \leq e^{-r_2' q \log n}.$$

Letting $S' = s_1' + s_2'$ and $R' = \min\{r_1', r_2'\}$, we get the required result. □

It remains to analyze the run-time of the second phase.

LEMMA 5.8. *There are constants S'' and R'' such that the second phase is executed in $2S''q \log n$ steps with probability $1 - e^{-R''q \log n}$.*

Proof. The proof is based on the near symmetry between the first and the second phases. The two phases are not entirely symmetric since the endpoint of a packet in the first phase is not uniformly distributed. The bias, however, is bounded by $1/n^2$ (Lemma 5.5), which enables us to prove that the number of packets traversing an edge e with priority i in the execution of an $(2qn, 2q, 2q)$-communication request stochastically bounds the number of packets traversing e with priority $2L - i$ in the execution of a (qn, q, q)-communication request.

Let τ be a path of length L from processor v to processor w, and let $\bar{\tau}$ be the path τ in a reverse order. Let P_τ be the probability that a packet going from v to w in the first phase of the algorithm will use the path τ. Let $A_1(2q)$ be the execution of the first phase of the algorithm on a $(2qn, 2q, 2q)$-communication request and let $A_2(q)$ be the execution of the second phase of the algorithm on a (qn, q, q)-communication request.

From Lemma 5.5 we know that the number of packets going from v to w in $A_1(2q)$ is stochastically lower bounded by the binomial distribution with parameters $B(2qn, 1/n - 1/n^2)$. Thus, the number of packets taking the route τ in $A_1(2q)$ is lower bounded by $B(2qn, (1/n - 1/n^2)P_\tau)$. On the other hand, the number of packets going from w to v in $A_2(q)$ is stochastically upper-bounded by $B(qn, 1/n + 1/n^3)$ and by the definition of the second phase of the algorithm the number of packets taking the route $\bar{\tau}$ is upper-bounded by $B(qn, (1/n + 1/n^3)P_\tau)$. Thus, the number of packets taking the route τ in $A_1(2q)$ is stochastically larger than the number of packets taking the route $\bar{\tau}$ in $A_2(q)$. Fixing an edge e and summing over all routes of length L that contain the edge e as the ith edge of the route, we get that the number of packets traversing e with priority $L - i$ in $A_2(q)$ is stochastically bounded by the number of packets traversing

e with priority i in $A_1(2q)$. Let $T[A_1(2q)]$ (respectively, $T[A_2(q)]$) be the run-time of $A_1(2q)$ (respectively, $A_2(q)$). Then by Lemma 5.6

$$Pr\{T[A_2(q)] \geqq S'2q \log n\} \leqq Pr\{T[A_1(2q)] \geqq S'2q \log n\} \leqq e^{-R'2q \log n}. \quad \square$$

6. Token distribution on general networks. In this section we return to token distribution and look at a generalized version of the problem. The application of token distribution as a tool for load balancing in distributed systems may require solving the problem on an arbitrary given network which might not have good expansion properties. This general problem is much harder. We first give a combinatorial characterization of a lower bound for any particular graph, depending on its specific properties. For each set of vertices $U \subseteq V$, denote by $\gamma(U)$ the number of edges leaving U, and let $t(U)$ denote the total number of tokens initially in U. Let

$$\mathcal{K} = \max\left\{\frac{t(U)-|U|}{\gamma(U)} \mid U \subseteq V, |U| \leqq \frac{n}{2}\right\}.$$

THEOREM 6.1. *The general (n, n)-token distribution problem requires $\Omega(\mathcal{K} + \text{diameter}(G))$ parallel steps on any network G.* $\square$

COROLLARY 6.2. *There exist bounded-degree networks for which the (n, K)-token distribution problem requires $\Omega(n)$ steps, even for $K = 2$.*

Proof. Consider a straight line graph of n nodes (i.e., in which node i is connected to $i+1$ and $i-1$, except the endpoints). Assume the initial distribution is such that

$$t(i) = \begin{cases} 2 & \text{if } i \leqq n/2, \\ 0 & \text{otherwise.} \end{cases}$$

$\square$

For the upper bound we make the assumption that a directed spanning tree was preconstructed on the graph, and is known to every node in the graph. The algorithm we give consists of two steps. In the first, the tokens are sent upwards to the root. In the second, the root sends the tokens downwards according to their destinations—the token labeled i is sent to node i.

It is clear that the algorithm works correctly, and it remains to analyze its running time. Clearly the second step runs in $O(n)$ time. In order to analyze the first phase, let us add the requirement that whenever a node has more than one token, it first sends the token numbered least.

Denote by $h(i)$ the initial distance of token i from the root, and let $\tau(i)$ be the time it took token i to reach the root.

LEMMA 6.3. $\tau(i) \leqq \max\{h(i), \tau(1)+1, \cdots, \tau(i-1)+1\}$.

Proof. If token i was not delayed on its way to the root by other tokens, then $\tau(i) = h(i)$ and we are done. Now assume i was delayed, and let j be the last token delaying i. It is clear that $j < i$, and also that $\tau(i) = \tau(j)+1$, since i was not delayed afterwards, and it followed the same path as j to the root. $\square$

LEMMA 6.4. $\tau(i) \leqq n + i - 1$.

Proof. The proof is by induction on i. For $i = 1$, $\tau(1) = h(1) \leqq n$. Now assume the claim for $1, \cdots, i-1$. By the previous lemma $\tau(i) \leqq \max\{h(i), \tau(1)+1, \cdots, \tau(i-1)+1\}$. By the inductive hypothesis $\tau(i) \leqq \max\{n, n+1, \cdots, (n+(i-1)-1)+1\} = n+i-1$. $\square$

COROLLARY 6.5. *For every token i, $\tau(i) \leqq 2n$.* $\square$

THEOREM 6.6. *The exact (n, K)-token distribution problem can be solved deterministically on any graph in $O(n)$ parallel steps.* $\square$

7. Some open problems. (1) Our algorithm works only for relatively strong expanders. Can it be extended to every expander?

(2) Can the exact token distribution algorithm be made deterministic?

(3) The optimal $O(K+\log n)$ run-time is achievable only on expander graphs. What is the complexity of this problem on other families of graphs? (The lower bound given in the previous section may well be nonoptimal.)

(4) Find a distributed algorithm that solves the problem on any graph in a time that matches its specific lower bound.

Acknowledgment. We thank Nick Pippenger for many stimulating discussions and helpful ideas.

REFERENCES

[Ale] R. ALELIUNAS, *Randomized parallel communication*, in Proc. 1st Symposium on Principles of Distributed Computing, Ottawa, Canada, 1982, pp. 60-72.

[Al1] N. ALON, *Eigenvalues and expanders*, Combinatorica, 6 (1986), pp. 83-96.

[Al2] ———, *Expanders, sorting in rounds and superconcentrators of limited depth*, in Proc. 17th Annual ACM Symposium on Theory of Computing, Providence, RI, 1985, pp. 98-102.

[AKS] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 1-9.

[AHMP] H. ALT, T. HAGERUP, K. MEHLHORN, AND F. P. PREPARATA, *Simulation of idealized parallel computers on more realistic ones*, SIAM J. Comput., 16 (1987), pp. 808-835.

[C] H. CHERNOFF, *A measure of asymptotic efficiency for tests of hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493-507.

[H] W. HOEFFDING, *On the distribution of the number of successes in independent trials*, Ann. Math. Statist., 27 (1956), pp. 713-721.

[K] A. KARLIN, private communication.

[L1] T. LEIGHTON, *Tight bounds on the complexity of parallel sorting*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Washington, DC, 1984, pp. 71-80.

[L2] ———, private communication.

[LPS] A. LUBOTZKY, R. PHILLIPS, AND P. SARNAK, *Ramanujan conjecture and explicit constructions of expanders*, in Proc. 18th Annual ACM Symposium Theory of Computing, Berkeley, CA, 1986, pp. 240-246.

[PU] D. PELEG AND E. UPFAL, *The generalized packet routing problem*, Theoret. Comput. Sci., 53 (1987), 281-293.

[P] N. PIPPENGER, *Parallel communication with bounded buffers*, in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984, pp. 127-136.

[RV] J. H. REIF AND L. G. VALIANT, *A logarithmic time sort for linear size networks*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 10-16.

[T] R. M. TANNER, *Explicit concentrators from generalized N-gons*, SIAM J. Algebraic Discrete Meth., 5 (1984), pp. 287-293.

[Ul] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.

[U] E. UPFAL, *Efficient schemes for parallel communication*, J. Assoc. Comput., Mach., 31 (1984), pp. 507-517.

[V] L. G. VALIANT, *A scheme for fast parallel communication*, SIAM J. Comput., 11 (1982), pp. 350-361.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 244-257, April 1989

004

SCHEDULING PRECEDENCE GRAPHS IN SYSTEMS WITH INTERPROCESSOR COMMUNICATION TIMES*

JING-JANG HWANG†, YUAN-CHIEH CHOW‡, FRANK D. ANGER§, AND CHUNG-YEE LEE‡

Abstract. The problem of nonpreemptively scheduling a set of m partially ordered tasks on n identical processors subject to interprocessor communication delays is studied in an effort to minimize the makespan. A new heuristic, called Earliest Task First (ETF), is designed and analyzed. It is shown that the makespan ω_{ETF} generated by ETF always satisfies $\omega_{\text{ETF}} \leqq (2-1/n)\omega_{\text{opt}}^{(i)} + C$, where $\omega_{\text{opt}}^{(i)}$ is the optimal makespan without considering communication delays and C is the communication requirements over some immediate predecessor-immediate successor pairs along one chain. An algorithm is also provided to calculate C. The time complexity of Algorithm ETF is $O(nm^2)$.

Key words. multiprocessor scheduling, worst-case analysis, communication delays

AMS(MOS) subject classifications. 68Q20, 68Q25

1. Introduction. An extensively studied problem in deterministic scheduling theory is that of scheduling a set of partially ordered tasks on a nonpreemptive multiprocessor system of identical processors in an effort to minimize the overall finishing time, or "makespan." So much literature has been produced in the related area that a number of review articles have been published, including excellent summaries by Coffman [1], Graham et al. [3], and Lawler, Lenstra, and Rinnooy Kan [6]. The underlying computing system of this classical problem is thought to have no interprocessor overhead such as processor communication or memory contention. Such an assumption is a reasonable approximation to some real multiprocessor systems; therefore, applications can be found for the derived theory [5]. The assumption, however, is no longer valid for message-passing multiprocessors or computer networks, since interprocessor communication overhead is clearly an important aspect in such systems and is not negligible.

In this paper, interprocessor communication overhead is made part of the problem formulation and corresponding solutions are derived. The augmented multiprocessing model starts with a given set $\Gamma = \{T_1, T_2, \cdots, T_m\}$ of m tasks, each with processing time $\mu(T_i)$, and a system of n identical processors. The tasks form a directed acyclic graph (DAG) in which each edge represents the temporal relationship between two tasks and is associated with a positive integer $\eta(T, T')$, the number of messages sent from an immediate predecessor T to an immediate successor T' upon the termination of the immediate predecessor. The task model is called an "enhanced directed acyclic graph (EDAG)" and is denoted as a quadruple $G = G(\Gamma, \rightarrow, \mu, \eta)$.

To characterize the underlying system, a parameter $\tau(P, P')$ is introduced to represent the time to transfer a message unit from processor P to P'. The system is then denoted as $S = S(n, \tau)$, where n is the number of identical processors. By varying the values of $\tau(P, P')$, the system model can be used to model several types of systems such as a fully connected system, a local area network, or a hypercube. To accommodate the deterministic scheduling approach, we further assume that the communication subsystem is contention free. Mathematically speaking, the time to take $\eta(T, T')$ units of messages from P to P' is $\tau(P, P') \times \eta(T, T')$, a deterministic value. Figure 1 shows an example of the computing model and a feasible schedule.

* Received by the editors August 31, 1987; accepted for publication (in revised form) June 13, 1988.
† National Chiao Tung University, Hsinchu, Taiwan 30049, Republic of China.
‡ Department of Computer Science, University of Florida, Gainesville, Florida 32611.
§ Computer Science Department, Florida Institute of Technology, Melbourne, Florida 32901.

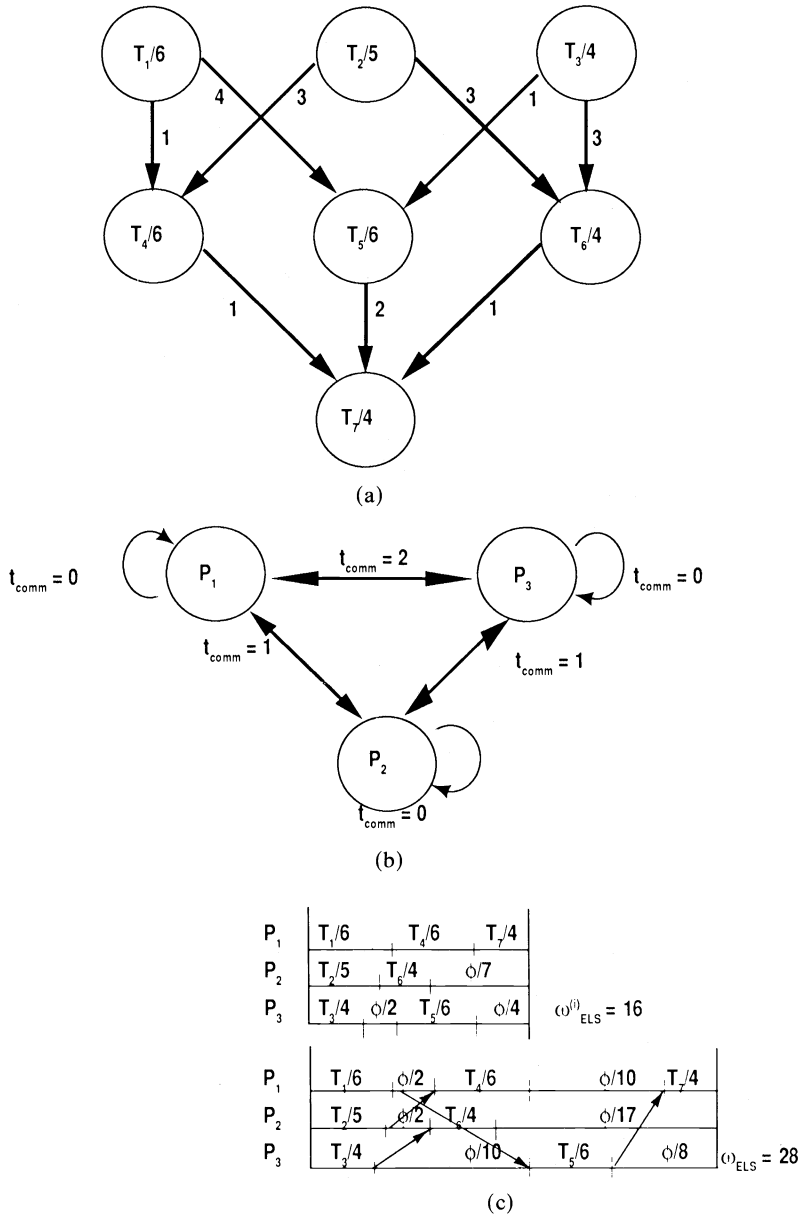


FIG. 1. *An example.* (a) *The* EDAG *model*; (b) *the system model*; (c) *schedules with and without communication delays.*

The computing model described above is an extension of Rayward-Smith's earlier model [7], which was confined to unit communication times (UCT) and unit execution times (UET). He shows that the problem of finding the minimum makespan is NP-complete and also presents a heuristic. The heuristic, called "generalized list scheduling," adopts the same greedy strategy as Graham's list scheduling [2]: No processor remains idle if there is some task available that it could process. For UET and UCT models, a task T can be processed on the processor P_i at time t if T has no immediate

predecessors, or each immediate predecessor of T has been scheduled (to start) on processor P_i at time $\leqq t-1$ or on processor $P_j \neq P_i$ at time $\leqq t-2$. A schedule using Rayward-Smith's heuristic always satisfies

$$\omega_g^c \leqq \left(3-\frac{2}{n}\right) \times \omega_{\text{opt}}^c - \left(1-\frac{1}{n}\right), \tag{1.1}$$

where ω_g^c is the length of the greedy schedule and ω_{opt}^c is the length of the optimal schedule.

This work generalizes Rayward and Smith's assumptions on communication times and execution times. The main result offers a new heuristic and its performance bound. The method is named "Earliest Task First (ETF)" and was first presented in Hwang's dissertation [4]. This paper simplifies the presentation, reduces its time complexity, and improves the performance bound. The main result indicates that the length of an ETF schedule is bounded by the sum of Graham's bound for list scheduling and the "communication requirement" over some immediate predecessor-immediate successor pairs along one chain that can be calculated using Algorithm C of § 3. In notation this is expressed as follows:

$$\omega_{\text{ETF}} \leqq \left(2-\frac{1}{n}\right)\omega_{\text{opt}}^{(i)} + C. \tag{1.2}$$

In (1.2), $\omega_{\text{opt}}^{(i)}$ denotes the optimal schedule length obtained by ignoring the interprocessor communication and is different from ω_{opt}^c in (1.1). The term $(2-1/n)\omega_{\text{opt}}^{(i)}$ is the Graham bound for list scheduling [2].

When ETF is applied to a problem with UET and UCT assumptions, its schedule length satisfies

$$\omega_{\text{ETF}} \leqq \left(3-\frac{1}{n}\right) \times \omega_{\text{opt}}^{(i)} - 1 \tag{1.3}$$

since $C \leqq \omega_{\text{opt}}^{(i)} - 1$. Both bounds ((1.1) and (1.3)) are close since 3 is the dominant factor in both expressions; but we should note that the bound on ETF uses $\omega_{\text{opt}}^{(i)}$, which is smaller than ω_{opt}^c in (1.1), as the base multiplier.

2. ETF—A new heuristic. Section 2.1 presents a simple approach, named ELS, for solving the scheduling problem formulated in § 1. ELS (extended list scheduling) is a straightforward extension of Graham's LS (list scheduling) method. The unsatisfactory performance of ELS motivates the development of ETF.

2.1. ELS—a simple solution. The ELS method adopts a two-phase strategy. First, it allocates tasks to processors by applying LS as if the underlying system were free of communication overhead. Second, it adds necessary communication delays to the schedule obtained in the first phase.

Graham's worst-case bound of LS as stated in (2.1) [2] provides a basis for deriving a similar bound for ELS:

$$\omega_{\text{LS}}^{(i)} \leqq \left(2-\frac{1}{n}\right)\omega_{\text{opt}}^{(i)}. \tag{2.1}$$

It is obvious that the difference between an ELS and an LS schedule, using the same task list L, is bounded by the total maximum communication requirement:

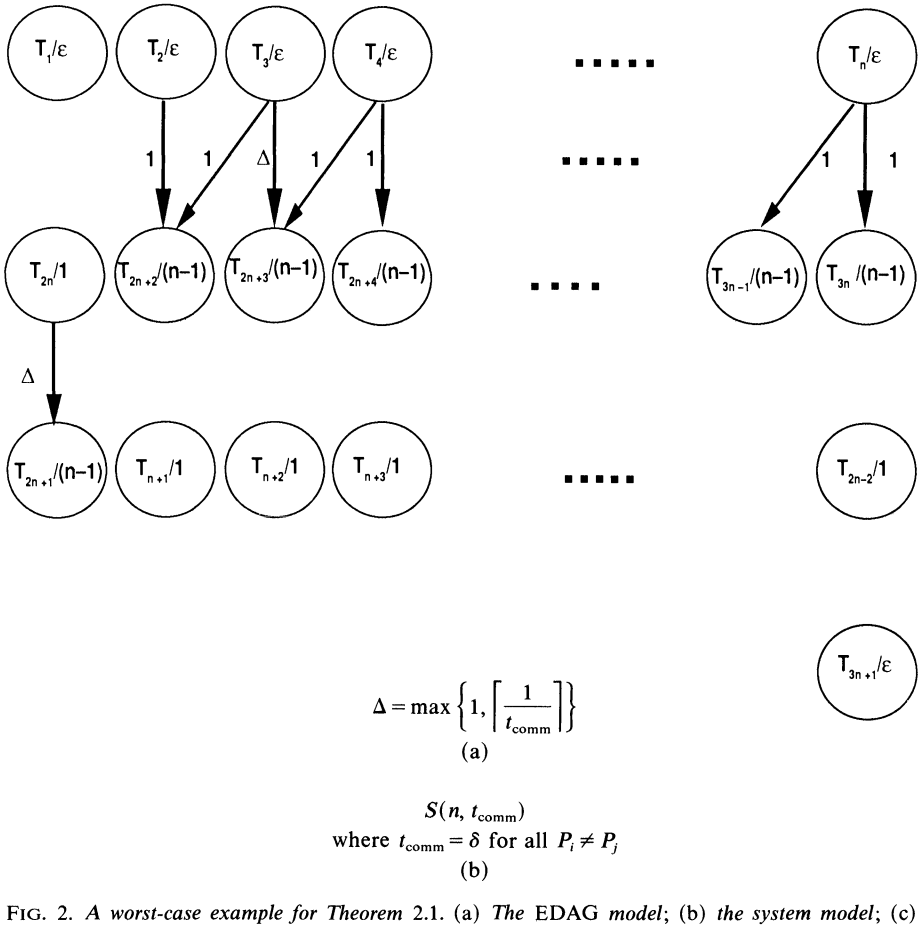
$$\omega_{\text{ELS}}(L) - \omega_{\text{LS}}^{(i)}(L) \leqq \tau_{\max} \times \sum_{T \in \Gamma, T' \in S_T} \eta(T, T'), \tag{2.2}$$

where $\tau_{\max} = \max\{\tau(P, P')\}$ is used to calculate the maximum communication requirements, and S_T denotes the set of immediate successors of T. We shall also use D_T to denote the set of immediate predecessors of T in later sections. Combining (2.1) and (2.2), we obtain (2.3):

$$\omega_{\mathrm{ELS}} \leqq \left(2 - \frac{1}{n}\right)\omega_{\mathrm{opt}}^{(i)} + \tau_{\max} \times \sum_{T \in \Gamma, T' \in S_T} \eta(T, T'). \tag{2.3}$$

The question raised here is whether the bound shown in (2.3) is the best possible. It is not hard to show that all maximum communication requirements may actually produce communication delays. However, to show that the bound stated in (2.3) is tight, it is necessary to produce an ELS schedule in which the schedule of computational tasks is the worst and, at the same time, almost no communications can be overlapped with the computations in the same schedule. This is not obvious since, in general, a longer schedule will allow more overlapping between computation and communication. Nevertheless, the example in Fig. 2 shows that the bound stated in (2.3) is asymptotically achievable as ε and δ approach zero. We summarize the result in Theorem 2.1.

THEOREM 2.1. *Any* ELS *schedule is bounded by the sum of Graham's bound and the total communication requirement* ((2.3)). *Furthermore, this bound is the best possible.*



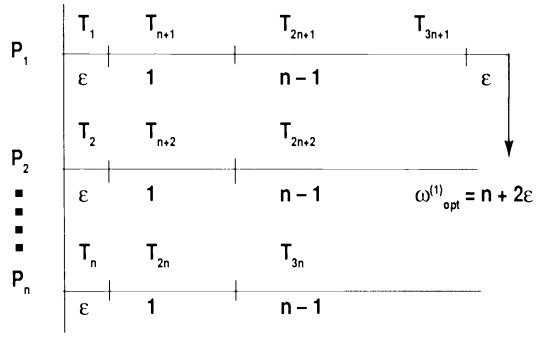
$$\Delta = \max\left\{1, \left\lceil \frac{1}{t_{\mathrm{comm}}} \right\rceil\right\}$$

(a)

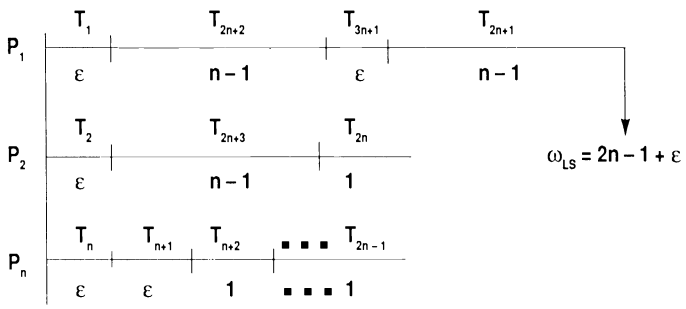
$S(n, t_{\mathrm{comm}})$
where $t_{\mathrm{comm}} = \delta$ for all $P_i \neq P_j$

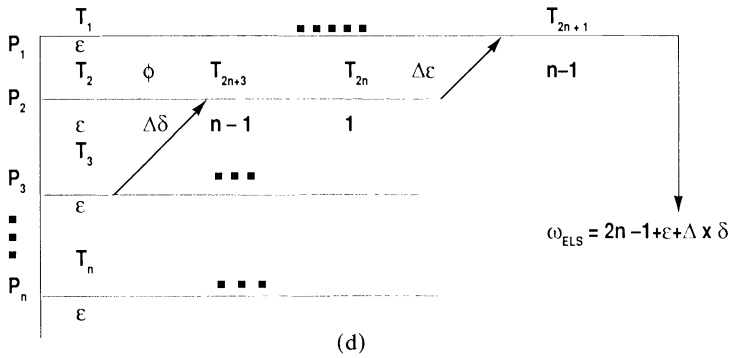
(b)

FIG. 2. *A worst-case example for Theorem* 2.1. (a) *The* EDAG *model*; (b) *the system model*; (c) *an optimal schedule on an ideal system*; (d) *an* LS *schedule and its corresponding* ELS *schedule.*



(c)





(d)

FIG. 2—continued

2.2. ETF—a new heuristic. The performance of ELS is unsatisfactory when dealing with communication delays, as indicated by Theorem 2.1. ETF, the core of this paper, has a much better performance bound. ETF adopts a simple greedy strategy: the earliest schedulable task is scheduled first. The algorithm is event-driven and is to be described in detail in the following paragraphs.

A task is called *available* when all its predecessors have been scheduled. Let A and I be the sets of available tasks and free processors, respectively, with $A = \{T: D_T = \varnothing\}$ and $I = \{P_1, \cdots, P_n\}$ initially. The starting time of T is denoted as $s(T)$; the finishing time is $f(T)$; the processor to which T is assigned is $p(T)$. These three values for all tasks are the output of the scheduling algorithm.

The starting time of an available task is determined by several factors: when its preceding tasks are finished, how long the communication delays take, and where the

task and its predecessors are allocated. Let $r(T, P)$ denote the time the last message for T arrives at processor P; mathematically,

$$r(T, P) = \begin{cases} 0 & \text{if } T \text{ has no predecessors,} \\ \max\limits_{T' \in D_T} \{f(T') + \eta(T', T) \times \tau(p(T'), P)\}. \end{cases} \tag{2.4}$$

Then the earliest starting time of an available task T can be calculated by

$$e_s(T) = \max \{CM, \min_{P \in I} \{r(T, P)\}\} \tag{2.5}$$

where CM, called "current moment," denotes the curent time of the event clock. By definition, I is the set of free processors at time = CM. Now, the earliest starting time among all available tasks, $e_s^\wedge$, is calculated as follows:

$$e_s^\wedge = \min_{T \in A} \{e_s(T)\}. \tag{2.6}$$

The greedy strategy of ETF wants to find $\hat{T} \in A$ and $\hat{P} \in I$ such that $e_s^\wedge = e_s(\hat{T}) = \max \{\text{CM}, r(\hat{T}, \hat{P})\}$.

Since we assume arbitrary communication delays in our model, a newly available task after the event clock is advanced may have an earlier starting time than that of the select task $\hat{T}$. To overcome such a difficulty, a second time variable, called "next moment" and abbreviated as NM, is introduced to keep track of the scheduling process. NM denotes the earliest time after CM at which one or more currently busy processors become free. NM is set to ∞ if all processors are free after CM. It is clear that the scheduler will not generate any available task that can be started earlier than $e_s^\wedge$ if $\text{NM} \geqq e_s^\wedge$. The scheduling decision will be made in this case; otherwise the event clock is advanced and the decision is postponed.

Conflicts may occur during the scheduling process when two available tasks have the same starting time. To resolve the conflicts, we adopt a priority task list L as we did in LS and ELS methods. Such a strategy is called ETF/LS. If the priority list L is obtained through the critical-path analysis, then the method is called ETF/CP. Algorithm ETF below is a simple ETF implementation in which conflicts are resolved arbitrarily.

Example. Consider the same EDAG task graph and the same three-processor system as given in Fig. 1. The ETF schedule is presented in Fig. 3. Table 1 shows the detail of the scheduling process. In Table 1, i denotes ith execution of the inner loop of executing Algorithm ETF when it is applied to the given problem. The values of CM and NM at the beginning of ith execution of the inner loop are denoted as $\text{cm}(i)$ and $\text{nm}(i)$, respectively. $\hat{T}$ is the available task selected when $\text{CM} = \text{cm}(i)$ and $\text{NM} = \text{nm}(i)$; $\hat{T} = \text{NONE}$ means no task available at the current moment. The scheduling

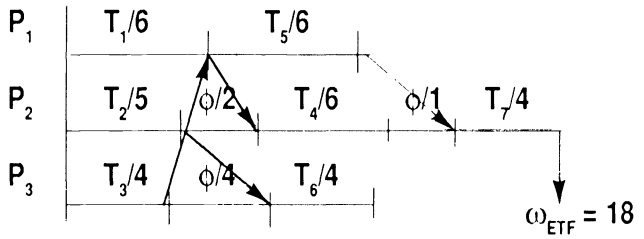


FIG. 3. *An* ETF *schedule.*

TABLE 1

I	CM (I)	NM (I)	$\hat{T}$	Decision
1	0	∞	T_1	made
2	0	6	T_2	made
3	0	5	T_3	made
4	0	4	None	
5	4	5	None	
6	5	6	T_6	postponed
7	6	∞	T_5	made
8	6	12	T_4	made
9	6	12	T_6	made
10	6	12	None	
11	12	13	None	
12	13	∞	T_7	made
13	All tasks have been scheduled			

decision of $\hat{T}$ is made if $e_s^{\wedge} \leqq \text{nm}(i)$ and postponed otherwise.

ALGORITHM ETF.

(0) Initialize: $I \leftarrow \{P_1, \cdots, P_n\}$, $A \leftarrow \{T: D_T = \varnothing\}$, $Q \leftarrow \varnothing$, CM $\leftarrow 0$, and NM $\leftarrow \infty$, $r(T, P) \leftarrow 0$ for each $T \in A$ and each $P \in I$.

(1) While $|Q| < m$ Do

begin

(1.1) While $I \neq \varnothing$ and A $\neq \varnothing$ Do

begin

(1.1.1) Find $\hat{T} \in A$ and $\hat{P} \in I$ such that
$\min_{T \in A} \min_{P \in I} r(T, P) = r(\hat{T}, \hat{P}) = \text{temp}$, let $e_s^{\wedge} = \max\{\text{CM}, \text{temp}\}$.

(1.1.2) If $e_s^{\wedge} \leqq$ NM
then
Assign $\hat{T}$ to run on $\hat{P}$: $p(\hat{T}) \leftarrow \hat{P}$; $s(\hat{T}) \leftarrow e_s^{\wedge}$; $f(\hat{T}) \leftarrow s(\hat{T}) + \mu(\hat{T})$,
$A \leftarrow A - \{\hat{T}\}$, $I \leftarrow I - \{\hat{P}\}$, append $\hat{T}$ to Q,
if $f(\hat{T}) \leqq$ NM then NM $\leftarrow f(\hat{T})$
else exit the inner loop.

end

(1.2) Proceed: CM $\leftarrow$ NM, find new NM.

(1.3) Repeat for each T, P such that T just finished on P:
$I \leftarrow I + \{P\}$, repeat for each $T' \in S_T$:
$d_{T'} = d_{T'} - 1$; if $d_{T'} = 0$ then T' is newly available

(1.4) For each newly available task T' do
$A \leftarrow A + \{A'\}$,
for each processor P, $r(T', P) = \max_{T \in D_{T'}} \{f(T) + \eta(T, T') \times \tau(p(T), P)\}$

end.

3. Analysis of ETF. Some properties of ETF are to be established in this section. The time complexity is analyzed first, followed by the three lemmas that help to explain the way ETF works. The establishment of a performance bound concludes this section.

THEOREM 3.1. *The time complexity of* ETF *is* $O(\text{nm}^2)$, *where n and m are the number of processors and the number of tasks, respectively.*

Proof. The ETF algorithm consists of a main loop containing an inner loop. It is obvious that the main loop repeats at most m times since some task "finishes" in step 1.2 each time. The running time depends on the number of repetitions of the inner

loop. We observe that there are two mutually exclusive possibilities after executing an iteration of the inner loop: (1) When $e_s^{\wedge} \leqq \mathrm{NM}$, the selected available task $\hat{T}$ was scheduled. (2) When $e_s^{\wedge} > \mathrm{NM}$ or $I = \varnothing$ or $A = \varnothing$, the inner loop was exited. For the latter case, the CM was moved forward to NM (by executing step 1.2, which follows the inner loop) and at least one task was finished at the new CM. Consequently, the inner loop repeats at most $2m$ times (since the algorithm either schedules a task or finishes at least one task for each iteration of the inner loop).

Each task becomes newly available once during the scheduling process; therefore, the total running time of step 1.4 is $O(\mathrm{nm}^2)$. The most time-consuming step is generally step 1.1.1, which takes $O(\mathrm{nm})$ for each execution and $O(\mathrm{nm}^2)$ for at most $2m$ executions. □

We introduce some conventions here. Let z be the number of iterations of statement 1.1, and let i be an index of a particular iteration, $i \leqq z$. By an execution of statement 1.1, we mean the execution of the whole inner loop. When $I = \varnothing$ or $A = \varnothing$, the statement 1.1 is also considered being executed once, though the substatements 1.1.1 and 1.1.2 are skipped. Let the sequences $\mathrm{cm}(1), \mathrm{cm}(2), \cdots, \mathrm{cm}(z)$ and $\mathrm{nm}(1), \mathrm{nm}(2), \cdots, \mathrm{nm}(z)$ denote values carried by variables CM and NM, respectively, at the beginning of each execution of statement 1.1. If the scheduling decision of a task T is made during the ith execution of statement 1.1, then i is called the *decision order* of T and denoted as $\mathrm{do}(T)$, and $\mathrm{cm}(i)$ is called the *decision time* of T and denoted as $\mathrm{dt}(T)$. Note that not every integer i, $1 \leqq i \leqq z$, is a decision order of some task. We define $e_s(i)$ as the earliest starting time among all available tasks while CM carries the value $\mathrm{cm}(i)$; $e_s(T, i)$ is similarly defined for each available task T and is calculated by (2.5).

LEMMA 1. *For any task* T, $\mathrm{dt}(T) \leqq s(T)$ *holds and there exists no task* T' *such that* $\mathrm{dt}(T) < f(T') < s(T)$ *or* $\mathrm{dt}(T) < \mathrm{dt}(T') < s(T)$.

Proof. The first part is obvious since each task was scheduled to start at a time no less than the value of the current moment at which the decision is made (see statements 1.1.1 and 1.1.2 of Algorithm ETF).

Suppose that T' is a task such that $\mathrm{dt}(T) < f(T') < s(T)$. Let $\mathrm{do}(T) = i$ and $\mathrm{do}(T') = j$. We have four cases:

(1) T was scheduled after $T'(i > j)$. We get $\mathrm{nm}(i) \leqq f(T')$ by definition and thus, $\mathrm{nm}(i) \leqq f(T') < s(T)$ follows. According to the decision principle (statement (1.1.2)), T should not be scheduled at the ith execution of statement 1.1. This contradicts $\mathrm{do}(T) = i$.

(2) T was scheduled before $T'(i < j)$ and T' has no predecessors. T' became available from the first execution of statement 1.1. Thus, $e_s(T, i) = s(T) > f(T') > s(T') = e_s(T', j) \geqq e_s(T', i)$. (The last inequality is due to the fact that the earliest starting time of a task is nondecreasing after it becomes available, since CM is nondecreasing (see (2.25)). The result $e_s(T, i) > e_s(T', i)$ implies that T' instead of T should have been selected in the ith execution of statement 1.1. This leads to a contradiction.

(3) T was scheduled before T' and $f(T^*) \leqq \mathrm{cm}(i)$, where T^* is a last finished predecessor of T'. The assumption $f(T^*) \leqq \mathrm{cm}(i)$ implies T' became available before the ith execution of statement 1.1. Thus, the same argument for case (2) leads to a contradiction.

(4) T was scheduled before T' and $f(T^*) > \mathrm{cm}(i)$, where T^* is a last finished predecessor of T'. For this case, we have $\mathrm{dt}(T) < f(T^*) < s(T)$. We replace T' by T^*, find the last predecessor of the new T', called T^* again, and repeat the same argument. One of the four cases may happen. If one of cases (1), (2), and (3) occurs, then we are done; otherwise the process is continued. Since a new task T^* with earlier finish

time is obtained in each cycle, we will reach the case $f(T^*) \leqq \text{cm}(i)$ in a finite number of cycles if case (4) occurs repeatedly. This concludes the proof of the second part.

Suppose now that $\text{dt}(T) < \text{dt}(T') < s(T)$. We can find a task T^* such that $f(T^*) = \text{dt}(T')$ since a decision is made only at some cm at which a task is finished. This contradicts the second part of this lemma. □

LEMMA 2. *If* $s(T) < s(T')$, *then* $\text{do}(T) < \text{do}(T')$ *and* $\text{dt}(T) \leqq \text{dt}(T')$.

Proof. Suppose $s(T) < s(T')$, $\text{do}(T) = i$, $\text{do}(T') = j$, $i > j$. Since the current-moment sequence is nondecreasing, two possibilities can be discussed: (1) $\text{dt}(T) > \text{dt}(T')$, or (2) $\text{dt}(T) = \text{dt}(T')$. For (1), we apply the first part of Lemma 1 and get $\text{dt}(T') < \text{dt}(T) \leqq s(T) < s(T')$. This contradicts the second part of Lemma 1. For (2) T and T' are available at $\text{cm}(i) = \text{cm}(j)$, and $e_s(T', j) = s(T') > s(T) = e_s(T, i) \geqq e_s(T, j)$. During the jth execution of statement 1.1, T should have been selected instead of T. This is a contradiction. □

LEMMA 3. *Let* T *and* T' *be two tasks satisfying* $f(T) < \text{dt}(T')$. *Then there exists an integer* i *such that* $\text{cm}(i) = f(T)$. *In other words, the current-moment sequence cannot skip any time point at which a task is finished until all tasks have been scheduled.*

Proof. Let $\text{do}(T') = j$. Then $0 = \text{cm}(1) \leqq f(T) < \text{cm}(j)$. Since the current-moment sequence is nondecreasing, an index k can be found such that $\text{cm}(k) \leqq f(T) < \text{cm}(k+1)$. If $\text{cm}(k) = f(T)$, then we are done. If $\text{cm}(k) < f(T) < \text{cm}(k+1)$ and T is not a task scheduled when $\text{CM} = \text{cm}(k)$, then T is a task whose scheduling decision was made earlier. Then $f(T) < \text{cm}(k+1) = \text{nm}(k)$ contradicts $\text{nm}(k) \leqq f(T)$. If $\text{cm}(k) < f(T) < \text{cm}(k+1)$ and T is a task scheduled at $cm(k)$, then NM should have been moved backward to $f(T)$, and thus $\text{cm}(k+1) = \text{nm}(k) = f(T)$ after the scheduling of T. This is also a contradiction. □

THEOREM 3.2. *For any* EDAG *task model* $G = G(\Gamma, \rightarrow, \mu, \eta)$ *to be scheduled on a system* $S = S(n, \tau)$, *the schedule length,* ω_{ETF}, *obtained by* ETF *always satisfies*

$$\omega_{\text{ETF}} \leqq \left(2 - \frac{1}{n}\right) \times \omega_{\text{opt}}^{(i)} + C_{\max} \tag{3.1}$$

where $C_{\max}$ *is the maximum communication requirement along all chains in* Γ. *That is,*

$$C_{\max} = \max\left\{\tau_{\max} \times \sum_{i=1}^{l-1} \eta(T_{c_i}, T_{c_{i+1}}) : (T_{c_1}, \cdots, T_{c_l}) \text{ is a chain in } \Gamma\right\}. \tag{3.2}$$

Proof. The set of all points of time in $(0, \omega_{\text{ETF}})$ can be partitioned into two subsets A and B. A is defined to be the set of all points for which all processors are executing some tasks. B is defined to be the set of all points of time for which at least one processor is idle (maybe all processors are idle due to simultaneous communication delays). If B is empty, then all processors complete their last assignment at ω_{ETF} and no idle interval can be found with $(0, \omega_{\text{ETF}})$. The ETF schedule is indeed optimal and thus the theorem holds obviously. We thus assume B is nonempty. In the interest of mathematical rigor, we suppose B is the disjoint union of q open intervals (b_{l_i}, b_{r_i}) as below:

$$B = (b_{l_1}, b_{r_1}) \cup \cdots \cup (b_{l_q}, b_{r_q})$$

where $b_{l_1} < b_{r_1} < b_{l_2} < b_{r_2} < \cdots < b_{l_q} < b_{r_q}$.

We claim that we can find a chain of tasks,

$$X : T_{J_l} \rightarrow T_{j_{l-1}} \rightarrow \cdots \rightarrow T_{j_1},$$

such that

$$\sum_{i=1}^{q} (b_{r_i} - b_{l_i}) \leqq \sum_{k=1}^{l} \mu(T_{j_k}) + \sum_{k=1}^{l-1} \tau_{\max} \times \eta(T_{j_{k+1}}, T_{j_k}). \tag{3.3}$$

In other words, the total length of B is no longer than the sum of all computational times and all pessimistic communication requirements along the chain X. We say that the chain X covers the set B.

Let T_{j_1} denote a task that finishes in the ETF schedule at time ω_{ETF}. Let $s(T_{j_1})$ denote the time at which T_{j_1} is started as scheduled by ETF. There are three possibilities regarding the starting time of T_{j_1}:

(1) $s(T_{j_1}) \leqq b_{l_1}$.

(2) $s(T_{j_1}) \in B$, i.e., there exists an integer h, $h \leqq q$, such that

$$b_{l_k} < s(T_{j_1}) < b_{r_h}. \tag{3.4}$$

(3) $s(T_{j_1}) \in A$ but $s(T_{j_1}) > b_{l_1}$, i.e., there exists an integer h, $h \leqq q-1$, such that

$$b_{r_k} \leqq s(T_{j_1}) \leqq b_{l_{k+1}} \quad \text{or} \quad b_{r_q} \leqq s(T_{j_1}). \tag{3.5}$$

If the first possibility occurs, then the task T_{j_1} by itself constitutes a chain that satisfies our claim. We shall show for the second and the third possibilities, respectively, that we can always add one or more tasks to the chain to extend the covered part of B to the left. To say a set of points is covered by a chain means that the length of the set is less than or equal to the sum of computation times and communication requirements along the chain, where communication requirements are estimated pessimistically (using $\tau_{\max}$).

Let us first consider the second possibility. Let h be the index satsifying (3.4). Then T_{j_1} alone has covered part of B from its right end to somewhere in between b_{l_k} and b_{r_k}. The mission here is to add the second task, T_{j_2}, to the chain. By the definition of B there is some processor P_α that was idle during $(s(T_{j_1})-\varepsilon, s(T_{j_1}))$ for some $\varepsilon > 0$. We consider three cases separately:

Case 1. T_{j_1} did not become available until $s(T_{j1})$.

Case 2. T_{j_1} became available at a time earlier than $s(T_{j_1})$ but later than time zero.

Case 3. T_{j_1} became available at time zero.

For Case 1, we simply take the last finished predecessor of T_{j_1} as T_{j_2}.

For Case 2, let T^* be the last finished immediate predecessor of T_{j_1}. Then $f(T^*) < s(T_{j_1})$. We further consider three subcases: (i) There exists no task T assigned to run P_α with $s(T) > s(T_{j_1}) - \varepsilon$. (ii) There exists at least one task T assigned to run on P_α with $s(T) > s(T_{j_1})$, but there exists no task T assigned on P_α with $s(T) = s(T_{j_1})$. (iii) There exists a task T assigned to run on P_α with $s(T) = s(T_{j_1})$.

For the first subcase (of Case 2), since no task is blocking T_{j_1} from starting earlier on P_α, it must be that $r(T_{j_1}, P_\alpha) \geqq s(T_{j_1})$. Let T_{j_2} be an immediate predecessor of T_{j_1} whose message to T_{j_1} arrives at P_α at time $r(T_{j_1}, P_\alpha)$. Let $p(T_{j_2}) = P_\beta$. Then $\tau(P_\beta, P_\alpha) \times \eta(T_{j_2}, T_{j_1}) = r(T_{j_1}, P_\alpha) - f(T_{j_2}) \geqq s(T_{j_1}) - f(T_{j_2})$. This result indicates that the communication requirement from T_{j_2} to T_{j_1} covers the interval $(f(T_{j_2}), s(T_{j_1}))$, and hence the covered part of B is shifted further left from $s(T_{j_1})$ after T_{j_2} is added to the chain. Furthermore, the parameter $\tau(P_\beta, P_\alpha)$ can be replaced by $\tau_{\max}$. This completes the first subcase.

For the second subcase (of Case 2), we let T_α be the first task allocated on P_α after $s(T_{j_1})$. By Lemma 2, do $(T_{j_1}) <$ do (T_α). The allocation of T_α on P_α cannot be a reason to block the possibility of T_{j_1} utilizing P_α at an earlier time. The reason T_{j_1} did not take such an advantage is again due to $r(T_{j_1}, P_\alpha) \geqq s(T_{j_1})$. T_{j_2} can therefore be found in the same way as in the first subcase.

We assume a task T_α satisfying $p(T_\alpha) = P_\alpha$ and $s(T_\alpha) = s(T_{j_1})$ in the last subcase. Now both do $(T_\alpha) >$ do (T_{j_1}) and do $(T_\alpha) <$ do (T_{j_1}) are possible. If do $(T_\alpha) >$ do (T_{j_1}), then T_α was scheduled later than T_{j_1}; hence, the same argument used in the previous subcase prevails. We need to argue why $r(T_{j_1}, P_\alpha) \geqq s(T_{j_1})$ when do $(T_\alpha) <$ do (T_{j_1}). Let

us assume do $(T_\alpha) = k$ and consider two situations separately: (1) dt $(T_\alpha) < s(T_\alpha)$, and (2) dt $(T_\alpha) = s(t_\alpha)$. For (1), at the beginning of the kth execution of the inner loop, both T_α and T_{j_1} were available. (T_{j_1} was available since $f(T^*) \leqq$ dt (T_α) guaranteed by Lemma 1.) If $r(T_{j_1}, P_\alpha) < s(T_\alpha)$, then T_{j_1}, and not T_α, should have been selected in the kth execution of the inner loop. For (2), we let cm be the largest value in the current-moment sequence satisfying $f(T^*) \leqq \text{cm} > s(T_\alpha)$. (The existence of such cm is guaranteed by Lemma 3.) At CM = cm, T_α was unscheduled and T_{j_1} was available. T_α could not block T_{j_1} from occupying P_α at a time earlier than $s(T_\alpha)$. If $r(T_{j_1}, P_\alpha) < s(T_{j_1})$, then T_{j_1} should have been allocated on P_α and started at an earlier time when CM carried the value cm. This contradiction implies that $r(T_{j_1}, P_\alpha) \geqq s(T_{j_1})$. Based on this result, the same technique as before can be applied to add a second task T_{j_2} to the chain finishing all subcases of Case 2.

Case 3 is indeed impossible. We can obtain $r(T_{j_1}, P_\alpha) \geqq s(T_{j_1})$ as we did above for Case 2; on the other hand, we have a contradictory fact that $r(T_{j_1}, P_\alpha) = 0$ due to $D_T = \varnothing$.

We have completed our discussion of the second possibility. Let us summarize the results obtained so far. Suppose $T_{j_1} < b_{r_h}$ for some $h \leqq q$. We constructed a second task T_{j_2} such that T_{j_2} precedes T_{j_1} and

$$\mu(T_{j_1}) + \mu(T_{j_2}) + \tau_{\max} \times \eta(T_{j_2}, T_{j_1}) \geqq (b_{r_q} - b_{l_q}) + \cdots + (b_{r_{h+1}} - b_{l_{h+1}}) + (b_{r_h} - s(T_{j_1})) + s((T_{j_1}) - s(T_{j_2})).$$

Since $s(T_{j_1})$ is an interior point of (b_{l_h}, b_{t_h}), at least some new portion (maybe all) of (b_{l_h}, b_{r_h}) has been covered now.

The location of $s(T_{j_2})$, once again, has three possibilities:

(1) $s(T_{j_2}) \leqq b_{l_1}$.

(2) $s(T_{j_2}) \in B$.

(3) $s(T_{j_2}) \in A$ but $s(T_{j_2}) > b_{l_1}$.

If the first possibility happens, then we are done. If not, we repeat our arguments to construct a third task or more tasks. (The third possibility may lead to adding more than one task in one cycle.) The cycle can be repeated until the starting time of the last added task satisfies the first possibility.

Now, it remains to show how to add tasks to the chain in the case of the third possibility (3.5). Suppose that h is the integer such that $b_{r_h} \leqq s(T_{j_1}) \leqq b_{l_{h+1}}$ or $h = q$ when $b_{r_q} \leqq s(T_{j_1})$. Let $\hat{\Gamma} = \{T\colon T$ is a predecessor of T_{j_1} satisfying $s(T) \geqq b_{r_h}$ or $T = T_{j_1}\}$. Clearly, $\hat{\Gamma} \neq \phi$. Let $T^\# \in \hat{\Gamma}$ such that $s(T) < b_{r_h}$ for any immediate predecessor T of $T^\#$. Then $T^\#$ is either a predecessor of T_{j_1} or T_{j_1} itself. In either case, we can always construct a sequence of tasks:

$$T^\# = T_{j_{d-1}} \to T_{j_{d-2}} \to \cdots \to T_{j_2} \to T_{j_1}.$$

Although these tasks are not scheduled during B, the last added task, $T^\#$, is closer to uncovered points of B. The task $T^\#$ will play the same role as T_{j_1} did in the discussion about the second possibility. The mission again is to add a task T_d to the chain. There are three cases:

Case 1. $T^\#$ did not become available until b_{r_h}.

Case 2. $T^\#$ became available at a time earlier than b_{r_h} but later than time zero.

Case 3. $T^\#$ became available at time zero.

We finish Case 1 by setting T_{j_d} to be the last finished predecessor of $T^\#$. Case 3 is impossible by the same argument for Case 3 of the second possibility.

For Case 2, let T^* be the last finished predecessor of $T^\#$. Then $f(T^*) < b_{r_h} \leqq s(T^\#)$. By the definition of B there is some processor P_α that was idle during the time $(b_{r_h} - \varepsilon, b_{r_h})$ for some $\varepsilon > 0$. We ask the same question as before. Why had $T^\#$ not

been allocated on P_α while it was available and P_α was idle? Let T_α be the task assigned to run on P_α at b_{r_h}. Then $f(T^*) \leqq \mathrm{dt}(T_\alpha) \leqq s(T_\alpha)$ by Lemma 1. As in the last subcase of Case 2 for the second possibility, (1) $\mathrm{do}(T_\alpha) > \mathrm{do}(T^\#)$ and (2) $\mathrm{do}(T_\alpha) < \mathrm{do}(T^\#)$ are distinguished. Two situations for (2) are further discussed separately: (1) $\mathrm{dt}(T_\alpha) < s(T_\alpha)$, and (2) $\mathrm{dt}(T_\alpha) = s(T_\alpha)$. The same techniques can be applied to conclude that $r(T^\#, P_\alpha) \geqq b_{r_h}$. Let T_{j_d} be the immediate predecessor whose message reaches P_α at $r(T^\#, P_\alpha)$ assuming $\hat{T}$ be allocated on P_α. This task is exactly what we want.

After T_{j_d} together with $T_{j_{d-1}}, \cdots, T_{j_2}$ is added to the chain, the enlarged chain clearly covers all of B from some point on, and this is a greater part of B than the initial one. We repeat the whole process by considering three possibilities regarding the position of $s(T_{j_d})$. The process is repeated until $s(T_{j_l}) \leqq b_{l_1}$ is satisfied by the added task in the chain. Finally, a chain satisfying our claim (see (3.3)) is constructed.

The important thing to note about this claim is

$$\sum_{\phi_i \in \Phi} \mu(\phi_i) \leqq (n-1) \times \sum_{k=1}^{l} \mu(T_{j_k}) + n \times \tau_{\max} \times \sum_{k=1}^{l-1} \eta(T_{j_{k+1}}, T_{j_k})$$

where the left-hand sum is over all empty tasks. (A processor is said to be executing an empty task when it is idle.) But the fact that $T_{j_l}, \cdots, T_{j_1}$ is a chain implies that it takes at least $\sum_{k=1}^{l} \mu(T_{j_k})$ to finish all tasks in Γ in any schedule, even on an ideal system, i.e.,

$$\omega_{\mathrm{opt}}^{(i)} \geqq \sum_{k=1}^{l} \mu(T_{j_k}).$$

The following inequality is obvious:

$$\sum_{T \in \Gamma} \mu(T) \leqq n \times \omega_{\mathrm{opt}}^{(i)}.$$

Consequently,

$$\begin{aligned}
\omega_{\mathrm{ETF}} &= \frac{1}{n}\left(\sum_{T_k \in \Gamma} \mu(T_k) + \sum_{\phi_i \in \Phi} \mu(\phi_i) \right) \\
&\leqq \frac{1}{n}\left(n\omega_{\mathrm{opt}}^{(i)} + (n-1)\omega_{\mathrm{opt}}^{(i)} + n \times \tau_{\max} \times \sum_{k=1}^{l-1} \eta(T_{j_{k+1}}, T_{j_k}) \right) \\
&= \left(2 - \frac{1}{n}\right)\omega_{\mathrm{opt}}^{(i)} + \tau_{\max} \times \sum_{k=1}^{l-1} \eta(T_{j_{k+1}}, T_{j_k}).
\end{aligned}$$

The theorem follows immediately by replacing the communication requirements along a particular chain by $C_{\max}$. □

The bound stated in Theorem 3.2 can be reduced by giving an algorithm to construct the task chain and the corresponding communication requirements used to cover idle times. The proof of Theorem 3.2 actually finds a chain X: $T_{j_l}, \cdots, T_{j_1}$ such that $|B| \leqq \sum_{k=1}^{l} \mu(T_{j_k}) + \textit{certain communication times.}$ The last term, called C in the following discussion, corresponds to the sum of just enough terms of the form $\tau(p(T_{j_{k+1}}), P_\alpha) \times \eta(T_{j_{k+1}}, T_{j_k})$ to "cover" the times at which (1) no task in the constructed chain is running, and (2) at least one processor is idle. Since it can frequently happen that all processors are busy while communication is taking place, much of the time $C_{\max}$ is "hidden" by being overlapped with computation of other tasks. We are therefore led to a better bound for ω_{ETF} that even coincides with Graham's bound if all communication is hidden. The algorithm assumes that all processors were idle for

$t<0$, and each task was scheduled to run during a closed time interval $[t_1, t_2]$. It calculates an upper bound C for the "unhidden" communication of the constructed chain.

ALGORITHM C
Input: An ETF Schedule
Output: C
(0) $C \leftarrow 0$, find a task T that was scheduled to finish last.
(1) Let $u = \text{l.u.b.}^1 \{t: t \leqq s(T)$ and at least one processor is idle at $t\}$.
(2) If $u = 0$ then output C and stop.
(3) If $u < s(T)$ then find a task T' such that T' is either a predecessor of T with $s(T') \geqq u$ or T itself but $s(T^*) < u$ for any immediate predecessor T^* of T'. Replace T by T'.
(4) Let T' be the last finished predecessor of T. If $f(T') = u$ then replace T by T' and go to (1).
(5) Find a processor P_α that was idle during $(u-\varepsilon, u)$ for some $\varepsilon > 0$, find a task $T' \in D_T$ such that $f(T') + \tau(p(T'), P_\alpha) \times \eta(T', T) = r(T, P_\alpha)$, $C \leftarrow C + \tau(p(T'), P_\alpha) \times \eta(T', T)$, replace T by T', go to step 1.

In general, C is much smaller than $C_{\max}$ depending on the case. It serves, however, to replace $C_{\max}$ in the performance bound of Theorem 3.2, and this is stated in the corollary below.

COROLLARY. *An* ETF *schedule is bounded from above by the sum of the Graham's bound and the output of Algorithm* C. *Mathematically,*

$$\omega_{\text{ETF}} \leqq \left(2 - \frac{1}{n}\right)\omega_{\text{opt}}^{(i)} + C. \tag{3.6}$$

4. Conclusion and discussion. An essential result of multiprocessor scheduling theory is the introduction of the list-scheduling method and the establishment of its performance guarantee. In this paper, we first extend the LS method to address the issue of communication delays. Theorem 2.1 indicates that the extended method, named ELS, fails to shorten communication delays, at least in worst cases. The new heuristic presented in this paper effectively reduces communication delays and also maintains the strength of the classical list scheduling.

It should be noted that the performance bound of ETF is made possible by the assumption of no communication contention; however, it is clear that the contention-free attribute alone does not guarantee the same bound.

Assume no communication contention is indispensable for maintaining the deterministic approach; but, strictly speaking, this assumption is only valid in fully connected systems or systems with contention-free protocols. For the later case, the protocol overhead should be included in the estimation of the communication parameters.

To expand the application domain of this theoretic work, we would rather consider the deterministic scheduling as a planning tool for task allocation suitable for most highly connected systems. Though each task is scheduled to start at some specific time, it will be started whenever enabled in the actual execution. In the case the actual start time of a task is later than its scheduled start time due to the queueing delay caused by communication contention, the succeeding tasks may become incapable of starting their execution at their scheduled times. This "postponement" may propagate. In the extreme case the postponement may propagate down to the critical execution path

[1] Least upper bound.

and then lengthen the total scheduled time of the graph. However, to some extent, queueing delay can be absorbed in noncritical paths. Only the portion of queueing delay that cannot be absorbed will lengthen the total graph's execution time. For this reason, the sensitivity of the no-communication contention assumption can be relaxed in real applications.

Acknowledgments. The authors are grateful to Drs. R. L. Graham and S. R. Kosaraju for their careful review and suggestions to the ETF algorithm. They are also indebted to the University of Florida and National Chiao Tung University. Without their support of the e-mail facility, the revision of the paper that required much communication across half the globe would have been very difficult.

REFERENCES

[1] E. G. COFFMAN, JR., ED., *Computer and Job-Shop Scheduling Theory*, John Wiley, New York, 1976.

[2] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[3] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: a survey*, Ann. Discrete Math., 5 (1979), pp. 287–326.

[4] J.-J. HWANG, *Deterministic scheduling in systems with interprocessor communication times*, Ph.D. dissertation, Computer and Information Sciences Department, University of Florida, Gainesville, FL, 1987.

[5] H. KASAHARA AND S. NARITA, *Parallel processing on robot-arm control computation on multiprocessing systems*, IEEE J. Robotics and Automation, (1985), pp. 104–113.

[6] E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Recent development in deterministic sequencing and scheduling: a survey*, in Deterministic and Stochastic Scheduling, M. H. Dempster, J. K. Lenstra, and A. H. G. Rinnooy Kan, eds., D. Reidel, Dordrecht, the Netherlands, 1982, pp. 367–374.

[7] V. J. RAYWARD-SMITH, UET *scheduling with interprocessor communication delays*, Internal Report SYS-C86-06, School of Information Systems, University of East Anglia, Norwich, United Kingdom, 1986.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 258-267, April 1989

005

FINDING AN APPROXIMATE MAXIMUM*

N. ALON†‡ AND Y. AZAR†

Abstract. Suppose that there are n elements from a totally ordered domain. The objective is to find, in a minimum possible number of rounds, an element that belongs to the biggest $n/2$, where in each round one is allowed to ask n binary comparisons. It is shown that $\log^* n + \Theta(1)$ rounds are both necessary and sufficient in the best algorithm for this problem.

Key words. searching, approximate maximum, parallel comparison algorithms

AMS(MOS) subject classification. 68E05

1. Introduction. Parallel comparison algorithms have received much attention during the last decade. The problems that have been considered include sorting [AA87], [AA88], [AAV86], [Ak85], [AKS83], [Al85], [AV87], [BT83], [BHe85], [HH81], [HH82], [Kn73], [Kr83], [Le84], [Pi86]; merging [BHo82], [HH82], [Kr83], [SV81]; selecting [AA88], [AKSS86a], [AP89], [Pi87], [Va75]; and approximate sorting [AA88], [AAV86], [AKSS86b], [BB87], [BR82]. The common model of computation considered is the parallel comparison model, introduced by Valiant [Va75], where only comparisons are counted. In this model, during each time unit (called a *round*) a set of binary comparisons is performed. The actual set of comparisons asked is chosen according to the results of the comparisons done in previous rounds. The objective is to solve the problem at hand, trying to minimize the number of comparison rounds as well as the total number of comparisons performed. Note that this model ignores the time corresponding to deducing consequences from comparisons performed, as well as communication and memory addressing time. However, in some situations this seems to be the relevant model. Moreover, any lower bound here applies to any comparison-based algorithm. There is an obvious, useful correspondence that associates each round of any comparison algorithm in the above parallel model with a graph whose vertices form the set of elements we have. The (undirected) edges of this graph are just the pairs compared during the round. The answer to each comparison corresponds to orienting the corresponding edge from the larger element to the smaller. Thus in each round we get an acyclic orientation of the corresponding graph, and the transitive closure of the union of the r oriented graphs obtained until round r represents the set of all pairs of elements whose relative order is known at the end of round r.

In many of the problems discussed so far in the parallel comparison model, the most interesting case is the one where the number n of elements is equal to the number of comparisons performed in each round. It is well known that in this case $\Theta(\log n)$ rounds are both necessary and sufficient for sorting. The lower bound follows trivially from the serial lower bound, and the upper bound follows from, e.g., the AKS sorting networks [AKS83]. As proved by Valiant, $\Theta(\log \log n)$ rounds are both necessary and sufficient for finding the maximum. The results of [AKSS86a] and [BHo82] show that the same $\Theta(\log \log n)$ bound holds for selecting and merging, respectively.

In the present paper we consider, motivated by the research on approximate sorting, another problem called the *approximate maximum* problem. This is the problem of finding, among n elements, an element whose rank belongs to the top $n/2$ ranks.

* Received by the editors January 10, 1988; accepted for publication (in revised form) May 18, 1988.

† Department of Mathematics and Computer Science, Sackler Faculty of Exact Sciences, Tel Aviv University, Ramat Aviv, Tel Aviv 69978, Israel.

‡ The research of this author was supported in part by an Allon Fellowship and by a grant from the United States-Israel Binational Science Foundation.

It is easy to show that in the serial comparison model this problem requires $n/2$ comparisons: only a constant factor better than the problem of finding the maximum. It is therefore rather surprising that with n comparisons in each round this problem can be solved much faster than that of finding the exact maximum in the same conditions. As it turns out, $\log^* n + \Theta(1)$ rounds are both necessary and sufficient for finding an approximate maximum among n elements, using n comparisons in each round. Moreover, the gap between the upper and lower bounds we obtain is only six rounds! The precise formulation of our result is the following. For $a \geq 1$, $k \geq 0$ define $a^{(k)}$ by $a^{(0)} = 1$ and $a^{(k)} = a^{a^{(k-1)}}$ for $k \geq 1$. Also define, as usual, $\log^* n = \min\{k : 2^{(k)} \geq n\}$. Let $r(n)$ denote the worst-case number of rounds of the best deterministic algorithm that finds an approximate maximum among n elements using n comparisons in each round. Our result is the following theorem.

THEOREM 1.1. *For every* $n \geq 2$,

$$\log^* n - 4 \leq r(n) \leq \log^* n + 2.$$

The upper bound here is not by explicit algorithm, as our algorithm uses certain random graphs. However, the known results about expanders easily supply (as in, e.g., [Al85], [Al86], [Pi87]) an explicit version of the algorithm, which will take about $\log^* n + 12$ rounds.

We note that our methods can be extended to the case where we have p comparisons in each round, as well as to the problem of finding an element whose rank is in the top εn ranks for all $p \geq 1$ and $1/n \leq \varepsilon \leq \frac{1}{2}$. With some additional work we can also obtain the corresponding results for approximate sorting. However, for the sake of simplicity we present here, in §§ 2 and 3, only the proof of Theorem 1.1 and only state the more general results (the detailed proofs of which will appear elsewhere) in the final § 4.

2. The upper bound. In this section we prove the upper bound, i.e., that using n processors we can find an element whose rank belongs to the top half of the ranks of the n elements in $\log^* n + 2$ rounds. Our algorithm uses some known results. The first is the algorithm of Valiant for finding the exact maximum. The others deal with properties of some random graphs (or explicit expanders). First we state a theorem from [Va75] and two lemmas from [Pi87].

THEOREM 2.1 [Va75]. *The maximum of* $n > 4$ *elements can be bound using* n *processors in* $\lceil \log \log n \rceil$ *rounds.* □

LEMMA 2.2 [Pi87]. *For every* m *and* a, *there is a graph with* m *vertices and* $2m^2 \log m/a$ *edges in which any two disjoint sets of* $a+1$ *vertices are joined by an edge.* □

LEMMA 2.3 [Pi87]. *If* m *elements are compared according to the edges of a graph in which any two disjoint sets of* $a+1$ *vertices are joined by an edge, then for every rank all but at most* $6a \log m$ *elements will be known to be too small or too large to have that rank.* □

For our algorithm we use Theorem 2.1 and a corollary of the last two lemmas. Note that Lemma 2.2 does not give an explicit construction of a graph with the specified properties; therefore, the algorithm seems to be nonexplicit. However, we can use some of the known results about explicit expanders (as in [Pi87], [Al85], [Al86]) to obtain an explicit version of our algorithm (with a slightly bigger additive constant). As the treatment for this case is analogous, we discuss here only the algorithm that uses Lemma 2.2.

PROPOSITION 2.4. *Assume we have* m *elements and* $p = 2m^2 \log m/a$. *Then, we can find in one round (using* p *comparisons) an element whose rank belongs to the top* $7a \log m$ *ranks of elements.*

Proof. Compare the elements according to the edges of the graph supplied by Lemma 2.2. By Lemma 2.3, all but at most $6a \log m$ elements will be known to be too small or too large to have rank $m - 7a \log m$. Therefore, at least $(7-6)a \log m$ elements will be known to belong to the top $7a \log m$ elements. □

For the description of our algorithm, we define the following function: $f(x) = 2^{x^{1/4}} \cdot x$, $x \geq 1$. f is a monotone increasing function and therefore $f^{-1}(y)$ exists (for $y \geq 2$) and is also a monotone (increasing) function. Define the following two sequences:

$b_0 = 2^{16}$ $b_{i+1} = f(b_i)$, $i \geq 0$ (We can easily check that all the $b_i's$ are integers),

$a_0 = n$, $a_{i+1} = f^{-1}(a_i)$, $i \geq 0$.

Define $k(n)$ by

$$k(n) = \min\{k : b_k \geq n\}. \tag{2.1}$$

By trivial induction using the monotonicity of f^{-1}, we get

$$a_i \leq b_{k-i} \quad \text{for } 0 \leq i < k(n). \tag{2.2}$$

Our algorithm is based on the following lemma, which is a consequence of the previous proposition.

LEMMA 2.5. *Assume we have $n \geq 2^{32}$ elements, partitioned into $m = \lceil n/f^{-1}(n) \rceil$ pairwise disjoint set, the ith having $t_i \leq \lceil f^{-1}(n) \rceil$ elements. Suppose that in each set we have an element that is smaller than at most εt_i elements in this set. Then we can find in one round using n processors an element smaller than at most $(\varepsilon + c/\sqrt{f^{-1}(n)})n$ elements among the n elements, where $c = 32$.*

Proof. Choose $a = \lceil 4m/\log^3 m \rceil$. Note that $m \geq 2^{16}$, $a \geq 4m/\log^3 m \geq 2^6$. Clearly, $(2m^2 \log m)/a \leq (m \log^4 m)/2 \leq n$, because by the definition of f: $\log^4 (n/f^{-1}(n)) = f^{-1}(n)$.

Thus we can use Proposition 2.4 for the m special elements with the n processors and find an element that belongs to the top $7a \log m$ elements out of the m elements. But

$$7a \log m = 7\left\lceil \frac{4m}{\log^3 m} \right\rceil \log m \leq 7\left(\frac{4m}{\log^3 m} + 1\right) \log m \leq 7 \frac{65}{64} \cdot \frac{4m}{\log^2 m} \leq \frac{30m}{\log^2 m}.$$

Therefore, the number of elements greater than this element is at most

$$\varepsilon n + \lceil f^{-1}(n) \rceil \frac{30m}{\log^2 m} \leq \varepsilon n + \left(1 + \frac{1}{2^{16}}\right) f^{-1}(n) \cdot \frac{30(1 + 1/2^{16}) n/f^{-1}(n)}{\log^2 (n/f^{-1}(n))}$$

$$\leq n\left(\varepsilon + \frac{32}{\log^2 (n/f^{-1}(n))}\right).$$

(In the last inequalities we used the facts that $f^{-1}(n)$, $m \geq 2^{16}$.)

Now $\log^2 (n/f^{-1}(n)) = \log^2 (2^{f^{-1}(n)^{1/4}}) = \sqrt{f^{-1}(n)}$. Hence we can find an element that is smaller than at most $(\varepsilon + (c/\sqrt{f^{-1}(n)}))n$, where $c = 32$. □

Now we can describe our algorithm and prove that it works.

THEOREM 2.6. *We can find an element the rank of which belongs to the top half of the ranks of $n \geq 2$ elements in $\log^* n + 2$ rounds using n processors.*

To obtain the last theorem, we prove by induction a more exact lemma.

LEMMA 2.7. *Let n be the number of elements and the number of processors. Then we can find in $k(n)+4$ rounds (where $k(n)$ is the number defined in (2.1)) an element that is smaller than at most $cn \sum_{i=0}^{k-2} 1/\sqrt{b_i}$ elements.*

Proof. We apply induction on n. The basic case is $n \leqq b_1$. In this case, we find the exact maximum using Theorem 2.1. Here $k \leqq 1$ so $cn \sum_{i=0}^{k-2} 1/\sqrt{b_i} = 0$, and we really find the exact maximum. To calculate the number of rounds we consider three cases. If $n \leqq 4$, four rounds are more than what is needed. If $n \leqq b_0$, then by Theorem 2.1 $\lceil \log\log n \rceil \leqq \lceil \log\log b_0 \rceil = \lceil \log\log 2^{16} \rceil = 4 = k(n)+4$. Otherwise $b_0 < n \leqq b_1$, $k(n) = 1$, and $\lceil \log\log n \rceil \leqq \lceil \log\log b_1 \rceil = \lceil \log\log 2^{32} \rceil = 5 = 1+4 = k(n)+4$. Assuming that the lemma is true for every $n' < n$, we prove it for n $(n > b_1 = 2^{32},\ k(n) \geqq 2)$. Split the n elements into $m = \lceil n/f^{-1}(n) \rceil$ pairwise disjoint sets, where the jth set has size $n_j \leqq \lceil f^{-1}(n) \rceil$. Assign to each set a number of processors equal to the size of the set. Now we can use the inductive assumption for each of the new sets, and find in $k'+4 \leqq k(\lceil f^{-1}(n) \rceil)+4$ rounds an element in each set, where the one in the jth set is smaller than at most $cn_j \sum_{i=0}^{k'-2} 1/\sqrt{b_i}$ elements in his set. By the definition of k, $n \leqq b_k$ so $f^{-1}(n) \leqq f^{-1}(b_k) = b_{k-1}$ and because the right-hand side is an integer $n_j \leqq \lceil f^{-1}(n) \rceil \leqq b_{k-1}$. Hence $k' \leqq k-1$; therefore we are allowed to have one more round to find the right element (among the m special elements). For that we use Lemma 2.5 and find an element smaller than at most

$$\left(\varepsilon + \frac{c}{\sqrt{f^{-1}(n)}}\right) n \leqq \left(\sum_{i=0}^{k-3} \frac{1}{\sqrt{b_i}} + \frac{1}{\sqrt{f^{-1}(n)}}\right) \cdot cn$$

other elements. Note that $b_{k-1} < n$ so $b_{k-2} \leqq f^{-1}(n)$, and hence the last expression is at most

$$cn\left(\sum_{i=0}^{k-3} \frac{1}{\sqrt{b_i}} + \frac{1}{\sqrt{b_{k-2}}}\right) = cn \sum_{i=0}^{k-2} \frac{1}{\sqrt{b_i}}.$$

Hence, we can find an element smaller than at most $cn \sum_{i=0}^{k-2} 1/\sqrt{b_i}$ elements among all the n elements in $k(n)+4$ rounds. This completes the proof of Lemma 2.7. □

To complete the proof of the main theorem, we just have to check the following two simple facts:

$$cn \sum_{i=0}^{\infty} \frac{1}{\sqrt{b_i}} \leqq \frac{1}{4} n \left(< \frac{1}{2} n \right), \tag{2.3}$$

$$k(n)+4 \leqq \log^* n + 2. \tag{2.4}$$

Inequality (2.3) is trivial, since $b_{i+1} = 2^{(b_i)^{1/4}} \cdot b_i \geqq 2^2 b_i$ or $1/\sqrt{b_{i+1}} \leqq \frac{1}{2}/\sqrt{b_i}$. Thus

$$c \sum_{i=0}^{\infty} \frac{1}{\sqrt{b_i}} \leqq \frac{2c}{\sqrt{b_0}} = \frac{64}{2^8} = \frac{1}{4}.$$

To prove (2.4) we need the following simple lemma:

LEMMA 2.8. *$b_i \geqq 2^8(2^{(i+2)})^4$ for every $i \geqq 0$.*

Proof. The proof is by induction on i. For $i = 0$, $b_0 = 2^{16} = 2^8(2^2)^4 = 2^8 \cdot (2^{(2)})^4$. Assuming the inequality holds for i, we prove it for $i+1$. Indeed, $b_{i+1} = 2^{(b_i)^{1/4}} \cdot b_i \geqq 2^{(2^8)^{1/4} \cdot 2^{(i+2)}} \cdot 2^8(2^{(i+2)})^4$ by the definition of b_{i+1} and by the inductive assumption. But the right-hand side is $\geqq 2^8 \cdot 2^{4 \cdot 2^{(i+2)}} = 2^8(2^{(i+3)})^4$, which completes the proof. □

Taking $i = \log^* n - 2$, we get that $b_i \geqq n$ and, therefore, $k(n) \leqq \log^* n - 2$. Thus, we have the complexity of the algorithm, which is $k(n)+4 \leqq \log^* n + 2$. This completes the proof of Theorem 2.6. □

3. The lower bound. It is convenient to establish our lower bound by considering the following (full information) game, called the *orientation game*, and played by two

players, the *graphs player* and the *order player*. Let V be a fixed set of n vertices. The game consists of *rounds*. In the first round the graphs player presents an undirected graph G_1 on V with at most n edges, and the order player chooses an acyclic orientation H_1 of G_1 and shows it to the graphs player, thus ending the first round. In the second round the graphs player chooses again, an undirected graph G_2 with at most n edges on V, and the order player gives it an acyclic orientation H_2, consistent with H_1 (i.e., the union of H_1 and H_2 is also acyclic), which he presents to the graphs player. The game continues in the same manner; in round i the graphs player chooses an undirected graph G_i with at most n edges on V, and the order player gives it an acyclic orientation H_i, such that the union $H_1 \cup \cdots \cup H_i$ is also acyclic. The game ends when, after, say, round r, there is a vertex v in V whose outdegree in the *transitive closure* of $H_1 \cup \cdots \cup H_r$ is at least $n/2$. The objective of the graphs player is to end the game as early as possible, and that of the order player is to end it as late as possible. The following fact states the (obvious) connection between the orientation game and the approximate maximum problem.

PROPOSITION 3.1. *The graphs player can end the orientation game in r rounds if and only if there is a comparison algorithm that finds an approximate maximum among n elements, using n comparisons in each round in r rounds.* □

In view of the last proposition and the results of the previous section, the graphs player can always end the orientation game in at most $\log^* n + O(1)$ rounds. A proof of existence of a strategy for the order player that enables him to avoid ending the orientation game in r rounds implies that $r+1$ is a lower bound for the time complexity of the approximate maximum problem. The next proposition is our main tool for establishing the existence of such a strategy for $r = \log^* n - 5$.

PROPOSITION 3.2. *There exists a strategy for the order player to maintain, for every $d \geqq 1$, the following property $P(d)$ of the directed acyclic graph constructed during the game.*

Property $P(d)$. Let $H(d) = H_1 \cup \cdots \cup H_d$ be the union of the oriented graphs constructed in the first d rounds. Then there is a subset $V_0 \subseteq V$ of size at most $|V_0| \leqq n/8 + n/16 + \cdots + n/2^{d+2}$ and a proper $D = 2000^{(d)}$-vertex-coloring of the induced subgraph of $H(d)$ on $V - V_0$ with color classes $V_1, V_2, \cdots, V_D$ (some of which may be empty), such that for each $i > j \geqq 1$ and each $v \in V_i$, v has at most 2^{i-j-2} neighbors in V_j. Furthermore, for every $i > j \geqq 0$ any edge of $H(d)$ that joins a member of V_i to a member of V_j is directed from V_i to V_j.

Proof. We apply induction on d. For $d = 1$, the graph $G_1 = (V, E_1)$ constructed by the graphs player has at most n edges. Let V_{00} be the set of all vertices in V whose degree is at least 32. Clearly,

$$|V_{00}| \leqq n/16. \tag{3.1}$$

Put $U = V - V_{00}$ and let K be the induced subgraph of G_1 on U. As the maximum degree in K is at most 31, K has, by a standard, easy result from extremal graph theory (see, e.g., [Bo78, p. 222]) a proper vertex-coloring by 32 colors and hence, certainly, a proper vertex coloring by 2000 colors. Let $U_1, U_2, \cdots, U_{2000}$ be the color classes. For every vertex u of K, let $N(u)$ denote the set of all its neighbors in K. For a permutation π of $1, 2, \cdots, 2000$ and any vertex n of K, define the π-degree $d(\pi, u)$ of u as follows. Let i satisfy $u \in U_{\pi(i)}$; then $d(\pi, u) = \sum_{j=1}^{i-1} |N(u) \cap U_{\pi(j)}|/2^{i-j}$. We claim that the expected value of $d(\pi, u)$ over all permutations π of $\{1, \cdots, 2000\}$, is at most $31/2000$. Indeed, for a random permutation π the probability that a fixed neighbor v of u contributes $1/2^r$ to $d(\pi, u)$ is at most $1/2000$ for every fixed $r > 0$. Hence, each neighbor contributes to this expected value at most $1/2000 \sum_{r>0} 1/2^r = 1/2000$ and the desired result follows, since $|N(u)| \leqq 31$.

Now consider the sum $\sum_{u\in U} d(\pi, u)$. The expected value of this sum (over all π's) is at most $31/2000|U|$, by the preceding paragraph. Hence, there is a fixed permutation σ such that $\sum_{u\in U} d(\sigma, u) \leqq 31/2000|U|$. Put $V_{01} = \{u \in U | d(\sigma, u) > \frac{1}{4}\}$. Clearly,

$$|V_{01}| \leqq \frac{4 \cdot 31}{2000}|U| \leqq \frac{|U|}{16} \leqq \frac{n}{16}.$$

Define $V_0 = V_{00} \cup V_{01}$, $W = U - V_{01} = V - V_0$. The last inequality together with (3.1) gives

$$|V_0| \leqq n/8.$$

Let F be the induced subgraph of G_1 on W and define $V_i = U_{\sigma(i)} \cap W$ $(1 \leqq i \leqq 2000)$. The V_i's clearly form a proper vertex coloring of F. Also, for every i, $1 \leqq i \leqq 2000$ and every $v \in V_i$

$$\sum_{j=1}^{i-1} \frac{|N(v) \cap V_j|}{2^{i-j}} < \frac{1}{4}$$

and hence v has at most 2^{i-j-2} neighbors in V_j for each j, $1 \leqq j < i$. Let H_1 be any acyclic orientation of G_1 in which all the edges that join a member of V_i to a member of V_j, where $i > j \geqq 0$, are directed from V_i to V_j (the edges inside V_0 can be directed in an arbitrary acyclic manner). Clearly $H(1) = H_1$ satisfies the property $P(1)$. Thus, the order player can orient G_1 according to H_1. This completes the proof of the case $d = 1$. □

Continuing the induction, we now assume that $H(r)$ has property $P(r)$ for all $r < d$, and prove that the order player can always guarantee that $H(d)$ will have property $P(d)$. We start by proving the following simple lemma.

LEMMA 3.3. *Let F be a directed acyclic graph with a proper g-vertex coloring with color classes $W_1, W_2, \cdots, W_g$. Suppose that for each $g \geqq i > j \geqq 1$ and each $v \in W_i$, v has at most 2^{i-j-2} neighbors in W_j, and that every edge of F whose ends are in W_i and W_j for some $i > j$ is directed from W_i to W_j. Then the outdegree of every vertex of F in the transitive closure of F is smaller than 4^g.*

Proof. Let v be an arbitrary vertex of F. The outdegree of v in the transitive closure of F is obviously smaller than or equal to the total number of directed paths in F that start from v. Suppose $v \in W_i$. Each such directed path must be of the form $v, v_{i_2}, v_{i_3}, \cdots, v_{i_r}$, where $i > i_2 > i_3 > \cdots i_r \geqq 1$, $v_{i_2} \in W_{i_2}, \cdots, v_{i_r} \in W_{i_r}$. There are 2^{i-1} possibilities for choosing $i_2, i_3, \cdots, i_r$. Also, as each vertex of the path is a neighbor of the previous one, there are at most 2^{i-i_2-2} possible choices for v_{i_2}, $2^{i_2-i_3-2}$ possible choices for v_{i_3} (for each fixed choice of v_{i_2}), etc. Hence, the total number of paths is at most $2^{i-1} \cdot 2^{i-i_2-2} \cdot 2^{i_2-i_3-2} \bullet \cdots \bullet 2^{i_{r-1}-i_r-2} < 2^g \cdot 2^{i-i_r} < 4^g$. This completes the proof of the lemma. □

Returning to the proof of Proposition 3.2, recall that $d \geqq 2$ and that by the induction hypothesis $H(d-1)$ has property $P(d-1)$. Thus, there is a subset $V_0 \subseteq V$ satisfying

$$|V_0| \leqq \frac{n}{8} + \frac{n}{16} + \cdots + \frac{n}{2^{d+1}} \tag{3.2}$$

and a proper $D = 2000^{(d-1)}$-vertex-coloring of the induced subgraph of $H(d-1)$ on $V - V_0$ with color classes $V_1, V_2, \cdots, V_D$ satisfying the conditions of property $P(d-1)$. Put $U = V - V_0$, let F be the induced subgraph of $H(d-1)$ on U, and let $T = (U, E(T))$ be the transitive closure of F. Let $G_d = (V, E_d)$ be the graph constructed by the graphs

player in round number d. Let $\bar{V}_{00}$ be the set of all vertices in U whose degree in G_d is at least $2^{d+4} \cdot 4^D$ and define

$$V_{00} = \bar{V}_{00} \cup \{u \in U : \exists v \in \bar{V}_{00} \text{ with } (v, u) \in E(T)\}.$$

Since G_d has at most n edges, $|\bar{V}_{00}| \leqq n/(2^{d+3} \cdot 4^D)$. Also, by Lemma 3.3, the outdegree of each $v \in \bar{V}_{00}$ in T is at most $4^D - 1$. Hence

$$|V_{00}| \leqq n/2^{d+3}. \tag{3.3}$$

Let $\bar{G}$ be the induced subgraph of G_d on $U - V_{00}$. Then the maximum degree in $\bar{G}$ is smaller than $2^{d+4} \cdot 4^D$. For each i, $1 \leqq i \leqq D$, let $\bar{G}^i$ denote the induced subgraph of $\bar{G}$ on $(U - V_{00}) \cap V_i$. As each $\bar{G}^i$ is a subgraph of $\bar{G}$, it has a proper vertex coloring with $2^{d+4} \cdot 4^D$ colors. For each i, $1 \leqq i \leqq D$, fix a proper n_i-vertex-coloring of $\bar{G}^i$ with color classes $U_{N_i+1}, U_{N_i+2}, \cdots, U_{N_i+n_i}$ (some of which may be empty), where $N_i = \sum_{j=1}^{i-1} n_j$ and

$$n_i \geqq 100 \cdot 2^{2d+7} \cdot 16^D \quad \text{for each } 1 \leqq i \leqq D \text{ and } \sum_{i=1}^{D} n_i = 2000^D. \tag{3.4}$$

(Note that since $D = 2000^{(d-1)}$, $d \geqq 2$, there is such a choice for the n_i's.) For every vertex u of $\bar{G}$, let $N(u)$ denote the set of all its neighbors in $\bar{G}$. Let us call a permutation π of $1, 2, 3, \cdots, \sum_{i=1}^{D} n_i$ *legal* if it maps each set of the form $\{N_i + 1, \cdots, N_i + n_i\}$ into itself (and only permute the elements inside these sets among themselves). For any vertex u of $\bar{G}$ and any legal permutation π, define the π-degree $d(\pi, u)$ as follows; let k satisfy $u \in U_{\pi(k)}$, then

$$d(\pi, u) = \sum_{j=1}^{k-1} |N(u) \cap U_{\pi(j)}|/2^{k-j}.$$

We claim that the expected value of $d(\pi, u)$, over all $\prod_{i=1}^{D} n_i!$ legal permutations, is at most $|N(u)|/\min_{1 \leqq i \leqq D} n_i \leqq 1/(100 \cdot 2^{d+3} \cdot 4^D)$. Indeed, consider a fixed neighbor v of u. If v belongs, as does u, to the same graph $\bar{G}^k$, then the probability that for a random legal permutation π, v will contribute $1/2^r$ to $d(\pi, u)$ is at most $1/n_k$, for each fixed $r > 0$. Otherwise, it is easy to check that this probability is even smaller. Hence, each neighbor contributes to this expected value at most $1/n_k \sum_{r>0} 1/2^r = 1/n_k$, and the claim follows.

Consider now the sum $\sum d(\pi, u)$, where u ranges over all vertices of $\bar{G}$. The expected value of this sum (over all legal permutations π) is at most $|V(\bar{G})|/(100 \cdot 2^{d+3} \cdot 4^D) \leqq n/(100 \cdot 2^{d+3} \cdot 4^D)$. Hence, there is a fixed legal permutation σ such that $\sum_{u \in V(\bar{G})} d(\sigma, u) \leqq n/(100 \cdot 2^{d+3} \cdot 4^D)$. Define $\bar{V}_{01} = \{u \in V(\bar{G}) : d(\sigma, u) > 1/100\}$ and $V_{01} = \bar{V}_{01} \cup \{u \in V(\bar{G}) : \exists v \in \bar{V}_{01} \text{ with } (v, u) \in E(T)\}$. Clearly, $|\bar{V}_{01}| \leqq n/(2^{d+3} 4^D)$ and, hence, by Lemma 3.3,

$$|V_{01}| \leqq n/2^{d+3}. \tag{3.5}$$

Put $V_0' = V_0 \cup V_{00} \cup V_{01}$, $W = V - V_0$. By (3.2), (3.3), and (3.5)

$$|V_0'| \leqq \frac{n}{8} + \frac{n}{16} + \cdots + \frac{n}{2^{d+2}}.$$

Let $\tilde{G}$ be the induced subgraph of $\bar{G}$ on W, and define $V_i' = U_{\sigma(i)} \cap W$ $(1 \leqq i \leqq 2000^D = 2000^{(d)})$. The sets V_i' clearly form a proper vertex coloring of $\tilde{G}$. Moreover, as each U_k is an independent set in $H(d-1)$, the sets V_i' actually form a proper vertex coloring of $H(d-1)$, as well. Moreover, for every i, $1 \leqq i \leqq 2000^{(d)}$, every $v \in V_i'$ satisfies

$$\sum_{j=1}^{i-1} \frac{|N(v) \cap V_j'|}{2^{i-j}} \leqq \frac{1}{100},$$

where $N(v)$ is the set of all neighbors of v in $\bar{G}$. Thus, for each fixed j, $1 \leqq j < i$, v has

at most $2^{i-j}/100$ neighbors in V'_j. Let H_d be any acyclic orientation of the edges of G_d obtained by orienting all the edges that join a member of V'_i and a member of V'_j, where $i>j\geq 0$, from V'_i to V'_j. The edges inside V'_0 are oriented in an arbitrary acyclic order consistent with the order given on $H(d-1)$. Notice that all the edges of $H(d-1)$ that do not lie inside V'_0 are also oriented from V'_i to V'_j with $i>j\geq 0$. In order to show that $H(d)=H(d-1)\cup H_d$ has the property $P(d)$, it remains to check that for every $i>j\geq 1$, every $v\in V'_i$ has at most 2^{i-j-2} neighbors in V'_j. By the construction, v has at most $2^{i-j}/100$ neighbors in V'_j in the new graph H_d. Recall that each V'_i is a subset of one of the sets V_k corresponding to the graph $H(d-1)$. Suppose $V'_i\subseteq V_k$, $V'_j\subseteq V_l$. Then $k\geq l$. If $l=k$ or $l=k-1$, then, since v has at most $[2^{k-l-2}]=0$ neighbors in V_l in the graph $H(d-1)$, it follows that in $H(d)$ v has at most $2^{i-j}/100\leq 2^{i-j-2}$ neighbors in V'_j, as needed. If $l\leq k-2$, observe that our construction implies that

$$(i-j)\geq(k-l-1)\min_{1\leq i\leq D} n_i\geq(k-l-1)\cdot 100\cdot 2^{2d+7}\cdot 16^D>(k-l)\cdot 100\geq 200.$$

Thus, in this case, the total number of neighbors of v in V'_j is at most $2^{i-j}/100+2^{k-l-2}\leq 2^{i-j}/100+2^{(i-j)/100}\leq 2^{i-j-2}$.

We conclude that the order player can orient G_d according to H_d, and maintain the property $P(d)$ of the graph $H(d)=H(d-1)\cup H_d$. This completes the induction and the proof of Proposition 3.2. □

The main result of this section, stated in Theorem 3.5 below, is an easy consequence of Proposition 3.2 and the following simple lemma.

LEMMA 3.4. *For every* $d\geq 1$, $2^{(d+3)}\geq 32\cdot 2000^{(d)}$.

Proof. We apply induction on d. For $d=1$ the inequality is trivial, as $2^{(4)}=2^{16}>64{,}000=32\cdot 2000^{(1)}$. Assuming it holds for $d-1$, we prove it for $d\geq 2$. By assumption $2^{(d+2)}\geq 32\cdot 2000^{(d-1)}$. Hence $2^{(d+3)}=2^{2^{(d+2)}}\geq 2^{32\cdot 2000^{(d-1)}}=(2^{32})^{2000^{(d-1)}}\geq(2^{21}\cdot 2000)^{2000^{(d-1)}}=(2^{21})^{2000^{(d-1)}}\cdot(2000)^{2000^{(d-1)}}>32\cdot(2000)^{(d)}$. □

THEOREM 3.5. *The order player can avoid ending the orientation game during the first* $\log^* n-5$ *rounds. Hence, by Proposition* 3.1, *the time required for finding an approximate maximum among n elements using n comparisons in each round is at least* $\log^* n-4$.

Proof. Clearly, we may assume that $\log^* n-5\geq 0$. By Proposition 3.2, the order player can maintain the property $P(d)$ for each of the graphs $H(d)$ constructed during the algorithm. Notice that by Lemma 3.3, the outdegree of every vertex in the transitive closure of a graph that satisfies $P(d)$ is at most $4^D+n/8+n/16+\cdots+n/2^{d+2}<4^D+n/4$, where $D=2000^{(d)}$. Thus, it follows that if $4^{2000^{(r)}}\leq n/4$, then the graphs player can keep playing for at least $r+1$ rounds. Therefore, by Lemma 3.4, the assertion of the theorem will follow if for $r=\log^* n-5$ the inequality $4^{2^{(r+3)}/32}\leq n/4$ holds. Since for $r>0$ $4\cdot 4^{2^{(r+3)}/32}<2^{(r+4)}$, this follows immediately from the definition of $\log^* n$. □

4. Extensions and related results. In this section we merely state, without proof, several extensions of the results of this paper and several related results. The proofs of these results combine the methods used here with some new ideas, somewhat similar to ones used in [AV87], [AAV86], [AP89]. The detailed proofs are somewhat complicated and will appear somewhere else.

For integers $n\geq 2$ and p, $1\leq p\leq\binom{n}{2}$, and for a real number ε, $1/n\leq\varepsilon\leq\frac{1}{2}$, let $r(n,p,\varepsilon)$ denote the time complexity of the best deterministic comparison algorithm that finds, among n elements, an element whose rank belongs to the top εn ranks, using p comparisons in each round. Clearly $r(n,n,\frac{1}{2})$ is just the function $r(n)$ discussed in this paper. For $\varepsilon=1/n$, the problem is that of finding the exact maximum, and the case $p=1$ corresponds to serial algorithms. We can prove the following theorem.

THEOREM 4.1. *For all admissible* n, p, ε,

$$r(n, p, \varepsilon) = \Theta\left(\frac{n}{p} + \log \frac{\log (1/\varepsilon)}{\log (2 + p/n)} + \log^* n - \log^* \left(1 + \frac{p}{n}\right)\right).$$

Thus for all n, $p \leqq 2n$, ε

$$r(n, p, \varepsilon) = \Theta\left(\frac{n}{p} + \log \log \frac{1}{\varepsilon} + \log^* n\right)$$

and for all n, $p \geqq 2n$, ε

$$r(n, p, \varepsilon) = \Theta\left(\log \frac{\log 1/\varepsilon}{\log (p/n)} + \log^* n - \log^* \left(\frac{p}{n}\right)\right).$$

For $\varepsilon = 1/n$ this theorem reduces to Valiant's result about finding the maximum. For $\varepsilon = \frac{1}{2}$, $p = n$ this reduces to our Theorem 1.1 (with a somewhat cruder estimate).

Next we consider approximate sorting. For $n \geqq 2$, $1 \leqq p \leqq \binom{n}{2}$, and $2/n^2 \leqq \varepsilon \leqq \frac{1}{2}$, let $a(n, p, \varepsilon)$ denote the time complexity of the best deterministic comparison algorithm that uses p comparisons in each round and finds, given n elements, all the order relations between pairs but at most $\varepsilon\binom{n}{2}$. The results of [BR82], [AA88], [AKSS86b], [BB87] deal with the minimum p for which $a(n, p, \varepsilon) = 1$ for some $\varepsilon = o(1)$. Notice that a precise determination of $a(n, p, \varepsilon)$ contains all the known results about deterministic comparison sorting or approximate sorting algorithms. We can prove the following result, determining $a(n, p, \varepsilon)$, up to a constant factor, for all possible n, p, and ε.

THEOREM 4.2. *For all admissible* n, p, ε

$$a(n, p, \varepsilon) = \Theta\left(\frac{\log 1/\varepsilon}{\log (1 + p/n)} + \log^* n - \log^*\left(1 + \frac{p}{n}\right)\right).$$

Thus, for $p \leqq 2n$, $a(n, p, \varepsilon) = \Theta(n \log (1/\varepsilon)/p + \log^* n)$ *and for* $p \geqq 2n$, $a(n, p, \varepsilon) = \Theta(\log (1/\varepsilon)/\log (p/n) + \log^* n - \log^* (p/n))$.

For $\varepsilon = 2/n^2$ this theorem corresponds to sorting and gives the known $\Theta(\log n/\log(1 + p/n))$ bound (which is $\Theta(n \log n/p)$ for $p \leqq 2n$ and is $\Theta(\log n/\log (p/n))$ for $p \geqq 2n$), (see [AV87], [AAV86]). Notice that for $p = n$ and for any $\varepsilon > 1/2^{\log^* n}$, $a(n, n, \varepsilon) = \Theta(\log^* n)$. As shown in § 3, $\Omega(\log^* n)$ rounds are required (with $p = n$), even if we wish to find one element known to be greater than $n/2$ others. By the last equality, $O(\log^* n)$ rounds are already sufficient to get almost all the order relations between pairs.

Finally, we consider the problem of approximate merging. In this case the results and the methods are simpler and similar to the methods of [Va75], [BHo82]. For n, $1 \leqq p \leqq n^2$ and $1/2n^2 \leqq \varepsilon \leqq \frac{1}{2}$, let $m(n, p, \varepsilon)$ denote the time complexity of the best comparison merging algorithm that uses p comparisons in each round and finds, given two sorted lists, each of size n, all the order relation between pairs but at most εn^2.

The results of [Va75], [BHo82] deal with full merging, i.e., the case $\varepsilon < 1/n^2$. We can prove the following theorem that determines $m(n, p, \varepsilon)$, up to a constant factor, for all admissible n, p, ε.

THEOREM 4.3. *For all admissible* n, p *and* $1/n \leqq \varepsilon \leqq \frac{1}{2}$,

$$m(n, p, \varepsilon) = \Theta\left(\frac{1}{\varepsilon p} + \log \frac{\log 1/\varepsilon}{\log (2 + \varepsilon p)}\right).$$

Thus for $p \leqq 2/\varepsilon$ $m(n, p, \varepsilon) = \Theta(1/(\varepsilon p) + \log \log 1/\varepsilon)$ and for $p \geqq 2/\varepsilon$ $m(n, p, \varepsilon) = \Theta(\log (\log 1/\varepsilon/\log \varepsilon p))$. For the case $\varepsilon \leqq 1/n$, the bounds are the same as for $\varepsilon = 1/n$

(up to a constant factor), which are the same bounds as for exact merging: $\Theta(n/p + \log(\log n/\log(2+p/n)))$.

Acknowledgment. We thank N. Pippenger, who brought the problem of finding an approximate maximum to our attention.

REFERENCES

[AA87] N. ALON AND Y. AZAR, *The average complexity of deterministic and randomized parallel comparison-sorting algorithms*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, Los Angeles, CA, 1987, pp. 489–498; SIAM J. Comput., 17 (1988), pp. 1178–1192.

[AA88] ———, *Sorting, approximate sorting and searching in rounds*, SIAM J. Discrete Math., 1 (1988), pp. 261–276.

[AAV86] N. ALON, Y. AZAR, AND U. VISHKIN, *Tight complexity bounds for parallel comparison sorting*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, 1986, pp. 502–510.

[Ak85] S. AKL, *Parallel Sorting Algorithm*, Academic Press, New York, 1985.

[AKSS86a] M. AJTAI, J. KOMLÓS, W. L. STEIGER, AND E. SZEMERÉDI, *Deterministic selection in $O(\log \log n)$ parallel time*, Proc. 18th Annual ACM Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 188–195.

[AKSS86b] ———, *Almost sorting in one round*, Adv. Comput. Res., to appear.

[AKS83] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium in Theory of Computing, 1983, pp. 1–9; *Sorting in $c \log n$ parallel steps*, Combinatorica, 3(1983), pp. 1–19.

[Al85] N. ALON, *Expanders, sorting in rounds and superconcentrators of limited depth*, in Proc. 17th Annual ACM Symposium on Theory of Computing, Providence, RI, 1985, pp. 98–102.

[Al86] ———, *Eigenvalues, geometric expanders, sorting in rounds and Ramsey Theory*, Combinatorica, 6(1986), pp. 207–219.

[AP89] Y. AZAR AND N. PIPPENGER, *Parallel selection*, Discrete Appl. Math., to appear.

[AV87] Y. AZAR AND U. VISHKIN, *Tight comparison bounds on the complexity of parallel sorting*, SIAM J. Comput., 3 (1987), pp. 458–464.

[BB87] B. BOLLOBÁS AND G. BRIGHTWELL, *Graphs whose every transitive orientation contains almost every relation*, Israel J. Math., 59 (1987), pp. 112–128.

[Bo78] B. BOLLOBÁS, *Extremal Graph Theory*, Academic Press, London, New York, 1978.

[BR82] B. BOLLOBÁS AND M. ROSENFELD, *Sorting in one round*, Israel J. Math., 38 (1981), pp. 154–160.

[BT83] B. BOLLOBÁS AND A. THOMASON, *Parallel sorting*, Discrete Appl. Math., 6 (1983), pp. 1–11.

[BHe85] B. BOLLOBÁS AND P. HELL, *Sorting and Graphs*, in Graphs and Orders, I. Rival, ed., D. Reidel, 1985, Boston, MA, pp. 169–184.

[BO86] B. BOLLOBÁS, *Random Graphs*, Academic Press, New York, 1986, Chap. 15.

[BHo82] A. BORODIN AND J. E. HOPCROFT, *Routing, merging and sorting on parallel models of computation*, in Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, 1982, pp. 338–344.

[HH80] R. HÄGGKVIST AND P. HELL, *Graphs and parallel comparison algorithms*, Congr. Numer., 29 (1980), pp. 497–509.

[HH81] ———, *Parallel sorting with constant time for comparisons*, SIAM J. Comput., 10 (1981), pp. 465–472.

[HH82] ———, *Sorting and merging in rounds*, SIAM J. Algebraic Discrete Methods, 3 (1982), pp. 465–473.

[Kn73] D. E. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Kr83] C. P. KRUSKAL, *Searching, merging and sorting in parallel computation*, IEEE Trans. Comput., 32 (1983), pp. 942–946.

[Le84] F. T. LEIGHTON, *Tight bounds on the complexity of parallel sorting*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Washington, DC, 1984, pp. 71–80.

[Pi87] N. PIPPENGER, *Sorting and selecting in rounds*, SIAM J. Comput., 6 (1987), pp. 1032–1038.

[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.

[Va75] L. G. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

SIAM J. COMPUT
Vol. 18, No. 2, pp. 268-277, April 1989

006

BOUNDS ON UNIVERSAL SEQUENCES*

AMOTZ BAR-NOY†‡, ALLAN BORODIN†§¶, MAURICIO KARCHMER†§, NATHAN LINIAL†, AND MICHAEL WERMAN†

Abstract. Universal sequences for graphs, a concept introduced by Aleliunas [M.Sc. thesis, University of Toronto, Toronto, Ontario, Canada, January 1978] and Aleliunas et al. [*Proc. 20th Annual Symposium on Foundation of Computer Science*, 1979, pp. 218-223] are studied. By letting $U(d, n)$ denote the minimum length of a universal sequence for d-regular undirected graphs with n nodes, the latter paper has proved the upper bound $U(d, n) = O(d^2 n^3 \log n)$ using a probabilistic argument. Here a lower bound of $U(2, n) = \Omega(n \log n)$ is proved from which $U(d, n) = \Omega(n \log n)$ for all d is deduced. Also, for complete graphs $U(n-1, n) = \Omega(n \log^2 n / \log \log n)$. An explicit construction of universal sequences for cycles ($d = 2$) of length $n^{O(\log n)}$ is given.

Key words. universal sequences, graph connectivity, complexity theory

AMS(MOS) subject classifications. 05C40, 68R10, 68Q15

1. Introduction. In addition to their obvious computational interest, graph connectivity problems play a central role in complexity theory. Let STCON (respectively, USTCON) denote the problem of determining if a directed (respectively, undirected) graph has a path from a given source node s to a given goal node t. As usual, let NSPACE(S) (respectively, DSPACE (S) and RSPACE (S)) denote those sets accepted in nondeterministic (respectively, deterministic and random) space S. Savitch's [7] fundamental result that NSPACE (S) $\subseteq$ DSPACE (S^2) is based on the fact that STCON is complete for NSPACE ($\log n$) with respect to log-space reducibility. (In fact, it is complete with respect to log-depth $= NC^1$ reducibility.) Similarly, Lewis and Papadimitriou [6] show that USTCON is complete for symmetric log-space bounded computation.

It is easy to see that STCON $\subseteq NC^2 \subseteq$ DSPACE ($\log^2 n$). However, despite considerable attention to this problem, there has been no improvement to this upper bound. In one of the few significant attempts to give evidence that STCON is not contained in DSPACE ($\log n$), Cook and Rackoff [4] introduce the JAG (Jumping Automata for Graphs) model and show that within this restricted model STCON requires space $\Omega(\log^2 n / \log \log n)$.

Although USTCON appears to be a computationally easier problem (and indeed Cook and Rackoff [4] cannot prove such a strong result for JAGs operating on undirected graphs), the best known deterministic algorithms for USTCON also apply to STCON. However, when we consider random space bounded computations, the situation seems to be different, since Aleliunas et al. [2] show that USTCON is in RSPACE ($\log n$).

Motivated by the Cook and Rackoff [4] paper, Aleliunas [1] (for the special case of degree two) and then Aleliunas et al. [2] introduce the concept of a "universal sequence" for graphs. Let $G(d, n)$ denote the class of all connected d-regular graphs with n nodes and with labeled edges. Think of every edge as a pair of directed edges.

* Received by the editors July 15, 1986; accepted for publication (in revised form) June 20, 1988.

† Computer Science Department, Hebrew University, Jerusalem 91904, Israel.

‡ Present address, Computer Science Department, Stanford University, Stanford, California 94305-2140.

§ Present address, Department of Computer Science, University of Toronto, Ontario, Canada M5S 1A7.

¶ This research was performed while this author was a Lady Davis Visiting Professor at the Hebrew University, Jerusalem 91904, Israel.

Every directed edge is labeled with a label from $\{0, \cdots, d-1\}$ in such a way that edges going out from the same vertex are labeled differently. (It is easy to verify that for $2 \leqq d \leqq n-1$, $G(d, n)$ is not empty if and only if dn is even, see Lovász [5, Ex. 5.2.] A sequence $s = s_1 s_2 \cdots s_r$ in $\{0, \cdots, d-1\}^*$ is interpreted as a sequence of edge traversal commands. Thus a sequence s and a node u_0 on a graph G in $G(d, n)$ define a unique sequence of nodes $u_0, u_1, \cdots, u_r$ in G with (u_{i-1}, u_i) labeled by s_i for $i = 1, \cdots, r$. We say that s *visits* the set of nodes $\{u_0, \cdots, u_r\}$. A sequence s *covers* a graph G in $G(d, n)$ if the sequence visits every node in G independent of the starting node. A sequence s is *universal* for $G(d, n)$ if s covers every G in $G(d, n)$. Finally, we let $U(d, n)$ denote the minimal length of a universal sequence for $G(d, n)$. We will see in § 5 that the restriction to regular graphs serves some aesthetic purposes.

Aleliunas et al. [2] show that the expected time for a random walk to visit all nodes of $G = (V, E)$ is at most $2|E||V|$. (No such result holds for directed graphs.) Hence the result that USTCON is in RSPACE $(\log n)$. They then use this result to assert the existence of a (nonuniform) universal sequence $s(d, n)$ for $G(d, n)$. The length of $s(d, n)$ is asymptotically bounded by

$$dn^2 \log (|G(d, n)|) = O(d^2 n^3 \log n).$$

In fact, they argue that most sequences of this length must be universal. Clearly, such universal sequences give a nonuniform method to test connectivity (using only two pebbles in the JAG model) within $O(\log n)$ space.

There are a number of reasons to study $U(d, n)$ further. If we could obtain a "sufficiently" explicit construction of polynomial length, then USTCON would be in DSPACE $(\log n)$. (We need to be able to generate any element of the sequence in DSPACE $(\log n)$.) In order to beat the previously mentioned $\log^2 n$ deterministic space bound, it suffices to show, by an explicit construction, that $U(3, n) = n^{o(\log n)}$. In this regard, we should also note that at present there is no deterministic sublinear space algorithm that runs in polynomial time. Second, for the purpose of time-space tradeoffs, it is important to determine the asymptotic behavior of $U(d, n)$ by any type of construction since lower bound techniques tend to apply to nonuniform models. In this regard an $U(d, n) = O(dn)$ or even $O(n^2)$ lower bound would have serious implications for any attempt at time-space lower bounds. In addition to complexity theory, universal sequences may play a role in the study of distributed systems (e.g., anonymous rings). And finally, of course, we think that the study of $U(d, n)$ raises a number of interesting combinatorial problems.

In §§ 2 and 3 we consider the special case of $d = 2$, the subject of Aleliunas [1]. First we give an explicit construction of length $n^{O(\log n)}$. Then we prove a nonlinear lower bound, $U(2, n) = \Omega(n \log n)$. Section 4 considers the other extreme, namely the case of complete graphs $(d = n-1)$. Here we observe that the probabilistic bound yields an upper bound of $n^3 \log^2 n$. We are able to prove that $U(n-1, n) = \Omega(n \log^2 n / \log \log n)$. In order to establish this lower bound, we view the problem as a game consisting of a graph generator (perhaps thought of as a taxi driver) versus a very powerful sequence generator (thought of as a passenger) where the passenger wants to see all n sites in as little time as possible and the driver would like to prolong the tour as long as possible. In § 5 we discuss the implication of the previous results for arbitrary d.

2. An explicit construction for the case of $d = 2$. There is only one regular connected graph of degree two, namely a cycle. In order to study $U(2, n)$, there is an equivalent way to formulate the problem as first discussed by Aleliunas in [1]. Instead of considering different labelings of the n-cycle, we consider the infinite line and label

each integer coordinate (= node) with a "0" or "1" with the interpretation that at a node labeled "b" the edge to the right is labeled "b" and the edge to the left is labeled "$1-b$." We now interpret $U(2, n)$ as the smallest length of a sequence that is guaranteed to visit at least n nodes on any labeled line.

From the probabilistic constructions of Aleliunas [1] and Aleliunas et al. [2] we know that $U(2, n) = O(n^3)$. In fact, Aleliunas [1] conjectured $U(2, n)$ to be exactly $\binom{n}{2}$. Exhaustive tests confirm $U(2, n) = \binom{n}{2}$ for $n < 8$ but we are aware of at least one claim (again by testing) that $U(2, n) < \binom{n}{2}$ for $n = 9$.

To gain insight for both a lower bound and an explicit construction, we first consider a special class of labeled lines. Let ODD denote the class of labeled lines of the form $\cdots 0^{i_1}1^{i_2}0^{i_3}1^{i_4}\cdots$, where all i_j are odd. Let $L(n)$ be the class of all sequences that cover at least n nodes on any line in ODD and let $U(n)$ denote the minimal length of any sequence in $L(n)$. Without loss of generality, we shall assume that n is even in order to avoid ceilings and floors. Let $n' = n+1$ so that n' is odd.

LEMMA 1. *The sequence* $w_n = (0^{n'}1^{n'})^{(n/2+1)}$ *has the following properties*:

(A) *If begun on the leftmost node of a block labeled with zeros (respectively, on the rightmost node of a block labeled with ones)* w_n *will move right (respectively, left) until encountering the first block of nodes with an even number of zeros or ones wherein it will terminate on the leftmost zero or rightmost one of this block. If no such block is encountered within the first n nodes visited, the sequence will be exhausted having visited at least n nodes.*

(B) *If not started on a leftmost zero or rightmost one the directional behavior of* w_n *on the line will depend on the parity of the initial location within the block on which the sequence is started. In any case the sequence will either visit n nodes or will terminate on the leftmost zero or rightmost one of some block of even length. In particular,* $w_n \in L(n)$ *and* $U(n) \leqq (n+1)^2$.

LEMMA 2. *The sequence* v_n *defined recursively by* $v_1 = 01$ *and* $v_n = v_{n/2}(0^{n'}1^{n'})v_{n/2}$ *is in* $L(n)$ *so that* $U(n) = O(n \log n)$.

We leave it to the reader to verify both lemmas. Let us remark that Theorem 2 of the next section shows that Lemma 2 is asymptotically optimal. We use the w_n sequences repeatedly to explicitly construct a universal sequence of length $n^{O(\log n)}$. We chose to use the w_n sequences for ODD rather than the shorter $O(n \log n)$ sequences v_n since its properties are easier to state and since the shorter length v_n would not significantly change the length of the universal sequence of Theorem 1.

THEOREM 1. *There is a recursively defined sequence* $s(n)$ *that is universal for* $G(2, n)$ *with length* $|s(n)| = n^{O(\log n)}$. *Furthermore, any bit of* $s(n)$ *can be computed in time bounded by a polynomial in n.*

Proof. By induction on n we construct $s(n)$. The basis of the induction is immediate. Let w_n be as in Lemma 1 and let $s(n/2) = s_1 s_2 \cdots s_t$. Then, $s(n) = w_n s_1 w_n s_2 \cdots w_n s_t w_n$. Consider any labeled line and mark the leftmost zero and rightmost one in every even-length block. Note that in a segment of length n at most $n/2$ nodes have been marked. Now after the first w_n, $s(n)$ has either visited n nodes or has positioned itself on a marked node. Once on a marked node, a sequence symbol s_i of $s(n/2)$ will move either left or right so that the next w_n (by Lemma 1) will continue to move in that direction until it is stopped at the next marked node. In this way we are guaranteed to visit at least n nodes with some w_n or at least $n/2$ marked nodes and all the nodes within the blocks containing those marked nodes. In either case, at least n nodes have been visited.

To bound the length we see that $|s(n)| \leqq (n+1)^2 |s(n/2)|$ from which the length bound easily follows. It is also easy to see how to compute any particular bit of $s(n)$ in polynomial time. □

3. A lower bound for the case of $d=2$. The aim of this section is to prove an $\Omega(n \log n)$ lower bound for $U(2, n)$. We shall pick a small subset of ODD and show that, just to traverse this subset, a sequence must be "long."

We begin by introducing some notation. Let S be the set of infinite lines of the form: $\cdots 0^a 1^a 0^a 1^a \cdots$, a odd. Let S_n be the set of segments of length n of lines in S with starting point a leftmost zero. We show that a sequence that traverses every line in S_n must be $\Omega(n \log n)$ long. We say that a sequence α is *good* for a segment w if, when started at the left of w, α eventually reaches the right of w. For a string $w \in \{0, 1\}^*$ and $x \in \{0, 1\}$ let $\#_x w$ be the number of occurrences of x in w.

We illustrate the idea of our proof by considering some very simple sequences. So let α cover every line in S_n where α is of the form $0^{r_1}1^{r_1}\cdots 0^{r_k}1^{r_k}$ with all r_i odd. Assume, moreover, that α covers at least $n/2$ locations to the right of the starting point.

FACT 1. $0^{r_i}1^{r_i}$ is good for 0^a1^a if and only if $r_i \geqq a$ and r_i is odd.

Define a_j to be the biggest odd number less than or equal to $n/(2j)$. Fact 1 implies that at least j r_i's must be bigger than a_j so that

$$|\alpha| \geqq \sum_{j=1}^{n/2} a_j \leqq \sum_{j=1}^{n/2} \left(\frac{n}{2j} - 2\right) = \Omega(n \log n).$$

We now give a lower bound for arbitrary α's. Consider the runs of a sequence α and the sequence 1α on a line from S. Since we start from a leftmost zero, these runs are symmetric with respect to the starting point. For a sequence β, let R_β (L_β) be the set of indices j such that β covers at least $n/2$ nodes to the right (left) of the starting point when run on the line $\cdots 0^{a_j}1^{a_j}0^{a_j}1^{a_j}\cdots$. Either $\sum_{j\in R_\alpha} a_j = \Omega(n \log n)$ or $\sum_{j\in L_\alpha} a_j = \Omega(n \log n)$. Since $R_\alpha = L_{1\alpha}$ and the lengths of α and 1α differ only by one we assume, without loss of generality, that $\sum_{j\in R_\alpha} a_j = \Omega(n \log n)$. We deal only with the runs of α on lines $\cdots 0^{a_j}1^{a_j}0^{a_j}1^{a_j}\cdots$, where $j \in R_\alpha$.

Fact 1 motivates the following lemma.

LEMMA 3. *Let α' be good for 0^a1^a, then $\alpha' = u_1\beta_a^0 u_2 \beta_a^1 u_3$, where*

(C1) $\#_0\beta_a^0 - \#_1\beta_a^0 = a$, *and* $\#_1\beta_a^1 - \#_0\beta_a^1 = a$;

(C2) *Every nonempty prefix and suffix of $\beta_a^0(\beta_a^1)$ has more zeros (ones) than ones (zeros).*

Proof. (C1) is necessary in order to pass the block of zeros or ones. (C2) can be obtained by extending u_1 to the right and u_2 to the left so as to make β_a^0 minimal, and extending u_2 to the right and u_3 to the left to make β_a^1 minimal. Note that $\#_0 u_2 = \#_1 u_2$. □

We denote $\beta_a^0 u_2 \beta_a^1$ by β_a and call it an a-block. β_a^0 and β_a^1 are called half blocks.

LEMMA 4. *Let α' be good for $(0^a1^a)^m$; then α' contains m disjoint a-blocks.* □

We say that two half blocks have a *trivial intersection* if they are either disjoint or one is contained in the other.

LEMMA 5. *Let β_{a_i}, β_{a_j} be an a_i-block and a_j-block respectively; then $\beta_{a_i}^0$ and $\beta_{a_j}^1$ have a trivial intersection.*

Proof. Follows by the prefix and suffix properties of $\beta_{a_i}^0$ and $\beta_{a_j}^1$. □

We say that a sequence $\{\beta_{a_j}^{x_j}\}_{j=1,\cdots,r}$; $x_j \in \{0, 1\}$, of half blocks is *nested* if we have the following.

(i) Every two half blocks have a trivial intersection;

(ii) $\beta_{a_j}^x \subseteq \beta_{a_k}^x$ implies that there exists an $l \neq j, k$ such that $\beta_{a_j}^x \subseteq \beta_{a_l}^{\bar{x}} \subseteq \beta_{a_k}^x$, $\bar{x}$ being the complement of x.

LEMMA 6. *Let α cover every line in S_n. Then α contains a nested sequence $\{\beta_{a_i}^{x_i}\}$, $i = 1, \cdots, n/2$ of half blocks where, again, a_j is the biggest odd number less than or equal to $n/(2j)$.*

Proof. The proof is by induction on i. Let B_{a_i} be a set of disjoint a_i-blocks in α. By Lemma 4 and the fact that $ia_i \leqq n$, $|B_{a_i}| \geqq i$. When $i=1$, pick any half block of any β_{a_1}. Assume the lemma is true for $i-1$. Let $B=\{\beta_{a_j}^{x_j}\}_{j=1,\cdots,i-1}$ be the nested sequence constructed up to step $i-1$. We will show how to find a half block of B_{a_i} that preserves the *nestedness* of B.

For each $\beta_{a_i} \in B_{a_i}$ define

$$In(\beta_{a_i})=\{j \mid \beta_{a_i}^{x_j} \cap \beta_{a_j}^{x_j} \neq \varnothing \wedge \beta_{a_i}^{\bar{x}_j} \not\subseteq B_{a_j}^{x_j}\}.$$

Intuitively, $In(\beta_{a_i})$ is the set of half blocks that prevent β_{a_i} from being properly nested.

CLAIM 1. $\{In(\beta_{a_i})\}$, $\beta_{a_i} \in B_{a_i}$, are pairwise disjoint. To see this, suppose that $j \in In(\beta_{a_i}) \cap In(\tilde{\beta}_{a_i})$ for some β_{a_i} and $\tilde{\beta}_{a_i}$ and without loss of generality assume that β_{a_i} appears in α to the left of $\tilde{\beta}_{a_i}$. Furthermore, assume that $\beta_{a_j}^{x_j}$ is of type 0, i.e., $x_j=0$. Then, either $\beta_{a_j}^0 \cap \beta_{a_i}=\varnothing$ or if $\beta_{a_j}^0 \cap \beta_{a_i} \neq \varnothing$, then $\beta_{a_i}^1 \subseteq \beta_{a_j}^0$ (because of suffix and prefix properties of the blocks). In either case, we get $j \notin In(\beta_{a_i})$. If $x_j=1$, a similar argument shows that $j \notin In(\tilde{\beta}_{a_i})$.

CLAIM 2. There exists a β_{a_i} such that $In(\beta_{a_i})=\varnothing$. This is true by Claim 1 and the fact that there are i a_i-blocks in B_{a_i}.

Choose a β_{a_i} as in Claim 2 and consider a minimal (in the inclusion sense) $\beta_{a_j}^{x_j}$ such that $\beta_{a_j}^{x_j} \cap \beta_{a_i} \neq \varnothing$. If no such $\beta_{a_j}^{x_j}$ exists then β_{a_i} is disjoint to every $\beta_{a_j}^{x_j}$ so that we can pick any half block of it. Otherwise, $\beta_{a_i}^{\bar{x}_j} \subseteq \beta_{a_j}^{x_j}$ and letting $x_i=\bar{x}_j$ we have that $\{\beta_{a_j}^{x_j}\}_{j=1,\cdots,i-1} \cup \{\beta_{a_i}^{x_i}\}$ is properly nested. □

LEMMA 7. *Let $B=\{\beta_{a_j}^{x_j}\}_{j=1,\cdots,i}$ be a nested sequence. Then*

$$\left|\bigcup_{j=1}^{i} \beta_{a_j}^{x_j}\right| \geqq \sum_{j=1}^{i} a_j.$$

Proof. Without loss of generality, assume that the half blocks in B are ordered so that $\beta_{a_i}^{x_i}$ is not contained in $\beta_{a_j}^{x_j}$ for $j=1,\cdots,i-1$. We proceed by induction on i. The case $i=1$ is obvious. Assume the lemma is true for $i-1$ so that $|\bigcup_{j=1}^{i-1} \beta_{a_j}^{x_j}| \geqq \sum_{j=1}^{i-1} a_j$.

By (ii) in the definition of nested sequences, we have that the maximal $\beta_{a_j}^{\bar{x}_j} \subseteq \beta_{a_i}^{x_i}$ are of opposite type, i.e., $x_i=\bar{x}_j$. Let β be the union of the $\beta_{a_j}^{\bar{x}_j}$, which are maximal half blocks contained in $\beta_{a_i}^{x_i}$. For $\beta_{a_i}^{x_i}$ to have (C1) we need $|\beta_{a_i}^{x_i} - \bigcup_{j=1}^{i-1} \beta_{a_j}^{x_j}| \geqq a_i$ so that $|\bigcup_{j=1}^{i} \beta_{a_j}^{x_j}| \geqq \sum_{j=1}^{i} a_j$. □

THEOREM 2. *A universal sequence α for S_n satisfies $|\alpha|=\Omega(n \log n)$.*

Proof. The proof follows immediately from Lemmas 6 and 7. □

4. The complete graph. There is only one graph in $G(n-1, n)$, namely K_n. While connectivity is no longer an issue, the question of $U(n-1, n)$ is still surprisingly difficult and interesting.

The probabilistic construction of Aleliunas et al. [2] shows that $U(n-1, n)=O(n^5 \log n)$ as the upper bound for any d. In fact, a more specific probabilistic analysis of random walks in K_n shows that the expected length to visit all nodes is $O(n \log n)$. From this follows $U(n-1, n)=O(n^3 \log^2 n)$.

For the lower bound, we consider the following two-player game, played between D (the driver) and P (the passenger). There is a fixed integer n and the game is with a taxi moving on a graph in $G(d, n)$. The game starts at any node of the graph. At each step, P can either direct the taxi along a directed edge that has already been traversed before, or he/she may let D choose any untraversed edge. In particular, if the present node is being visited for the first time, then D moves. The game ends at the first time when all nodes of the graph have been visited. P pays D the number of steps the game took.

We denote by $DP(d, n)$ the value of this game. Our result is Theorem 3.

THEOREM 3.

$$n^2-3n+3 \geqq DP(n-1, n) \geqq \Omega\left(\frac{n \log^2 n}{\log \log n}\right).$$

COROLLARY 1.

$$U(n-1, n) \geqq \Omega\left(\frac{n \log^2 n}{\log \log n}\right). \qquad \square$$

Proof of Theorem 3. The upper bound is obvious. P plays a strategy according to which he lets D play all the time. This insures that no edge is traversed twice in the same direction, so when $(n-1)(n-2)+1$ steps elapse all nodes must be visited.

To prove the lower bound, we consider a strategy for D that is defined inductively. At any time T in the game there is a digraph B_T on $V(K_n)$ consisting of all directed edges traversed thus far. The induction hypothesis follows:

(*) D has a strategy that causes the game to last at least $T(n) = (n \log^2 n)/(30 \log \log n)$ steps in such a way that all indegrees in B_T do not exceed $\log^2 n$.

We proceed to show how the strategy is carried over from n to $2n$. In the first stage D applies (*) to the first n nodes, thus he stays there at least $T(n) = (n \log^2 n)/(30 \log \log n)$ steps, with no indegree exceeding $\log^2 n$. At the first time after $T(n)$ at which D gets the right to move, he moves to node $n+1$. Now for another $T(n)$ steps he stays at nodes $\{n+1, \cdots, 2n\}$ according to (*). After these two stages, which take more than $2T(n)$ steps, no indegree exceeds $\log^2 n$ and node $2n+1$ is still isolated. Now begins a merging stage. To carry out the induction we show that for $(n \log n)/(5 \log \log n)$ steps D can proceed in the game with no indegree exceeding $\log^2 (2n)$. Since $T(2n)-2T(n) \leqq (n \log n)/(5 \log \log n)$ the induction hypothesis is maintained. We claim the following easy lemma.

LEMMA 8. *Let G be a digraph, $S \subseteq V(G)$, and let $d \geqq 3$ be the largest indegree in G. Then there are at least $|V|/2$ nodes u for which all paths from u to S have length at least* $(\log |V| - \log |S| - 1)/\log d$.

Now let us consider the set S of all nodes of largest outdegree. Whenever D is given the move, he chooses to go to a node whose indegree is strictly less than $\log^2 (2n)$ and whose distance in B_T from S is as large as possible. Note that the average indegree is at most $T(2n)/(2n) = (\log^2 (2n))/(30 \log \log (2n))$ so that all but $2n/(30 \log \log (2n)) = o(n)$ nodes have indegree strictly less than $\log^2 (2n)$. Thus during the whole process there are always many nodes with small indegree. In our case, all indegrees are at most $\log^2 (2n)$ and it follows by Lemma 8 that for every node u and for at least half of the nodes v, the distance from v to u is at least $\log n/(5 \log \log n)$. Therefore, if $|S| = 1$ then with this strategy the game proceeds at least $\log n/(5 \log \log n)$ steps more before the maximum outdegree increases. Ignoring momentarily the possibility that $|S|$ is larger, no degree reaches $2n$ before $(n \log n)/(5 \log \log n)$ steps. As long as the maximum outdegree is less then $2n$ the missing node will not be reached and the induction hypothesis is established since $T(2n)-2T(n) \leqq (n \log n)/(5 \log \log n)$.

To complete the proof for arbitrary $|S| \geqq 1$, consider the number of steps needed to increase the outdegree by two. We investigate a segment of the game during which the largest outdegree in B_T increases from m to $m+2$. Let us concentrate on that step where for the first time some outdegree reaches $m+2$ and let us say that at this point the number of nodes of degree $m+1$ is k. If $k \geqq (2 \log n)/(5 \log \log n)$ then our claim about the number of steps remains valid since the increase of the outdegree of any

node requires at least one move (of D). On the other hand, if $k < (2 \log n)/(5 \log \log n)$ then by Lemma 8 at the beginning of the stage there is a node whose distance from all k nodes of outdegree $m+1$ is at least $\log (n/k)/\log (\log^2 n) > (2 \log n)/(5 \log \log n)$ and, all of our previous arguments carry through. Thus $DP(n) \geqq T(n) = (n \log^2 n)/(30 \log \log n)$ and the proof is complete. □

5. Bounds for arbitrary graphs. Given the $U(2, n) = \Omega(n \log n)$ result, it is a little tedious but not difficult to derive the same (or improved as a function of n) lower bounds for all degrees. We again note that $G(d, n)$ is nonempty for $2 \leqq d \leqq n-1$ if and only if dn is even, so that we always assume that dn is even.

Aleliunas [1] has shown that $U(2, n) \leqq U(d, (d-1)n)$. We modify his construction to obtain Lemma 9.

LEMMA 9. $U(2, n) \leqq (2/d) U(d, (d-1)n)$.

Proof. We show how to derive a universal sequence s' for $G(2, n)$ from a universal sequence s for $G(d, (d-1)n)$. For any $a, b \in \{0, 1, \cdots, d-1\}$ let $s(a, b)$ be the sequence obtained from s by replacing each a by 0, each b by 1 and deleting all other symbols. Now let a, b be the least frequently occurring symbols in s and define $s' = s(a, b)$ so that $|s'| \leqq (2/d)|s|$. We will show that s' is universal for $G(2, n)$.

Let C be any labeled n-cycle. We want to construct a labeled graph G_C in $G(d, (d-1)n)$ whose traversal by s will guarantee that s' covers C. Let $K^i_{d-1} = (V^i, E^i)$ for $0 \leqq i \leqq n-1$ be n copies of the complete graph K_{d-1}. Say $V^i = \{v^i_1, \cdots, v^i_{d-1}\}$. Then $G_C = \langle \cup_i V^i, \cup_i E^i \cup D \rangle$, where

$$D = \{(v^i_j, v^{(i+1)(\mathrm{mod}\ n)}_j) | 0 \leqq i \leqq n-1, 1 \leqq j \leqq d-1\}.$$

Intuitively, the K^i_{d-1} correspond to nodes in C while the edges in D correspond to the edges in C. We label the edges in E^i by any labeling from $\{0, 1, \cdots, d-1\} \backslash \{a, b\}$. We label the edges in D in a way which corresponds exactly to the labeling in C. That is, if $\langle i, i+1 \rangle$ has label "0" (respectively, "1") in C then for all j, $\langle v^i_j, v^{i+1}_j \rangle$ has label a (respectively, b) and $\langle v^i_j, v^{i-1}_j \rangle$ has label b (respectively, a). Clearly G_C is in $G(d, (d-1)n)$. Now it should be clear that as s traverses the graph G_C, it is precisely the labels $\{a, b\}$ that cause a traversal between neighboring (in the cycle) copies of K_{d-1}. Thus s covers G_c implies s' covers C. □

LEMMA 10. $U(d_1, n) \leqq (d_1/d_2) U(d_2, (d_2 - d_1 + 1)n)$ *for all* $d_1 \leqq d_2$.

Proof. This is an immediate generalization of the construction in Lemma 9. □

LEMMA 11. $U(d, n) = \Omega(n \log n - n \log d)$.

Proof. If n was divisible by $d-1$, this lemma would follow immediately from Lemma 9 and Theorem 2. For arbitrary sufficiently large $n = q(d-1) + r$ we proceed as follows. If $r = 0$ then we would follow the construction of Lemma 9 and form a "cycle" with $q = n/(d-1)$ copies of K_{d-1}. If $r > 0$ we form $q-3$ copies of $K^i_{d-1} = (V^i, E^i)$ and a set of nodes W with $|W| = n - (q-3)(d-1) = 3(d-1) + r$. As before, each copy K^i_{d-1} will play the role of a node in a cycle as will W. We only have to describe how to fit W into a cyclic structure.

Suppose we want W to have K^1_{d-1} and K^{q-3}_{d-1} as its cyclic neighbors. Since dn is even, it follows that $d|W|$ is even and $|W| \geqq d+1$. Thus we can form a d-regular graph (W, E) and remove $d-1$ node-disjoint edges, say (u_j, w_j). We connect W to K^1_{d-1} and K^{q-3}_{d-1} by edges $\{(u_j, v^1_j)\}$ and $\{(w_j, v^{q-3}_j)\}$ for $1 \leqq j \leqq d-1$. We connect K^i_{d-1} to K^{i+1}_{d-1} $(1 \leqq i \leqq q-3)$ as in Lemma 9. In this way we are able to construct a d-regular graph G_C on n nodes. And again, as in Lemma 9, for any universal sequence s for $G(d, n)$ we construct $s' = s(a, b)$, where a and b are the least frequently occurring symbols in s. By labeling the "cycle" in G_C to mimic the labeling in C we can argue that if s is started on some node in $K^{\lceil (q-3)/2 \rceil}_{d-1}$, then s' would cover at least $\lceil (q-3)/2 \rceil$ nodes in

C. Since C is an arbitrary member of $G(2, n)$, this insures that s' covers at least $\lceil (q-3)/2 \rceil$ nodes in any infinite labeled line. Therefore $U(2, n/(3d)) \leqq U(2, (q-3)/2) \leqq (2/d)\, U(d, n)$ for any sufficiently large n which together with Theorem 2 yields Lemma 11. □

Lemma 11 shows that for small d (say $d \leqq n^{\varepsilon}$, $\varepsilon < 1$), we have $U(d, n) = \Omega(n \log n)$. For large d, there is a simple way to achieve the same bound using the driver-passenger game introduced in § 4.

LEMMA 12. $U(d, n) \geqq DP(d, n) = \Omega(n \log d)$.

Proof. The driver's strategy is simply first to form a directed cycle on n nodes with the first n labels. Then whenever the driver has a free choice he chooses to direct the new edge to the nearest nonadjacent node on the cycle. This continues until some node has degree d. On the ith tour of the cycle, the driver takes at least $\lfloor n/i \rfloor$ steps and the game continues for at least d tours. Thus,

$$DP(d, n) \geqq \sum_{i=1}^{d} \left\lfloor \frac{n}{i} \right\rfloor = \Omega(n \log d). \qquad \square$$

In terms of n, the largest known lower bound is obtained for $d = n/2 - 1$ by another simple driver-passenger game.

LEMMA 13. *For* $d \leqq n/2 - 1$, $U(d, n) \geqq DP(d, n) \geqq d(n-d)$.

Proof. The game will construct $G = (V, E)$ in $G(d, n)$ with $V = V_1 \cup V_2$, $|V_1| = n - d - 1$, $|V_2| = d + 1$. G is constructed so that a given universal sequence will visit only nodes in V_1 within the first $d(n-d-1)$ steps. This is simply achieved by thinking of the driver generating a d-regular graph on V_1 minus an edge. We complete the construction of G by choosing any complete graph on V_2 minus an edge with an arbitrary labeling. Now if (u_1, w_1) (respectively, (u_2, w_2)) is missing from V_1 (respectively, V_2), then G is the union of these graphs on V_1 and V_2 joined by the two edges (u_1, u_2) and (v_1, v_2). The driver forces the sequence (passenger) to stay on V_1 until $d(n-d-1)$ steps have expired and then trivially forces another d steps to cover the nodes of V_2. □

THEOREM 4. *For all* $d \leqq (n/2 - 1)$,

$$U(d, n) = \Omega((n \log n) + d(n-d)).$$

Proof. The proof is immediate from Lemmas 11, 12, and 13. □

Our final result emphasizes the importance of the case $d = 3$. Theorem 5 below is based on Theorem 4.13 of Cook and Rackoff [4]. Let $G'(d, n)$ denote the class of all graphs with n nodes and all degrees less than or equal to d labeled by $\{0, 1, \cdots, d-1\}$. In this case a sequence s in $\{0, 1, \cdots, d-1\}^*$ and a node u_0 in a graph G in $G'(d, n)$ uniquely defines a sequence of nodes $u_0, u_1, \cdots, u_{|s|}$ in G with (u_{i-1}, u_i) labeled s_i if u_{i-1} has an out edge so labeled and $u_i = u_{i-1}$ if u_{i-1} does not have an out edge labeled by s_i. And now, as before, let $U'(d, n)$ denote the minimal length of a universal sequence for $G'(d, n)$.

The following is a direct consequence of Theorem 4.13 in Cook and Rackoff [4].

LEMMA 14. *There is a finite state transducer computing a function*

$$f: \{0, 1, 2\}^* \to \{0, 1, \cdots, d-1\}^*$$

with the property that if s is universal for $G'(3, (2d-1)n)$ then $f(s)$ is universal for $G'(d, n)$. Furthermore, $|f(s)| \leqq (1/\log d)\, |s|$ so that

$$U'(d, n) \leqq \frac{1}{\log d}\, U'(3, (2d-1)n).$$

In order to place this result within the context of regular graphs we need to justify our introductory comment that regularity is not a significant restriction.

LEMMA 15. *$U'(d, n) \leqq U(d, d'n)$, where $d' = d$ if d is even and $d' = d+1$ if d is odd.*

Proof. For any G' in $G'(d, n)$ we construct a G in $G(d, d'n)$ by taking d' copies of G'. If x has degree $\delta < d$ in G' we connect all d' copies of x by a graph in $G(d-\delta, d')$ (a perfect matching if $\delta = d-1$). The new edges are labeled by the labels missing at x in G'. It is easily verified that a sequence that covers G covers G' as well. If all the degrees in G' are d and $d-1$ a slight modification is needed, which we omit. □

THEOREM 5.

$$U(d, n) \leqq \frac{1}{\log d} U(3, 2(2d-1)n).$$

Furthermore, there is a finite state transducer computing a function

$$f: \{0, 1, 2\}^* \to \{0, 1, \cdots, d-1\}^*$$

such that if s is universal for $G(3, 2(2d-1)n)$ then $f(s)$ is universal for $G(d, n)$.

Proof. The Cook and Rackoff construction on which Lemma 14 is based produces graphs where every node has degree two or three. In this case, we can take $d' = d-1 = 2$ in the construction of Lemma 15. The theorem then follows immediately from Lemmas 14 and 15. □

6. Conclusion. Perhaps the main technical result of this paper is the proof of a nonlinear lower bound for $d = 2$, thus answering a specific challenge in Aleliunas et. al. [2]. However it is clear that even for this restricted case we are far from understanding the true nature of $U(2, n)$. We know that the crucial aspect of labeled lines is the parity of the blocks. (We claim that, within a factor of n, we can assume that every block has length 1 or 2.) It seems feasible to us that some of the ideas developed here will lead to an explicit polynomial length universal sequence for $G(2, n)$. We also expect to be able to narrow the gap between the lower and upper bounds for $U(2, n)$.

For the complete graph, many obvious questions remain. It seems reasonable to be able to explicitly construct a "good" universal sequence. At present, we only know the brute force approach that gives $n^{n^2} \approx |G(n-1, n)|$. It is not difficult to see that a sequence universal for $G(n-1, n)$ will traverse at least n nodes when applied to members of $G(n, n+1)$. But we cannot see how to use this fact to construct such sequences. It also seems reasonable that we can narrow the gap for $\mathrm{DP}(n-1, n)$.

Theorem 5 emphasizes the importance of $U(3, n)$. In particular, any explicit universal sequence beyond brute force for $G(3, n)$ would be of interest. It will also be of interest if we could find for $d \geqq 3$ a simple d-ary infinite graph that would play the role that the infinite line played for $d = 2$.

Finally, there are many alternative universal sequence formulations that could be used for determining graph connectivity. One formulation we find particularly interesting is to number the nodes $V = \{1, \cdots, n\}$ and consider sequences in $\{1, \cdots, n\}^*$. Now, a sequence command i causes a move to node i if there is an edge from the currently scanned node to node i. Otherwise, it remains in the current node. Random walk arguments again show the existence of polynomial length universal sequences.

Note added in proof. Bridgland [*J. Algorithms*, 8 (1987), pp. 395-404] has given another construction for a universal sequence of length $n^{O(\log n)}$. His construction differs from ours. We were informed also of a construction by Barrington [private communication] for the same problem. Since the completion of this research in June

1986 there has been much activity in this area, and some of our results have been improved. Istrail [*Proc. 20th Annual ACM Symposium on Theory of Computing*, 1988, pp. 491-503] presents an explicit construction of a sequence of polynomial length that is universal for $d = 2$. The ideas required go beyond the ones presented here. Karloff, Paturi, and Simon [unpublished manuscript] have given an explicit construction of a sequence universal for complete graphs whose length is $n^{O(\log n)}$. The construction of a polynomial length sequence even for complete graphs remains open. Alon and Ravid [*Discrete Appl. Math.*, to appear] have improved our lower bound for $U(n-1, n)$ to $n^2/\log n$. Their bound does not apply to our driver-passenger game. Also in the closely related area of random walks on graphs there has been considerable progress. A special issue of the *Journal of Theoretical Probability*, D. Aldous, ed., will be dedicated to the subject. A paper by Kahn, Linial, Nisan, and Saks that will appear therein shows that the expected cover time for regular graphs is only $O(n^2)$. This yields an improvement on the upper bound for $U(d, n)$ over the one in [2].

REFERENCES

[1] R. ALELIUNAS, *A simple graph traversing problem*, M.Sc. thesis, University of Toronto, Toronto, Ontario, Canada, January 1978.

[2] R. ALELIUNAS, R. M. KARP, R. J. LIPTON, L. LOVÁSZ, AND C. RACKOFF, *Random walks, universal traversal sequences, and the complexity of maze problems*, in Proc. 20th Annual Symposium on Foundations of Computer Science, 1979, pp. 218-223.

[3] M. F. BRIDGLAND, *Universal traversal sequences for paths and cycles*, J. Algorithms, 8 (1987), pp. 395-404.

[4] S. COOK AND C. RACKOFF, *Space lower bounds by maze threadability on restricted machines*, SIAM J. Comput., 9 (1980), pp. 636-652.

[5] L. LOVÁSZ, *Combinatorial Problems and Exercises*, North-Holland, Amsterdam, 1979. p. 249 (solution to exercise 5.2).

[6] H. R. LEWIS AND C. H. PAPADIMITRIOU, *Symmetric space bounded computation*, Theoret. Comput. Sci., 9 (1982), pp. 161-187.

[7] W. J. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177-192.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 278-287, April 1989

007

WORST-CASE GROWTH RATES OF SOME CLASSICAL PROBLEMS OF COMBINATORIAL OPTIMIZATION*

J. MICHAEL STEELE† AND TIMOTHY LAW SNYDER‡

Abstract. A method is presented for determining the asymptotic worst-case behavior of quantities like the length of the minimal spanning tree or the length of an optimal traveling salesman tour of n points in the unit d-cube. In each of these classical problems, the worst-case lengths are proved to have the exact asymptotic growth rate of $\beta_n^{(d-1)/d}$ as $n \to \infty$, where β is a positive constant depending on the problem and the dimension. These results complement known results on the growth rates for the analogous quantities under probabilistic assumptions on the points, but the results given here are free of any probabilistic hypotheses.

Key words. asymptotics, traveling salesman problem, minimal spanning tree, Beardwood-Halton-Hammersley theorem

AMS(MOS) subject classifications. 05C35, 90B10, 90C10, 52A40

1. Introduction.

The purpose of this paper is to illustrate a general method for determining the asymptotic behavior of some classical quantities of operations research and combinatorial optimization. For specificity, we focus on the traveling salesman problem and on the minimal spanning tree of n points in the unit d-cube, but the general applicability of our method to a number of other problems will be made evident.

To set our problem precisely, we first note that a Euclidean minimal spanning tree or a traveling salesman tour can be represented by a graph $G = (V_n, E)$, where V_n denotes a set of n points in $[0, 1]^d$, where $d \geqq 2$, and E denotes a subset of the edges of the complete graph on the points of V_n. The length of an edge $e = \{x_i, x_j\}$ is taken to be the usual Euclidean distance, and we write $|e| = |x_i - x_j|$ for that length. For a collection E of edges we will often use $L(E)$ to denote the sum of the lengths of the edges in E, i.e., we define $L(E) = \sum_{e \in E} |e|$. Still, when V is a finite set there will be no ambiguity in using $|V|$ to denote the cardinality of V.

The objects of principal interest here are the sequences $\rho_{\mathrm{MST}}(n)$ and $\rho_{\mathrm{TSP}}(n)$, defined by

$$\rho_{\mathrm{MST}}(n) = \max_{\substack{V_n \subset [0,1]^d \\ |V_n| = n}} \left\{ \min_T \sum_{e \in T} |e| \colon T \text{ is a spanning tree of } V_n \right\}$$

and

$$\rho_{\mathrm{TSP}}(n) = \max_{\substack{V_n \subset [0,1]^d \\ |V_n| = n}} \left\{ \min_T \sum_{e \in T} |e| \colon T \text{ is a tour of } V_n \right\}.$$

In other words, $\rho_{\mathrm{MST}}(n)$ is equal to the largest possible length of any minimal spanning tree formed from n points in $[0, 1]^d$. Similarly, $\rho_{\mathrm{TSP}}(n)$ is the largest possible length of any optimal traveling salesman tour through n points in $[0, 1]^d$. The use of max instead of sup in the definitions of $\rho_{\mathrm{MST}}(n)$ and $\rho_{\mathrm{TSP}}(n)$ is justified by the fact that the expressions in braces can be viewed as continuous functions on the compact set obtained by forming the product of n copies of $[0, 1]^d$, i.e., $\prod_{1 \leqq i \leqq n} [0, 1]^d$.

* Received by the editors November 30, 1987; accepted for publication June 17, 1988. This research was supported in part by National Science Foundation grant DMS-8414069.

† Program in Statistics and Operations Research, Princeton University, Princeton, New Jersey 08544.

‡ Department of Computer Science, Georgetown University, Washington, DC 20057.

One should note that the functions ρ_{MST} and ρ_{TSP} depend on the dimension d. This fact also applies to all of the other functions and constants that are used here. Since $d \geqq 2$ is fixed, we will suppress the dependence of ρ_{MST}, ρ_{TSP}, and other functions on d, but the reader should be mindful of this dependence, especially in the main result.

THEOREM. *There are constants β_{MST} and β_{TSP} depending on the dimension $d \geqq 2$ such that*

$$\rho_{MST}(n) \sim \beta_{MST} n^{(d-1)/d} \tag{1.1}$$

and

$$\rho_{TSP}(n) \sim \beta_{TSP} n^{(d-1)/d} \tag{1.2}$$

as $n \to \infty$, with $\beta_{TSP} \geqq \beta_{MST} \geqq 1$.

This result provides the determination of the exact asymptotic order of the functions ρ_{MST} and ρ_{TSP} in any dimension $d \geqq 2$. Considerable earlier effort focused on bounds for $\rho_{MST}(n)$ and $\rho_{TSP}(n)$, but none of the inequalities provided by that work is tight enough to determine that $\rho_{MST}(n)$ or $\rho_{TSP}(n)$ are actually asymptotic to a constant times $n^{(d-1)/d}$. Some earlier results of particular interest are the bound of Verblunsky (1951), which says that in $d = 2$ one has $\rho_{TSP}(n) \leqq (2.8n)^{1/2} + 3.15$, and the bounds of Fejes-Tóth (1940), which say that $\rho_{TSP}(n)$ and $\rho_{MST}(n)$ are both at least as large as $(1-\varepsilon)(4/3)^{1/4} n^{1/2}$ for all $n \geqq N(\varepsilon)$. Few (1955) improved the upper bound of Verblunsky (1951) to $\rho_{TSP}(n) \leqq (2n)^{1/2} + 1.75$ in $d = 2$ and obtained $\rho_{TSP}(n) \leqq d\{2(d-1)\}^{(1-d)/2d} n^{(d-1)/d} + 0(n^{1-2/d})$ for general $d \geqq 2$.

Recent results have improved these bounds. Few's bound on $\rho_{TSP}(n)$ in dimension two is sharpened in Supowit, Reingold, and Plaisted (1983), to show that $\rho_{TSP}(n) \geqq (4/3)^{1/4} n^{1/2}$, for all $n \geqq 1$. Moran (1984) used inequalities on sphere packing to obtain essential improvements on the upper bounds of Few for large values of d. Goldstein and Reingold (1988) carefully analyze Few's heuristic algorithm to improve the upper bounds in dimensions $3 \leqq d \leqq 7$. They also improve lower bounds, using the exact densities of sphere packings for $2 \leqq d \leqq 8$. Goldstein (personal communication) has further improved the upper bounds in dimensions three and four.

The $(2n)^{1/2}$ barrier on $\rho_{TSP}(n)$ in dimension two is broken by bounds of Karloff (1987) that show $\rho_{TSP}(n) < 0.984(2n)^{1/2} + 11$. Also, for low dimensions $d \geqq 3$, Goddyn (1988) improves all known upper bounds on $\rho_{TSP}(n)$ by considering an infinite number of translations of quantizers other than cubical cylinders.

Some other early work focused on the probabilistic circumstances under which one can provide bounds for the lengths of the minimal spanning tree or optimal traveling salesman tour. For example, Ghosh (1949) sharpened earlier results of Mahalanobis (1940) and Jessen (1942) to establish that the expected length of an optimal traveling salesman tour of n points chosen at random from the unit square was at most $1.27n^{1/2} + 0(1)$. The bound of Marks (1948) complements the upper bound of Ghosh (1949) by providing a lower bound of $(n^{1/2} - 1/n^{1/2})/2$ on the expected length of an optimal traveling salesman tour in $d = 2$.

The culminating result on the length of an optimal traveling salesman tour under probabilistic assumptions was provided by Beardwood, Halton, and Hammersley (1959). That work showed that if T_n denotes the length of an optimal traveling salesman tour of X_i, where $1 \leqq i \leqq n$ and the X_i are bounded independent identically distributed random vectors in $\mathbb{R}^d$, then with probability one we have the asymptotic relation

$$T_n \sim c_d n^{(d-1)/d} \int_{\mathbb{R}^d} f(x)^{(d-1)/d}\, dx. \tag{1.3}$$

Here, f denotes the density of the absolutely continuous part of the distribution of the X_i, and c_d is a constant depending only on the dimension.

In addition to providing improved upper and lower bounds on the constant c_d, Beardwood, Halton, and Hammersley (1959) also indicated that a result analogous to (1.3) holds for the minimal spanning tree. A review of the probability theory which has grown out of the Beardwood, Halton, and Hammersley theorem is given in Steele (1987), and a review oriented toward algorithmic applications is given in Karp and Steele (1985).

The focus of the present work is on the growth rates of the *worst-case* lengths of the traveling salesman tour and minimal spanning tree. There are no probabilistic assumptions used here, and it is perhaps remarkable that one obtains asymptotics that are so close in form to the probabilistic results. Another intriguing aspect of these limit theorems is that the same method applies both to a computationally difficult problem (the TSP) and to one which is computationally easy (the MST).

The proof of the main theorem is given in three sections. The first of these sections provides a general lemma that isolates inequalities that are sufficient to determine the asymptotic behavior of ρ_{MST} and ρ_{TSP}. The following section focuses on minimal spanning trees, and, in particular, it provides an approximate recursion relation for ρ_{MST}. The construction used to study ρ_{TSP} in § 4 is much like that used for ρ_{MST}; so the analysis required for the optimal traveling salesman tour is quite brief.

The final section points out some limitations of this method and comments on some open problems.

2. Asymptotics from an approximate recursion. One principle underlying our asymptotic analysis is that both $\rho_{\mathrm{MST}}(n)$ and $\rho_{\mathrm{TSP}}(n)$ satisfy inequalities which bound their rates of growth and express an approximate recursiveness. The following lemma shows that a slow incremental rate of growth (as expressed by (2.1(i))) and an approximate recursiveness (as expressed by (2.1(ii))) are together sufficient to determine the exact asymptotic behavior of a sequence. Even though the lemma appears technical, we will later see that the two required conditions are quite natural to the objects under study.

LEMMA 2.1. *If* $\rho(1)=0$ *and there is a constant* $c_1 \geqq 0$ *such that for all* $m \geqq 1$ *and* $k \geqq 1$

$$\text{(i)}\quad \rho(n+1) \leqq \rho(n) + c_1 n^{-1/d}$$

(2.1) *and*

$$\text{(ii)}\quad m^{d-1}\rho(k) - m^{d-1}k^{(d-1)/d}r(k) \leqq \rho(m^d k),$$

where $r(k) \to 0$ *as* $k \to \infty$, *then as* $n \to \infty$

$$\rho(n) \sim \beta n^{(d-1)/d}$$

for a constant β.

Proof. From the hypothesis (2.1(i)) and the fact that $\rho(1)=0$ we first note that for $1 \leqq i < j < \infty$ we have

$$\begin{aligned} \rho(j) - \rho(i) &= \sum_{k=i}^{j-1} \{\rho(k+1) - \rho(k)\} \\ &\leqq c_1 \int_{i-1}^{j-1} x^{-1/d}\, dx \leqq 5c_1(j^{(d-1)/d} - i^{(d-1)/d}). \end{aligned} \tag{2.2}$$

Letting $i=1$ and $j=n$ in (2.2) shows that $\rho(n) \leqq 5c_1 n^{(d-1)/d}$, so if we define $\psi(k) = \rho(k)/k^{(d-1)/d}$, then we see that $\psi(k) < 5c_1$ for all k. We can then introduce a candidate

for our limit by

$$\gamma = \limsup_{k\to\infty} \psi(k) < \infty. \tag{2.3}$$

Inequality (2.1(ii)) tells us that for all k and m,

$$\psi(k) - r(k) \leqq \psi(m^d k), \tag{2.4}$$

so, given any fixed $\varepsilon > 0$, we can choose a k_ε such that $\gamma - \varepsilon \leqq \psi(k_\varepsilon)$ and $r(k_\varepsilon) \leqq \varepsilon$, thus obtaining from (2.4) that

$$\gamma - 2\varepsilon \leqq \psi(m^d k_\varepsilon) \tag{2.5}$$

for all $m \geqq 1$.

Next define $j_m = m^d k_\varepsilon$ and consider n such that $j_m \leqq n \leqq j_{m+1}$. To bound the absolute difference $|\psi(n) - \psi(j_m)|$, we use (2.2):

$$\begin{aligned} \sup_{j_m \leqq n \leqq j_{m+1}} |\rho(n) - \rho(j_m)| &\leqq 5c_1 (j_{m+1}^{(d-1)/d} - j_m^{(d-1)/d}) \\ &= 5c_1 k_\varepsilon^{(d-1)/d} m^{d-1} [(1+1/m)^{d-1} - 1], \end{aligned} \tag{2.6}$$

or, in terms of ψ, the binomial expansion gives

$$\sup_{j_m \leqq n \leqq j_{m+1}} |\psi(n) - \psi(j_m)| \leqq 5c_1 \{(1+1/m)^{d-1} - 1\} < 5c_1 m^{-1} 2^{d-1}. \tag{2.7}$$

From (2.7) and (2.5) we find for $j_m \leqq n \leqq j_{m+1}$ that

$$\gamma - 2\varepsilon - 5c_1 m^{-1} 2^{d-1} \leqq \psi(n),$$

and, hence, $\gamma - 2\varepsilon \leqq \liminf_{n\to\infty} \psi(n)$. By the arbitrariness of $\varepsilon > 0$, we have proved

$$\limsup_{n\to\infty} \psi(n) \leqq \liminf_{n\to\infty} \psi(n)$$

and the lemma is complete.

3. Minimal spanning trees. We will now show that ρ_{MST} satisfies the hypotheses of the preceding lemma. The key issue is the derivation of an inequality like (2.1(ii)). This will be done by a recursive construction of a point set for which a minimal spanning tree has near maximal length.

We first divide the d-cube $Q = [0,1]^d$ into m^d *cells* Q_i, where $1 \leqq i \leqq m^d$ and each cell has side length $1/m$. The boundaries ∂Q_i of the cells Q_i create a natural *grating* in the unit d-cube which we denote by H, i.e., we set $H = \bigcup_{i=1}^{m^d} \partial Q_i$. For $0 < \alpha < 1/m$, let H^α denote the set of points of $[0,1]^d$ which are within $\alpha/2$ of H, thus H^α is the grating H fattened to a width of α. Similarly, we define *subcells* Q_i^α of Q_i by $Q_i^\alpha = Q_i - H^\alpha$.

Inside each of the Q_i^α we now place a set S_i of k points for which the length of the minimal spanning tree is $(m^{-1} - \alpha)\rho_{\mathrm{MST}}(k)$, i.e., inside each subcell we place a copy of a set of k points that attains the worst-case bound on the length of a minimal spanning tree of k points. The factor of $(m^{-1} - \alpha)$ equals the side length of Q_i^α, and it reflects the scaling of $\rho_{\mathrm{MST}}(k)$ down to the smaller cube. Next, we let T be a minimal spanning tree of the set of $m^d k$ points $\bigcup_{i=1}^{m^d} S_i$, and we let T_i denote a minimal spanning tree of S_i. We will now develop a relationship between $L(T)$ and $L(\bigcup_{i=1}^{m^d} T_i)$ that moves us toward an inequality like (2.1(ii)) for ρ_{MST}.

First consider the forest that is obtained from T by deleting from T all the edges that have length as great as α. We let $\lambda(\alpha)$ denote the number of edges deleted from T, i.e., we set

$$\lambda(\alpha) = |\{e \in T: |e| \geqq \alpha\}|.$$

Since T was connected, the graph that remains following the deletion of $\lambda(\alpha)$ edges has at most $\lambda(\alpha)+1$ connected components. Moreover, each of these connected components is contained entirely in some subcell Q_i^{α}.

Next, if two or more connected components of T coexist in the same subcell Q_i^{α}, then we join them together to make a tree on the point set S_i. Since, within any given cell, we can rejoin any two components at a cost not exceeding $d^{1/2}m^{-1}$, the total cost of rejoining all the within-cell components is bounded by $d^{1/2}m^{-1}\lambda(\alpha)$.

So far we have constructed a spanning tree for each S_i, where $1 \leqq i \leqq m^d$. Since the length of each of these trees must be at least as great as the length of the minimal spanning tree T_i of the point set S_i, we have the bound

$$\sum_{i=1}^{m^d} L(T_i) \leqq L(T) + d^{1/2}m^{-1}\lambda(\alpha).$$

But we know $L(T_i) = (m^{-1} - \alpha)\rho_{\mathrm{MST}}(k)$ and $L(T) \leqq \rho_{\mathrm{MST}}(m^d k)$, so we can rewrite this bound to provide

$$m^d(m^{-1} - \alpha)\rho_{\mathrm{MST}}(k) - d^{1/2}m^{-1}\lambda(\alpha) \leqq \rho_{\mathrm{MST}}(m^d k). \tag{3.1}$$

In order to extract an equality like (2.1(ii)) for ρ_{MST} from (3.1), we need some elementary facts about sets of points in $[0,1]^d$ and their associated minimal spanning trees. We begin by recalling an easy pigeonhole argument, which says that from any set of n points in $[0,1]^d$, one can always find a pair that are close together.

LEMMA 3.1. *There exists a constant c_2 such that for any $\{x_1, x_2, \cdots, x_n\} \subset [0,1]^d$, where $n \geqq 2$, one has*

$$|x_i - x_j| \leqq c_2 n^{-1/d}$$

for some x_i and x_j, $1 \leqq i < j \leqq n$.

Proof. Cover each x_i with a ball of radius r centered at x_i, and note that such a ball has volume $\omega_d r^d$, where ω_d is the volume of the unit d-ball. If all of the balls constructed in this way were non-intersecting, then each would cover at least $2^{-d-1}\omega_d r^d$ of $[0,1]^d$, even if we generously assume that each of the balls were centered exactly in a corner of the hypercube. In total, the n balls would cover at least a volume of $2^{-d-1}\omega_d r^d n$, and since $[0,1]^d$ has unit volume, we have $2^{-d-1}\omega_d r^d n \leqq 1$. The lemma is therefore established with $c_2 = 2^{(2d+1)/d}\omega_d^{-1/d}$.

One can easily improve the constant c_2, but this simply derived constant is sufficient for our purposes. It is now easy to give a bound on ρ_{MST} that shows the validity of the first hypothesis of Lemma 2.1.

LEMMA 3.2. *There exists a constant c_3 such that for all $n \geqq 1$, one has the bound*

$$\rho_{\mathrm{MST}}(n+1) \leqq \rho_{\mathrm{MST}}(n) + c_3 n^{-1/d}. \tag{3.2}$$

Proof. Let $S = \{x_1, x_2, \cdots, x_{n+1}\}$ denote a set of $n+1$ points in $[0,1]^d$ for which the length of a minimal spanning tree is $\rho_{\mathrm{MST}}(n+1)$. By Lemma 3.1, there exist x_i and x_j in S such that $|x_i - x_j| \leqq c_2(n+1)^{-1/d} \leqq c_2 n^{-1/d}$. We form a minimal spanning tree T of $\{x_1, x_2, \cdots, x_{i-1}, x_{i+1}, \cdots, x_{n+1}\}$ and then augment the tree by adding to it the edge

$\{x_i, x_j\}$. This construction provides a spanning tree of S at a cost of no more than $L(T)+c_2 n^{-1/d}$. Therefore, we have

$$\rho_{\mathrm{MST}}(n+1) \leqq L(T)+c_2 n^{-1/d} \leqq \rho_{\mathrm{MST}}(n)+c_2 n^{-1/d},$$

which proves our lemma with $c_3 = c_2$.

Naturally we can sum inequality (3.2) to provide a bound on $\rho_{\mathrm{MST}}(n)$.

COROLLARY. *There is a constant c_4 such that for all $n \geqq 1$, one has the bound*

$$\rho_{\mathrm{MST}}(n) \leqq c_4 n^{(d-1)/d}. \tag{3.3}$$

Here we note that $c_4 = 2c_3$ is a sufficient choice for the constant c_4.

The next lemma provides a tool for understanding how a minimal spanning tree changes as edges are added or deleted. While the result is reasonably intuitive and can be established by modification of Kruskal's algorithm (see, e.g., Aho, Hopcroft, and Ullman (1974)), the rigorous justification of the modified Kruskal algorithm does not seem to be as easy as the characterization-based proof used here.

LEMMA 3.3. *Let E be a subset of a minimal spanning tree of $S = \{x_1, x_2, \cdots, x_n\} \subset [0,1]^d$, and let S' be the set of points incident with the edges of E. Then, there exists a minimal spanning tree of S' that contains E.*

Proof. The graph corresponding to the set E consists of k connected components $(S_1, T_1), (S_2, T_2), \cdots, (S_k, T_k)$, where $1 \leqq k \leqq |E|$. We first show that for all $1 \leqq i \leqq k$, T_i is a minimal spanning tree of S_i. To see this, consider a minimal spanning tree T. If we form a forest of two trees by removing an edge from T, then it is trivial to note that each resulting tree is a minimal spanning tree of the respective set of points incident with it. Now let T be a minimal spanning tree of S, and recursively apply this fact by removing from the tree T all the edges of $T-E$. As each edge $e \in T-E$ is removed, the minimal spanning tree to which e belongs becomes two minimal spanning trees. After removing all the edges of $T-E$, the result is the edge set E, which is a forest of minimal spanning trees.

We first recall a well-known fundamental property of minimal spanning trees. If $\{(V_1, E_1), (V_2, E_2), \cdots, (V_k, E_k)\}$, where $k > 1$, is a forest spanning the point set S, and $e = \{x_i, x_j\}$ is an edge of minimum length such that e has exactly one endpoint in V_1, then there exists a tree T^* spanning S and including $\bigcup_{i=1}^{k} E_i \cup \{e\}$ such that $L(T^*) = \min\{L(T)\colon T \text{ is a tree spanning } S \text{ and } \bigcup_{i=1}^{k} E_i \subset T\}$. We use this easily proved fact (see Aho, Hopcroft, and Ullman (1974), or Papadimitriou and Steiglitz (1982)) to construct from the edge set E a minimal spanning tree of S'.

Begin with the edges of E, which constitute a forest of minimal spanning trees, and iteratively add to T_1 an edge of minimal length over all those edges having exactly one endpoint in S_1. Merging components this way, we obtain a tree T that spans S'. Moreover, T is a minimum-cost tree over all trees that span S' and contain E. Hence, the only way we could lessen the cost of T would be lessen the cost of a tree T_i, where $1 \leqq i \leqq k$. But, since T_i is a minimal spanning tree, this is impossible, and we conclude that T is a minimal spanning tree of S'. Since T contains E, the proof is complete.

We now use Lemma 3.3 and the corollary to Lemma 3.2 to bound the total length of any k edges of a minimal spanning tree.

LEMMA 3.4. *There is a constant c_5 such that if E is any subset of the edges of a minimal spanning tree of $\{x_1, x_2, \cdots, x_n\} \subset [0,1]^d$, then*

$$L(E) \leqq c_5 |E|^{(d-1)/d}.$$

Proof. Let S be the set of endpoints of the edges of E and note $|S| \leqq 2|E|$. By Lemma 3.3, there exists a minimal spanning tree of S that contains E. By inequality

(3.3) we have therefore that

$$\sum_{e \in E} |e| \leqq c_4 |S|^{(d-1)/d},$$

so the lemma is proved with $c_5 = 2^{(d-1)/d} c_4$.

We require one more general inequality in order to bound $\lambda(\alpha)$ in our key relation (3.1). Formally, we let $\nu_{\mathrm{MST}}(x)$ denote the maximal value k such that there exists a minimal spanning tree of some $V_n = \{x_1, x_2, \cdots, x_n\} \subset [0,1]^d$ with k edges greater than or equal to x in length.

LEMMA 3.5. *There is a constant c_6 such that for all $x > 0$, one has*

$$\nu_{\mathrm{MST}}(x) \leqq c_6 x^{-d}. \tag{3.4}$$

Proof. Let T be a minimal spanning tree of $\{x_1, x_2, \cdots, x_n\}$, and set $\phi_T(x) = |\{e \in T: |e| \geqq x\}|$. By Lemma 3.4 any set of $\phi_T(x)$ edges of T must have length bounded by $c_5 \phi_T(x)^{(d-1)/d}$, so

$$x\phi_T(x) \leqq \sum_{\substack{|e| \geqq x \\ e \in T}} |e| \leqq c_5 \phi_T(x)^{(d-1)/d}. \tag{3.5}$$

Clearing ϕ_T to the left side gives

$$\phi_T(x) \leqq c_5^d x^{-d},$$

and, since this bound holds for all minimal spanning trees T, the lemma is proved with $c_6 = c_5^d$.

Returning to our basic recurrence relation (3.1), we can write it in a form closer to the hypothesis of Lemma 2.1 as follows:

$$m^{d-1}\rho_{\mathrm{MST}}(k) - \{m^d \alpha \rho_{\mathrm{MST}}(k) + d^{1/2} m^{-1} \lambda(\alpha)\} \leqq \rho_{\mathrm{MST}}(m^d k). \tag{3.6}$$

By Lemma 3.2 and its corollary $\rho_{\mathrm{MST}}(k) \leqq c_4 k^{(d-1)/d}$, and by Lemma 3.5 $\lambda(\alpha) \leqq c_6 \alpha^{-d}$, so the bracketed expression of inequality (3.6) is majorized by

$$c_4 \alpha m^d k^{(d-1)/d} + c_6 d^{1/2} m^{-1} \alpha^{-d}.$$

This quantity is approximately minimized by choosing $\alpha = m^{-1} k^{(1-d)/d(d+1)}$, and making that choice proves that there is a constant c_7 such that the inequality

$$m^{d-1}\rho_{\mathrm{MST}}(k) - c_7 m^{d-1} k^{(d-1)/(d+1)} \leqq \rho_{\mathrm{MST}}(m^d k) \tag{3.7}$$

holds for all $m \geqq 1$ and $k \geqq 1$. This last inequality shows that the main hypothesis of Lemma 2.1 is valid with $r(k) = c_7 k^{(1-d)/d(d+1)}$. Since we already know that $\rho_{\mathrm{MST}}(n+1) \leqq \rho_{\mathrm{MST}}(n) + c_3 n^{-1/d}$, we have verified all of the hypotheses of Lemma 2.1. We have therefore proved that $\rho_{\mathrm{MST}}(n) \sim \beta_{\mathrm{MST}} n^{(d-1)/d}$ as $n \to \infty$ for all $d \geqq 2$.

To see that $\beta_{\mathrm{MST}} \geqq 1$, we just note that one can place n points in the unit d-cube in such a way that no two are closer together than $n^{-1/d}$. This proves that $\beta_{\mathrm{MST}} \geqq 1$, since any connected tree has $n-1$ edges.

4. The traveling salesman problem. Just as in the treatment of minimal spanning trees, the central task is to prove the validity of (2.1(ii)). For the traveling salesman problem the task actually turns out to be easier than it was for minimal spanning trees.

As before, we partition $[0,1]^d$ into m^d cells Q_i of edge length m^{-1}. We then obtain a fattened grating H^α of width α, where $0 < \alpha < m^{-1}$, and define corresponding subcells Q_i^α with edge length $m^{-1} - \alpha$. Into each subcell Q_i^α we insert a set S_i of k points having an optimal traveling salesman tour with length $\rho_{\mathrm{TSP}}(k)(m^{-1} - \alpha)$, i.e., the set S_i attains the maximal length of any set of k points in a cube of edge length $m^{-1} - \alpha$.

Now, for each $1 \leqq i \leqq m^d$, we let T_i denote an optimal traveling salesman tour of S_i, and we further let T be an optimal traveling salesman tour of the $m^d k$ points of $\bigcup_{i=1}^{m^d} S_i$. We need to establish a relationship between the total lengths of the two sets of edges T and $\bigcup_{i=1}^{m^d} T_i$.

To build a heuristic tour T_i' through S_i, we start by taking the set T_i' to be E_i, the set of all of the edges of T that are completely contained in Q_i^α. If this set of edges forms a graph $G_i = (S_i, E_i)$ with k_i connected components, then there is a set C_i of at least k_i vertices that are in different components of G_i and have degree one or zero. The case of degree zero occurs exactly for those components consisting of a single vertex.

Since C_i has cardinality at least k_i, we can apply Lemma 3.1 to find a pair of vertices in C_i that are separated by a distance of at most $c_2 k_i^{-1/d}(m^{-1} - \alpha)$. We now add the edge determined by this pair of vertices to T_i'. Repeating this construction, we can add a total of $k_i - 1$ edges to E_i and obtain a path T_i' through all of the vertices in S_i. The ends of this path can now be joined by one final edge in order to complete the heuristic tour T_i'.

This process shows that the length of T_i' is bounded by

$$L(T_i') \leqq L(E_i) + c_2 \sum_{j=1}^{k_i} j^{-1/d} m^{-1}. \tag{4.1}$$

Now, since $\rho_{\mathrm{TSP}}(k)(m^{-1} - \alpha) \leqq L(T_i')$ and $\sum_{i=1}^{m^d} L(E_i) \leqq L(T) \leqq \rho_{\mathrm{TSP}}(m^d k)$, we can sum (4.1) over $1 \leqq i \leqq m^d$ and obtain

$$m^d \rho_{\mathrm{TSP}}(k)(m^{-1} - \alpha) \leqq \rho_{\mathrm{TSP}}(m^d k) + c_2 m^{-1} \sum_{i=1}^{m^d} \sum_{j=1}^{k_i} j^{-1/d}. \tag{4.2}$$

Next, let $\lambda(\alpha)$ denote the number of edges of T that intersect H^α. The number of connected components of $\bigcup_{i=1}^{m^d} G_i$ equals $\sum_{i=1}^{m^d} k_i \leqq \lambda(\alpha)$; so, estimating the inner sum of (4.2) by $\sum_{j=1}^{k_i} j^{-1/d} \leqq 1 + \int_1^{k_i} x^{-1/d}\, dx \leqq 2k_i^{(d-1)/d}$ and then applying Hölders inequality, we have

$$\begin{aligned} m^d \rho_{\mathrm{TSP}}(k)(m^{-1} - \alpha) &\leqq \rho_{\mathrm{TSP}}(m^d k) + 2c_2 m^{-1} \sum_{i=1}^{m^d} k_i^{(d-1)/d} \\ &\leqq \rho_{\mathrm{TSP}}(m^d k) + 2c_2 m^{-1} \left(\sum_{i=1}^{m^d} 1 \right)^{1/d} \left(\sum_{i=1}^{m^d} k_i \right)^{(d-1)/d} \\ &\leqq \rho_{\mathrm{TSP}}(m^d k) + 2c_2 \lambda(\alpha)^{(d-1)/d}. \end{aligned} \tag{4.3}$$

This inequality will now be put in the form needed to verify (2.1(ii)). The only real issue which remains is that of bounding $\lambda(\alpha)$, but some intermediate facts are required. First, we note that we can show

$$\rho_{\mathrm{TSP}}(n+1) \leqq \rho_{\mathrm{TSP}}(n) + c_8 n^{-1/d} \tag{4.4}$$

by taking $n+1$ points S such that $\rho_{\mathrm{TSP}}(n+1)$ is the length of the shortest tour through S and then using Lemma 3.1 to exhibit a heuristic tour through S with cost bounded by $\rho_{\mathrm{TSP}}(n) + 2c_2 n^{-1/d}$, so we have inequality (4.4) with $c_8 = 2c_2$. (For examples of this type of argument, where more attention is given to obtaining good values for the associated constants, one can consult Moran (1984). For an easier, but less quantitative, version one can consult Few (1955).)

One immediate consequence of (4.4) is that by summing over $1 \leqq i \leqq n$, we have

$$\rho_{\mathrm{TSP}}(n) \leqq c_9 n^{(d-1)/d}, \tag{4.5}$$

where $c_9 = 2c_8$.

Now, for an edge of T to intersect H^{α}, it must have endpoints in two different Q_i^{α} and therefore have length at least α. This gives the bound

$$\alpha\lambda(\alpha) \leqq \sum_{\substack{|e| \geqq \alpha \\ e \in T}} |e| \leqq \rho_{\mathrm{TSP}}(m^d k) \leqq c_9 m^{d-1} k^{(d-1)/d}. \tag{4.6}$$

When we apply (4.5) and (4.6) to (4.3) we have

$$m^{d-1}\rho_{\mathrm{TSP}}(k) \leqq \rho_{\mathrm{TSP}}(m^d k) + \alpha m^d c_9 k^{(d-1)/d} + c_{10}\alpha^{-(d-1)/d} m^{(d-1)^2/d} k^{(d-1)^2/d^2}, \tag{4.7}$$

where $c_{10} = 2c_2 c_9^{(d-1)/d}$. If we now choose $\alpha = m^{-1}k^{(1-d)/d(2d-1)} < m^{-1}$, we see that (4.7) simplifies to give

$$m^{d-1}\rho_{\mathrm{TSP}}(k) \leqq \rho_{\mathrm{TSP}}(m^d k) + c_{11} m^{d-1} k^{2(d-1)^2/d(2d-1)}, \tag{4.8}$$

for a constant c_{11}.

From inequality (4.8) we see that the main hypothesis of Lemma 2.1 is justified with $r(k) = c_{11}k^{(1-d)/d(2d-1)}$. Since (4.4) verifies the first hypothesis of Lemma 2.1, we have completed the proof that $\rho_{\mathrm{TSP}}(n) \sim \beta_{\mathrm{TSP}} n^{(d-1)/d}$ as $n \to \infty$. Naturally, since the minimal spanning tree problem is a relaxation of the traveling salesman problem, we have $\beta_{\mathrm{MST}} \leqq \beta_{\mathrm{TSP}}$.

5. Summary and concluding remarks. The two classical examples that were studied here follow a general pattern that can be used for other problems. One pursues the following recipe: (1) divide the unit d-cube into m^d subcells of equal size that are separated by a fattened grating; (2) fill the ith subcell with S_i, a set of k points on which the geometric object being analyzed attains its worst-case length in the subcell; (3) construct a graph G that is associated with the points $\bigcup_{i=1}^{m^d} S_i \subset [0, 1]^d$; (4) delete all edges of G that are long enough to span the fattened grating; (5) in each subcell, add edges to what remains following the deletion to form a heuristic graph G_i on S_i; (6) derive from this construction a recursion involving the length of a worst-case edge set; and (7) show that the recursion justifies (2.1(ii)) of Lemma 2.1. Of course, we must also show that the worst-case length satisfies (2.1(i)) of Lemma 2.1 to guarantee the result, although proving that the second recursion of Lemma 2.1 is satisfied is usually the task of greater difficulty.

This recipe would be unacceptably vague in the absence of explicit examples, but, be referring to the detailed treatment of the MST and TSP, the application of this technique to other problems should be reasonably straightforward.

The fact that the traveling salesman problem is computationally difficult and the minimal spanning tree problem is computationally easy serves to show that computational complexity is not at the heart of the technique used here. This intriguing circumstance provided one of our motivations for illustrating our technique with these particular problems. A second motivation came from the heuristic algorithms developed by Held and Karp (1970), (1971) which are driven by the observation that the minimal spanning tree problem is a relaxation of the traveling salesman problem.

Limit results like those given here seem to provoke two inevitable questions. The first question concerns the determination of the constants β_{MST} and β_{TSP} (for each $d \geqq 2$), and the second concerns the possibility of providing convergence rates more precise than $\rho(n) = \beta n^{(d-1)/d} + o(n^{(d-1)/d})$. The experience of trying to deal with the analogous questions under probabilistic assumptions leaves us with little hope for progress on these points. In particular, one should note that to sharpen the results of Moran (1984) to give the exact value of β_{TSP} would seem to require new geometric insights into the traveling salesman problem as well as improvements on the best

available results on sphere packing. These steps would be major advances in their own right. Perhaps the problem of improving the error term in our limit theorem to something sharper than $o(n^{(d-1)/d})$ would be easier than determining β; but, still, one would have to develop a technique that would be completely different than that given here.

REFERENCES

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

J. BEARDWOOD, J. H. HALTON, AND J. M. HAMMERSLEY (1959), *The shortest path through many points*, Proc. Cambridge Philos. Soc., 55, pp. 299-327.

L. T. FEJES-TÓTH (1940), *Uber einen geometrischen Satz*, Math. Z., 46, pp. 83-85.

L. FEW (1955), *The shortest path and the shortest road through n points in a region*, Mathematika, 2, pp. 141-144.

H. W. GHOSH (1949), *Expected travel among random points*, Bull. Calcutta Statist. Assoc., 2, pp. 83-87.

L. GODDYN (1988), *Quantizers and the worst case Euclidean traveling salesman problem*, J. Combin. Theory, Series B, submitted.

A. S. GOLDSTEIN (1988), personal communication.

A. S. GOLDSTEIN AND E. M. REINGOLD (1988), *Improved bounds on the traveling salesman problem in the unit cube*, Tech. Report, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL.

M. HELD AND R. M. KARP (1970), *The traveling salesman problem and minimum spanning trees*, Oper. Res., 18, pp. 1138-1162.

——— (1971), *The traveling salesman problem and minimum spanning trees: Part* II, Math. Programming, 1, pp. 6-25.

R. J. JESSEN (1942), *Statistical investigation of a sample survey for obtaining farm facts*, Bulletin of Iowa State College Agricultural Research Report 304.

H. J. KARLOFF (1987), *How long can a Euclidean traveling salesman tour be?* Tech. Report, Department of Computer Science, University of Chicago, Chicago, IL.

R. M. KARP AND J. M. STEELE (1985), *Probabilistic analysis of heuristics*, in The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds., John Wiley, New York.

P. C. MAHALANOBIS (1940), *A sample survey of the acreage under jute in Bengal*, Sankyā Ser. B., 4, pp. 511-531.

E. S. MARKS (1948), *A lower bound for the expected travel among m random points*, Ann. Math. Statist., 19, pp. 419-422.

S. MORAN (1984), *On the length of optimal* TSP *circuits in sets of bounded diameter*, J. Combin. Theory, Series B, 37, pp. 113-141.

C. H. PAPADIMITRIOU AND K. STEIGLITZ (1982), *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.

J. M. STEELE (1987), *Probabilistic and worst case analyses of some problems of combinatorial optimization in Euclidean space*, Tech. Report, Program in Operations Research and Statistics, Princeton University, Princeton, NJ.

K. J. SUPOWIT, E. M. REINGOLD, AND D. A. PLAISTED (1983), *The traveling salesman problem and minimum matchings in the unit square*, SIAM J. Comput., 12, pp. 144-156.

S. VERBLUNSKY (1951), *On the shortest path through a number of points*, Proc. Amer. Math. Soc., 2, pp. 904-913.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 288-300, April 1989

008

OPTIMAL PARALLEL 5-COLOURING OF PLANAR GRAPHS*

TORBEN HAGERUP†, MAREK CHROBAK‡, AND KRZYSZTOF DIKS‡

Abstract. We show that a 5-colouring of the vertices of an n-vertex planar graph may be computed in $O(\log n \log^* n)$ time by an exclusive-read exclusive-write parallel RAM with $O(n/(\log n \log^* n))$ processors. Our algorithm, while faster than all previously known methods, is at the same time the first parallel 5-colouring algorithm to exhibit an optimal speedup. Optimality is achieved through a method based on the accelerating cascades technique and of independent interest. It should be emphasized that although input to the algorithm is a planar graph, we do not require a planar embedding to be given as part of the input.

Other results concern the colouring of graphs of bounded genus and the construction of search structures for triangular planar subdivisions.

Key words. planar graphs, graph colouring, parallel random access machines (PRAMs), optimal parallel algorithms, planar subdivisions

AMS(MOS) subject classifications. 68C05, 68C25, 68E10

1. Introduction. The problem of colouring the vertices of a graph using few colours has given rise to one of the most intensively studied areas of graph theory. It is also important in practical terms because of its many applications in such fields as scheduling, resource allocation, and the construction and testing of VLSI circuits. A frequently encountered special case is that in which the graph to be coloured is planar. Computing a colouring that uses the smallest possible number of colours is known to be an NP-complete problem, even when restricted to the class of planar graphs [15] (more precisely, it is an NP-complete problem to decide in general whether a given planar graph is 3-colourable). Hence algorithms that colour planar graphs using a fixed and small number of colours are of interest.

Although every planar graph is 4-colourable, this is the famous "4-colour conjecture" whose lengthy proof [2], [3] seems so far to have put it somewhat out of the reach of efficient algorithm design. The situation changes if one allows five colours. A simple proof which establishes that every planar graph may be coloured using five colours also directly yields a sequential $O(n^2)$-time algorithm for doing so (n is the number of vertices in the graph). The algorithm first disassembles the graph by repeatedly removing a vertex of degree 5 or less. The vertices are then added back to an initially empty graph in the reverse order of their removal while a 5-colouring of the partially constructed graph is maintained. At each step it may be necessary to search and recolour almost the entire graph, which accounts for the quadratic running time. Lipton and Miller [19] reduced the running time to $O(n \log n)$ by a "batching" method in which a large number of vertices are removed from (and later added to) the graph in each step without increasing the time needed per step for searching and

* Received by the editors June 15, 1987; accepted for publication June 16, 1988. This research was carried out partly during a visit to the Institute of Theoretical Cybernetics, Slovak Academy of Sciences, Bratislava (M. Chrobak and K. Diks), and partly during a leave spent at Columbia University, New York, New York (M. Chrobak). A preliminary and abridged version of this paper was presented at the 14th International Colloquium on Automata, Languages, and Programming, Karlsruhe, Federal Republic of Germany, in July 1987.

† Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Federal Republic of Germany. The work of this author was supported by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität.

‡ Institute of Informatics, Warsaw University, PKiN VIII p., 00-901 Warsaw, Poland.

recolouring beyond linear. Since then a number of optimal $O(n)$-time algorithms have appeared [7], [20], [14], [22], [8]; even a sequential simulation of the algorithm presented in this paper could be added to the list. In general terms, most of these algorithms revert to the idea of removing vertices one by one but introduce small changes to the remaining graph in order to make it possible to later reinsert each vertex at constant cost.

The parallel version of the problem did not attract much attention until fairly recently. Bauernöppel and Jung [4] described a colouring method that may be used for several classes of graphs. When applied to planar graphs and run on a CRCW PRAM with $\Theta(n^4)$ processors, it yields an 8-colouring in $O(\log n)$ time. Pushing further in the same direction and concentrating on planar graphs, Diks [12] reduced the number of colours used to six. These algorithms were parallel "by birth," i.e., not derived from sequential algorithms. Parallel 5-colouring algorithms based on various sequential methods were subsequently discovered, for the most part independently, at least five times [21], [6], [17], [8], [16]. Our algorithm, while using the batching idea of [19], is largely a parallel implementation of the linear-time sequential 5-colouring algorithm by Chiba, Nishizeki, and Saito [7], with ideas of [10] used crucially in the parallelization. By generalizing a theorem due to Cole and Vishkin and based on the accelerating cascades technique [10], we achieve a running time of $O(\log n \log^* n)$ on an EREW PRAM with $O(n/(\log n \log^* n))$ processors. This result is optimal in the sense of having a linear product of time and number of processors and represents a considerable improvement on previous work: The earlier algorithms mentioned above have time bounds between $O((\log n)^2)$ and $O((\log n)^5)$ and miss optimality by a factor of at least $\Theta((\log n)^2)$. Two of the algorithms show a better performance on graphs that have already been embedded in the plane; however, here we do not consider a planar embedding to be available for free.

2. Notation and definitions. We assume familiarity with basic notions concerning directed and undirected graphs. Throughout, graphs are finite and without multiple edges, and undirected graphs are without loops.

In the following, let $G = (V, E)$ be a graph. If G is undirected, two vertices $u, v \in V$ are called *neighbours* in G if $(u, v) \in E$. If G is directed, we call u and v neighbours in G if either $(u, v) \in E$ or $(v, u) \in E$. A set $S \subseteq V$ is said to be *independent* in G if it does not contain two vertices that are neighbours in G. S is *maximal* independent in G if S is independent in G and there is no independent vertex set S' in G with $S \subset S'$. For $w \in V$, the graph obtained from G by removing w together with all edges incident on w is denoted by $G - \{w\}$. In general, whenever vertices are removed from a graph, the intended meaning is that edges incident on such vertices are removed as well.

From now on, let G be undirected. For $u \in V$, we use $N(u)$ to denote the set of neighbours of u and $\deg(u)$ to denote the degree of u, i.e., $\deg(u) = |N(u)|$.

A (vertex) *colouring* of G is an assignment of colours to the vertices of G such that adjacent vertices receive distinct colours, i.e., a function g defined on V with the property that $g(u) \neq g(v)$ for all $(u, v) \in E$. g uses $|g(V)|$ colours, and for $k \geq 1$, G is said to be *k-colourable* if there is a colouring of G that uses at most k colours.

For $u, v \in V$ with $(u, v) \notin E$, the *identification* of u and v is an operation which replaces u and v and their incident edges by a new vertex z, adjacent to exactly those vertices in $V \setminus \{u, v\}$ that were adjacent to either u or v (or both) in the original graph. Identification of more than two pairwise nonadjacent vertices is defined analogously or may be thought of as repeated identification of two vertices. When C is an independent vertex set, we denote the identification of all vertices in C by $\langle C \rangle$.

A *planar embedding* of G is a function $\mathscr{E}$ that maps the vertices of G to distinct points in $\mathbb{R}^2$ and each edge $e=(u, v)\in E$ to a Jordan curve in $\mathbb{R}^2$ from $\mathscr{E}(u)$ to $\mathscr{E}(v)$ such that for all $e=(u, v)\in E$, $\mathscr{E}(e)\cap(\mathscr{E}(V)\cup\mathscr{E}(E\backslash\{e\}))\subseteq\{\mathscr{E}(u), \mathscr{E}(v)\}$ (i.e., edges do not cross). G is *planar* if there exists a planar embedding of G. Euler's formula [13] implies that if $G=(V, E)$ is planar, then $|E|\leqq 3|V|$. We occasionally do not distinguish between a vertex and its image under a given planar embedding.

Given a planar embedding $\mathscr{E}$ of G and a vertex $w\in V$ with $N(w)=\{u_1, \cdots, u_l\}$, the curves $\mathscr{E}((w, u_1)), \cdots, \mathscr{E}((w, u_l))$ occur in a particular cyclic order around $\mathscr{E}(w)$, namely the order in which they are encountered in a counterclockwise scan around $\mathscr{E}(w)$. If this cyclic order is $\mathscr{E}((w, u_1)), \cdots, \mathscr{E}((w, u_l))$, we also say that the neighbours of w occur in the cyclic order $u_1, \cdots, u_l$ around w in the planar embedding $\mathscr{E}$. The *faces* of $\mathscr{E}$ are the connected regions of $\mathbb{R}^2-\mathscr{E}(V\cup E)$. All graphs introduced in following sections, unless otherwise qualified, will be undirected.

A PRAM (parallel random access machine) consists of a finite number p of processors (RAMs) operating synchronously on common, shared memory cells numbered $0, 1, \cdots$. We assume that the processors are numbered $1, \cdots, p$ and that each processor is able to read its own number. All processors execute the same program. We use the unit-cost model in which each memory cell can hold integers of size polynomial in the size of the input, and each processor is able to carry out usual arithmetic operations including multiplication and integer division on such numbers in constant time. In addition, we assume the existence of (constant-time) instructions for bitwise logical operations on integers, which are then considered as represented in the binary number system (e.g., 2's complement), as well as for conversion from the unary to the binary number system.

One distinguishes between various types of PRAMs. EREW (exclusive-read exclusive-write) PRAMs allow no memory cell to be accessed simultaneously by more than one processor. In contrast, CRCW (concurrent-read concurrent-write) PRAMs allow simultaneous reading as well as simultaneous writing of each cell by arbitrary sets of processors, with some rule defining the exact semantics of simultaneous writing. CREW (concurrent-read exclusive-write) PRAMs allow simultaneous reading, but not simultaneous writing.

For $k\geqq 0$, $\log^{(k)}$ denotes the k-fold iterated logarithm function, i.e., $\log^{(0)} n=n$ and $\log^{(i)} n=\log\log^{(i-1)} n$, for $i\geqq 1$. For $n\geqq 1$, $\log^* n=\min\{i\geqq 1|\log^{(i)} n\leqq 1\}$. All logarithms in the paper are to base 2.

3. 5-colouring using a linear number of processors. Before we explain the technical details of the algorithm, we provide the following sketch of the main ideas involved. Note that the description assumes the availability of n processors. A later refinement reduces the number of processors needed to $O(n/(\log n \log^* n))$.

Starting from the given planar graph $G_0=G$, the algorithm produces in successive stages a sequence $G_0, G_1, G_2, \cdots$ of planar graphs. The size of each G_i is at most c times that of its predecessor, for some constant $c<1$. Hence after $O(\log n)$ stages there remains a graph which is trivial to colour.

For $i\geqq 0$, G_{i+1} is derived from G_i by applying a number of *reductions* to vertices of G_i. A reduction at a vertex w consists of the removal of w together with the possible identification of certain neighbours of w.

Suppose that w is a vertex in a graph G. If we derive a graph G' from G by identifying sufficiently many neighbours of w to reduce the degree of w to at most 4, then it is easy to extend any 5-colouring of $G'-\{w\}$ to a 5-colouring of G. First undo the identifications of neighbours of w, letting each new vertex created in the process

inherit the colour of the vertex from which it is derived (note that vertices that are identified are always nonadjacent). Then stick w back in place. By the colouring convention just mentioned, the neighbours of w will be coloured with at most four colours; hence the fifth colour may be used to colour w.

The above procedure cannot be applied indiscriminately. In order to be able to even guarantee that $G'-\{w\}$ is 5-colourable, one must ensure that it is planar. It turns out that there is a possible reduction preserving planarity at every vertex of degree at most 6; while such reductions are not always easy to determine, we identify a sufficiently large set of vertices for which they can be found in constant time.

The reductions that produce G_{i+1} from G_i are all executed in parallel. The set W of vertices at which reductions take place must be chosen with some care. First of all it must satisfy the constraints imposed by the need to preserve planarity. Secondly, the vertices in W must possess certain independence properties, both with respect to the current planar graph G_i and with respect to its representation. And lastly, W should contain a constant fraction of all vertices. The computation of a suitable set W is made difficult by the presence of vertices of high degree. Therefore, we show that such vertices are sufficiently rare that one may refrain from attempting certain types of reductions in their vicinity. After the exclusion of such reductions, the problem may be cast as one of finding a sufficiently large independent vertex set in a graph of bounded degree. Using a technique first employed by Cole and Vishkin, we show how to solve the latter problem in time $O(\log^* n)$.

Our choice of W is such that the reductions at vertices in W can be carried out in constant time. Hence each stage takes $O(\log^* n)$ time, and the running time of the entire algorithm is $O(\log n \log^* n)$.

Given a graph G, we formally define a *reduction* in G to be a set $r=\{w, C_1, \cdots, C_s\}$, where w is a vertex in G and $C_1, \cdots, C_s$ are pairwise disjoint independent subsets of $N(w)$. w is called the *center* of r. To *execute* the reduction r means to remove w and to apply the identifications $\langle C_1\rangle, \cdots, \langle C_s\rangle$ to the resulting graph (in any order).

Now fix a constant $K \geqq 12$. We call a vertex *small* if it has at most K neighbours, *large* otherwise.

DEFINITION. A vertex w in a graph G is called *reducible* if one of the following holds:

(1) $\deg(w) \leqq 4$.

(2) $\deg(w)=5$, and w has at most one large neighbour.

(3) $\deg(w)=6$, all neighbours of w are small, and the subgraph spanned by $N(w)$ is Hamiltonian.

The last part of condition (3) states that G contains a simple cycle whose vertices are exactly the neighbours of w.

DEFINITION. A reduction r centered at a reducible vertex w is called *safe* if one of the following holds:

(1) $\deg(w) \leqq 4$, and $r=\{w\}$.

(2) $\deg(w)=5$, and $r=\{w, \{x, y\}\}$ for some pair $\{x, y\}$ of distinct, small neighbours of w.

(3) $\deg(w)=6$, and either

(a) $r=\{w, C\}$ for some $C \subseteq N(w)$ with $|C|=3$, or

(b) $r=\{w, \{x_1, y_1\}, \{x_2, y_2\}\}$ for some pairwise distinct vertices $x_1, y_1, x_2, y_2 \in N(w)$ that occur in the cyclic order x_1, y_1, x_2, y_2 on a Hamilton cycle of the graph spanned by $N(w)$.

Note that by the definition of a reduction, the vertices to be identified in (2) and (3) must be pairwise nonadjacent.

The algorithm works by executing safe reductions centered at reducible vertices. It is based on six lemmas given below. Lemma 1 considers the problem locally and shows that there is a safe reduction centered at every reducible vertex. Lemma 2 guarantees that safe reductions preserve planarity. Lemmas 3 and 4, technical in nature, demonstrate that the number of reducible vertices is not too small, and Lemma 5 shows that a large independent vertex set in an n-vertex graph of bounded degree can be found in $O(\log^* n)$ time. Lemma 6, building on the previous lemma, finally states that reductions centered at a constant fraction of the reducible vertices can be carried out in $O(\log^* n)$ time.

LEMMA 1. *Given a reducible vertex w in a planar graph G, there is a safe reduction centered at w which can be determined in constant time by one processor from the adjacency lists of w and its small neighbours.*

Proof. If $\deg(w) \leqq 4$, there is nothing to show. If $\deg(w) = 5$, then we simply have to find and identify two small nonadjacent neighbours of w. These necessarily exist by Kuratowski's theorem [13] since otherwise G would contain a complete subgraph on five vertices, namely w and four of its small neighbours.

Suppose now that w has exactly six neighbours $u_1, \cdots, u_6$, and let their cyclic order on a Hamilton cycle of the graph spanned by $N(w)$ be $u_1, \cdots, u_6$. Note that this must also (up to reversal) be the cyclic order in which $u_1, \cdots, u_6$ occur around w in any planar embedding of G.

If $N(w)$ contains an independent set of three vertices, then we can identify these. Otherwise we may assume (after a cyclic renaming) that G contains the edge (u_1, u_3). Then G cannot also contain the edge (u_1, u_5), since in that case $\{u_2, u_4, u_6\}$ would form an independent set, cf. Fig. 1. Hence $\{w, \{u_1, u_5\}, \{u_2, u_4\}\}$ is a safe reduction centered at w. □

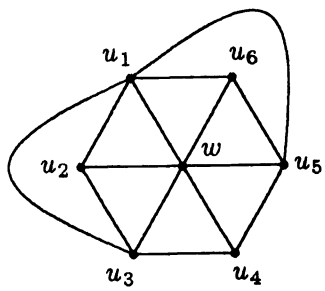


FIG. 1. *If u_1 is adjacent to both u_3 and u_5, then $\{u_2, u_4, u_6\}$ is an independent set.*

LEMMA 2. *The graph G' obtained from a planar graph G by the execution of a safe reduction is again planar.*

Proof. Consider first the case of a safe reduction of the form $\{w, \{x_1, y_1\}, \{x_2, y_2\}\}$. Note as above that x_1 and y_1 as well as x_2 and y_2 are adjacent in the cyclic order of x_1, y_1, x_2, and y_2 around w in any planar embedding of G. The planarity of G' is now evident from geometric considerations (see Fig. 2). The remaining cases are easy.

LEMMA 3. *Let $G = (V, E)$ be a planar graph with n vertices and m edges, and let*

$$Z = \{w \in V \mid \deg(w) \geqq 3 \text{ and the subgraph spanned by } N(w) \text{ is not Hamiltonian}\}.$$

Then $m \leqq 3n - \frac{1}{4}|Z|$.

Proof. Take any planar embedding of G and observe that each vertex in Z lies on the boundary of a face whose boundary contains at least four distinct vertices. Now for each such face F, add to G a new vertex u_F and edges joining u_F to all vertices

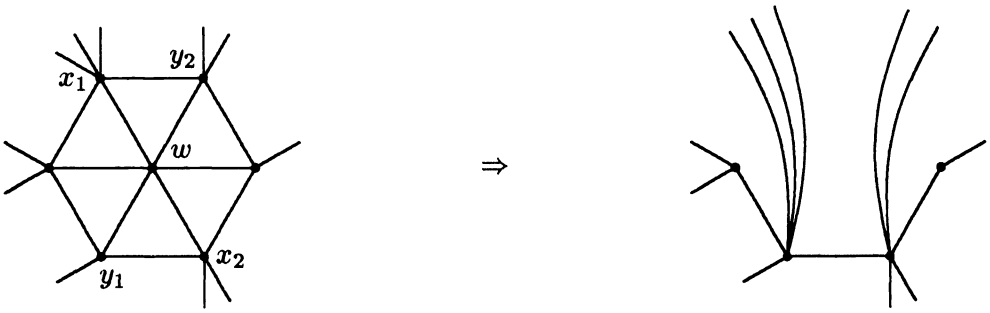


FIG. 2. *Execution of the safe reduction* $\{w, \{x_1, y_1\}, \{x_2, y_2\}\}$.

on the boundary of F (if a vertex occurs more than once on the boundary, join it to u_F by just one edge). The resulting graph is obviously planar. Let n' and m' be the total number of vertices and edges added, respectively. Clearly $m' \geqq 4n'$ and $m' \geqq |Z|$, and Euler's formula applied to the augmented graph gives

$$(m+m') \leqq 3(n+n')$$

or

$$m \leqq 3n+3n'-m' \leqq 3n-\tfrac{1}{4}m' \leqq 3n-\tfrac{1}{4}|Z|. \qquad \square$$

LEMMA 4. *Let A be the set of reducible vertices in a planar graph G on n vertices. Then $|A| \geqq n/196$.*

Proof. For $j = 0, 1, \cdots$, let n_j be the number of vertices in G of degree j, and let $\hat{n}_5$ be the number of reducible vertices of degree 5, $\bar{n}_6$ the number of reducible vertices of degree 6, and $\hat{n}_6$ the number of vertices of degree 6 all of whose neighbours are small. Then

$$|A| = \sum_{j=0}^{4} n_j + \hat{n}_5 + \bar{n}_6.$$

We first informally sketch why one has $|A| = \Omega(n)$. Let $n' = n_0 + \cdots + n_4 + \hat{n}_5 + \hat{n}_6$. As was shown in [8] and in [17], $n' = \Omega(n)$. Now, if $\hat{n}_6 - \bar{n}_6$ is small, then $|A|$ is close to n' and hence $|A| = \Omega(n)$. On the other hand, if $\hat{n}_6 - \bar{n}_6$ is large, then we may conclude from Lemma 3 that G is very sparse, i.e., contains considerably fewer than $3n$ edges. It then follows immediately from Euler's formula that G contains many vertices of degree at most 5. With some care, one may even show that $n_0 + \cdots + n_4 + \hat{n}_5 = \Omega(n)$. The details follow.

Let m be the number of edges in G and put $\varepsilon = 6 - 2m/n$. $\varepsilon \geqq 0$ by Euler's formula, and

$$2m = (6-\varepsilon)n$$

or

$$\sum_{j=0}^{\infty} jn_j = (6-\varepsilon) \sum_{j=0}^{\infty} n_j.$$

Isolating the terms containing n_5 gives

$$(1-\varepsilon)n_5 = \sum_{j=0}^{4} (j-6+\varepsilon)n_j + \sum_{j=6}^{\infty} (j-6+\varepsilon)n_j$$

and hence

$$n_5 \geqq \sum_{j=6}^{\infty} (j-6+\varepsilon)n_j - 6 \sum_{j=0}^{4} n_j. \tag{1}$$

Let m' be the number of edges in G that have at least one large endpoint. By the definition of $\hat{n}_5$ and $\hat{n}_6$,

$$m' \geqq 2(n_5 - \hat{n}_5) + (n_6 - \hat{n}_6).$$

On the other hand,

$$m' \leqq \sum_{j=K+1}^{\infty} jn_j,$$

and therefore

$$2(n_5 - \hat{n}_5) + (n_6 - \hat{n}_6) \leqq \sum_{j=K+1}^{\infty} jn_j$$

or

$$2\hat{n}_5 + \hat{n}_6 \geqq 2n_5 + n_6 - \sum_{j=K+1}^{\infty} jn_j. \tag{2}$$

Next add $\alpha = \frac{15}{8}$ times inequality (1) to the above inequality (2). The result is

$$\begin{aligned} 2\hat{n}_5 + \hat{n}_6 &\geqq (2-\alpha)n_5 + n_6 - \sum_{j=K+1}^{\infty} jn_j + \alpha \sum_{j=6}^{\infty} (j-6+\varepsilon)n_j - 6\alpha \sum_{j=0}^{4} n_j \\ &= \frac{n_5}{8} + n_6 + \alpha \sum_{j=6}^{K} (j-6+\varepsilon)n_j + \sum_{j=K+1}^{\infty} ((\alpha-1)j - 6\alpha + \alpha\varepsilon)n_j - 6\alpha \sum_{j=0}^{4} n_j \\ &\geqq \frac{n_5}{8} + n_6(1+\alpha\varepsilon) + \alpha \sum_{j=7}^{K} n_j + \sum_{j=K+1}^{\infty} ((\alpha-1)(K+1) - 6\alpha)n_j - 6\alpha \sum_{j=0}^{4} n_j. \end{aligned}$$

Since $(\alpha-1)(K+1) - 6\alpha \geqq \frac{1}{8}$, we finally get

$$6\alpha \sum_{j=0}^{4} n_j + 2\hat{n}_5 + \hat{n}_6 \geqq \frac{n_5}{8} + n_6(1+\alpha\varepsilon) + \frac{1}{8} \sum_{j=7}^{\infty} n_j. \tag{3}$$

Now consider two cases:

Case 1. $\varepsilon \geqq \frac{1}{32}$. Since $n_6 \geqq \hat{n}_6$,

$$6\alpha \sum_{j=0}^{4} n_j + 2\hat{n}_5 \geqq \frac{n_5}{8} + \frac{\alpha n_6}{32} + \frac{1}{8} \sum_{j=7}^{\infty} n_j$$

and, after multiplication by $32/\alpha > 8$ and addition of $\sum_{j=0}^{4} n_j$,

$$193|A| \geqq (6 \cdot 32 + 1) \sum_{j=0}^{4} n_j + \frac{64}{\alpha} \hat{n}_5 \geqq \sum_{j=0}^{4} n_j + n_5 + n_6 + \sum_{j=7}^{\infty} n_j = n,$$

from which the desired conclusion follows.

Case 2. $\varepsilon < \frac{1}{32}$. Since $m = 3n - (\varepsilon n/2)$, Lemma 3 implies that $\hat{n}_6 - \bar{n}_6 \leqq 2\varepsilon n$. Hence from (3),

$$6\alpha \sum_{j=0}^{4} n_j + 2\hat{n}_5 + \bar{n}_6 \geqq \frac{n_5}{8} + n_6 + \frac{1}{8} \sum_{j=7}^{\infty} n_j - 2\varepsilon n$$

and

$$(6\alpha+1)|A| \geqq (6\alpha+1) \sum_{j=0}^{4} n_j + 2\hat{n}_5 + \bar{n}_6 \geqq \left(\frac{1}{8} - 2\varepsilon\right) n \geqq \frac{n}{16}.$$

The desired conclusion again follows since $16(6\alpha+1) = 196$. □

We use a standard representation of undirected graphs. Vertices are represented by integers, and the graph itself is represented by a set of doubly linked adjacency lists. The adjacency list of each vertex u contains exactly one entry for each neighbour v of u in the graph. This entry contains, in addition to an identification of v, a pointer to u's entry in the adjacency list of v. Pointers of the latter type are called *cross links.*

Each vertex has an associated processor with constant-time access to the adjacency list of the vertex. Note that the presence of cross links enables conflict-free communication between processors associated with adjacent vertices.

We shall say that a graph is represented with *vertex number bound q* if each integer representing a vertex in the graph lies in the range $0, \cdots, q-1$, and if q and $\log^* q$ are known to each processor associated with a vertex. Our model of computation is the EREW PRAM.

LEMMA 5. *Let a graph $G=(V, E)$ with n vertices and maximum vertex degree bounded by a constant d be represented with vertex number bound q. Then an independent vertex set in G containing at least $n/6^d$ vertices may be computed by n processors in $O(\log^* q)$ time.*

Remark. A highly elegant algorithm computing a *maximal* independent vertex set within the stated resource bounds was found recently by Goldberg, Plotkin, and Shannon [16]. In terms of constant factors, their algorithm is far superior to the one given here.

Proof. Consider G as a directed graph by treating each undirected edge (u, v) as two directed edges (u, v) and (v, u). Each processor associated with some $u \in V$ with, say, l outgoing edges labels these $1, \cdots, l$ in some arbitrary order. For $j=1, \cdots, d$, let E_j be the set of edges labeled j. For $j=1, \cdots, d$, we eliminate "conflicts" caused by edges in E_j. More precisely, execute the following:

$V_0 := V$;
for $j := 1$ **to** d
do $V_j :=$ an independent vertex set in the graph $(V_{j-1}, E_j \cap (V_{j-1} \times V_{j-1}))$
with $|V_j| \geqq |V_{j-1}|/6$;

Then V_d will be an independent set in G containing at least $n/6^d$ vertices. It hence remains only to show that in any subgraph of G with t vertices and maximum out-degree 1, an independent vertex set containing at least $t/6$ vertices may be found in $O(\log^* q)$ steps.

Consider such a subgraph and remove each vertex with in-degree $\geqq 2$. This leaves a collection H of vertex-disjoint simple paths and simple cycles. Note that H contains at least $t/2$ vertices. A trivial extension of a result in [10], which considers the case of a *single* simple path or simple cycle, shows that a maximal independent vertex set in H can be found in $O(\log^* q)$ time. This concludes the proof since a maximal independent vertex set in H necessarily contains at least one third of the vertices in H, i.e., at least $t/6$ vertices. □

LEMMA 6. *There is a constant $\gamma > 0$ with the following property: Let A be a set of reducible vertices in an n-vertex graph $G=(V, E)$ represented with vertex number bound q and, for each $u \in A$, let r_u be a safe reduction centered at u. Then at least $\gamma|A|$ of the reductions in the set $\{r_u | u \in A\}$ may be executed by n processors in $O(\log^* q)$ time.*

Proof. Consider first the execution of a single safe reduction. The identification of vertices $x_1, \cdots, x_l$ is carried out by choosing a representative, x_1, say, renaming all adjacency list entries for $x_2, \cdots, x_l$ to be entries for x_1, concatenating the adjacency lists of $x_1, \cdots, x_l$ to obtain the new adjacency list for x_1, and removing $x_2, \cdots, x_l$ as well as duplicate entries of edges incident on x_1. Since the total number of vertices and edges involved is bounded by a constant, the computation can be done in constant time by a single processor. This also applies to the removal of w together with its at most six incident edges.

We now introduce an undirected auxiliary graph H on the vertex set A and with edges (u, v) intended to mean that the reductions r_u and r_v should not both be executed.

For all $u, v \in A$ with $u \neq v$, H contains an edge (u, v) exactly if one of the following holds:

(1) There is a path in G from u to v of length at most 4 and containing no large vertices.

(2) There are small vertices x and y, $x \in \{u\} \cup N(u)$, $y \in \{v\} \cup N(v)$, such that x and y have a common neighbour in whose adjacency list the entries for x and y are consecutive.

The maximum vertex degree of H is clearly bounded by a constant, and it is not difficult to see that H can be constructed in constant time, even in our chosen EREW PRAM model. By Lemma 5, there is a constant $\gamma > 0$ such that an independent set W in H of size at least $\gamma|A|$ can be computed in $O(\log^* q)$ time. We finish by letting the processor associated with w execute the reduction r_w, simultaneously for all $w \in W$, as described above. The well-definedness of this, as well as the fact that the computation can be carried out in constant time and yields the same graph as would have been obtained by sequentially executing the reductions one by one in any order, follow from the observation that processors executing reductions centered at distinct vertices in W operate on (read and write) disjoint sets of memory cells (let us say that they operate without *contention*). To see this, note first that if two such processors P_1 and P_2 operate on the same adjacency list L, then by (1) above, L is necessarily the adjacency list of a large vertex, the operations of P_1 and P_2 on L are limited to updating or deleting certain sets of single list entries, and no entry in L is updated or deleted by both P_1 and P_2; therefore updates cause no contention. And by (2), L does not even contain two consecutive entries such that P_1 deletes one, and P_2 the other. Hence deletion, which involves resetting pointers in successor and predecessor entries, causes no contention either. □

THEOREM 1. *Given a planar graph G on n vertices, a 5-colouring of the vertices of G may be computed in $O(\log n \log^* n)$ time by an* EREW PRAM *with n processors using $O(n)$ space.*

Proof. We use the following algorithm, γ being the constant from Lemma 6 and β another constant:

```
(01)  G_0 = (V_0, E_0) := G;
(02)  k := β⌈log n⌉;
(03)  for i := 0 to k − 1
(04)  do begin
(05)       A_i := the set of reducible vertices in G_i;
(06)       For all u ∈ A_i, compute a safe reduction r_u centered at u;
(07)       Let G_{i+1} = (V_{i+1}, E_{i+1}) be a graph obtained from G_i
               by executing at least γ|A_i| of the reductions in
               {r_u | u ∈ A_i};
(08)     end;
(09)  Colour G_k;
(10)  for i := k − 1 downto 0
(11)  do begin
(12)       Reconstruct G_i from G_{i+1};
(13)       Extend the colouring of G_{i+1} to a 5-colouring of G_i;
(14)     end;
```

We assume, as is standard, that G is presented to the algorithm in the form of a set of adjacency lists, and that vertices are represented by integers of size $O(n)$. An internal representation of G with vertex number bound $O(n)$ may then be constructed in

$O(\log n)$ time. In particular, cross links are computed by means of a sorting of the (undirected) edges [1], [9].

Lines (05) and (06) take constant time, whereas line (07) can be executed in time $O(\log^* n)$ by Lemma 6. Since $|V_{i+1}| \leqq |V_i| - \gamma|A_i| \leqq (1-\gamma/196)|V_i|$ for $i = 0, 1, \cdots$, a suitable choice of the constant β ensures that $|V_k| \leqq 1$, making the colouring of G_k trivial (strictly speaking, if $|V_i| = 0$, G_i is no longer a graph). Hence lines (01)-(09) take $O(\log n \log^* n)$ time. By running the process backwards, the reconstruction in lines (10)-(14) may be done within the same time bound. The procedure for extending a colouring of G_{i+1} to a colouring of G_i was already sketched: Whenever a vertex z is split into two vertices x and y, x and y inherit the colour of z, and whenever a vertex w is reinserted, it is coloured by a colour different from all of the at most 4 colours with which its neighbours are coloured. This is easy except that in order to avoid read conflicts, one must associate with each edge the colours of its endpoints. □

4. The optimal algorithm.

THEOREM 2. *Given a planar graph G on n vertices, a 5-colouring of the vertices of G may be computed in $O(\log n \log^* n)$ time by an* EREW PRAM *with $O(n/(\log n \log^* n))$ processors using $O(n^2)$ space.*

Proof. An internal representation of G with vertex number bound $O(n)$ may be constructed in $O(\log n \log^* n)$ time. In particular, since it is no longer feasible to sort the edges of G, cross links are established by the following simple method which uses $\Theta(n^2)$ space: Each processor which initially discovers in the input the entry of a vertex v in the adjacency list of a vertex u places a pointer to this entry in a cell $\Gamma_{u,v}$ associated with (the ordered pair) (u, v) and proceeds, if desired, to copy the pointer to a cell found using the information in $\Gamma_{v,u}$.

The remaining analysis is in fact identical to the proof of Theorem 5.1 in [10]. It applies to any algorithm that has an n-processor implementation consisting of $O(\log n)$ stages with the following characteristics:

(1) Each stage consists of some constant-time computation plus a constant number of computations of maximal independent vertex sets in simple cycles or simple paths by the Cole/Vishkin [10] method.

(2) For $i = 1, 2, \cdots$, the number of active processors in the ith stage is at most $2^{1-i}n$. Once a processor has become inactive, it remains so.

To see how the algorithm of Theorem 1 fits into this more general setting, note that the number of active processors decreases geometrically (in the first loop of the algorithm; the second loop may of course be treated analogously) because the number of vertices does. If a sufficiently large but constant number of stages is considered as a unit, each such unit will decrease the number of active processors by a factor of at least 2.

Let us deal with the general case and use the word *task* for the computation carried out by a single processor in the n-processor implementation. A task is said to be *active* when its associated processor is active. Our method will be a stage-by-stage simulation of the n-processor implementation using the available $p = O(n/(\log n \log^* n))$ processors, with each processor being responsible for (i.e., executing the instructions of) several tasks. Define the *maximum load* at a particular point during the simulation to be the maximum number of tasks for which a single processor is responsible. We need two preliminary results, both of which are proved in detail in Cole/Vishkin [10].

LEMMA 7. *Suppose that a set of m "items" is distributed among the p processors with each processor holding at most h items. Then the set of items may be evenly distributed*

among the processors (i.e., *such that each processor has either* $\lfloor m/p \rfloor$ *or* $\lceil m/p \rceil$ *items*) *in time* $O(\log p + h) = O(\log n + h)$.

LEMMA 8. *Let G be a simple cycle or a simple path on t vertices represented with vertex number bound $O(n)$. Then a maximal independent vertex set in G may be found in $O(\log n)$ time by $O(t/\log n)$ processors.*

We denote by REDISTRIBUTE the algorithm implied by Lemma 7. It is a simple application of the standard "parallel prefix computation." The redistributed items will be task descriptions or, equivalently, processor numbers of the processors associated with the given tasks in the n-processor implementation. After a call of REDISTRIBUTE, executed when the number of active tasks is m, the maximum load will be $O(m/p)$.

The algorithm guaranteed by Lemma 8 is much slower than the one used as a subroutine in the algorithm of Lemma 5, but it possesses an optimal time-processor product. A stage in which this algorithm is used instead of its faster counterpart will be called an *optimal* stage. We may substitute at will optimal stages for usual stages.

Let $k \geqq 0$ be the integer such that $\log^{(k+1)} n < \log^* n \leqq \log^{(k)} n$. Note that $k \leqq \log^* n$. We execute the following algorithm:

```
(01)  Distribute the n tasks evenly among the p processors;
(02)  for i := 1 to ⌈2 log^(k+1) n⌉
(03)  do begin
(04)       Execute one optimal stage;
(05)       REDISTRIBUTE;
(06)     end;
(07)  for j := k − 1 downto 1
(08)  do begin
(09)       Execute ⌈2 log^(j+1) n⌉ (usual) stages;
(10)       REDISTRIBUTE;
(11)     end;
(12)  Execute the remaining O(log n) (usual) stages;
```

It remains only to show that the execution time of the algorithm, using p processors, is $O(n/p)$.

Line (01) clearly takes constant time. For $i = 1, \cdots, \lceil 2 \log^{(k+1)} n \rceil$, the maximum load immediately before the ith execution of lines (04)-(05) is $O(n/(2^{i-1}p))$. Hence by Lemmas 7 and 8, the time needed for the ith execution of lines (04)-(05) is $O(n/(2^{i-1}p) + \log n)$, and the execution of the entire loop in lines (02)-(06) takes time

$$O\left(\sum_{i=1}^{\lceil 2 \log^{(k+1)} n \rceil} \left(\frac{n}{p}\frac{1}{2^{i-1}} + \log n\right)\right) = O(n/p).$$

The execution of lines (01)-(06) reduces the maximum load to $O(n/(p(\log^{(k)} n)^2)) = O(\log n)$. Hence by Lemma 7, each execution of line (10) takes $O(\log n)$ time. Furthermore, the maximum load immediately before the $(k-j)$th execution of line (09), for $j = k-1, \cdots, 1$, is $O(n/(p(\log^{(j+1)} n)^2))$. Hence the time required for the $(k-j)$th execution of line (09) is

$$O\left(\frac{n}{p}\frac{\log^{(j+1)} n \log^* n}{(\log^{(j+1)} n)^2}\right),$$

giving a total execution time for the loop in lines (07)-(11) of

$$O\left(\sum_{j=1}^{k-1} \left(\frac{n}{p}\frac{\log^* n}{\log^{(j+1)} n} + \log n\right)\right) = O(n/p).$$

Finally, when line (12) is reached, the maximum load has been reduced to 1. Hence the remaining stages may be executed as in the n-processor implementation in time $O(\log n \log^* n) = O(n/p)$. □

5. Further results. Techniques used in this paper may also be applied to the following problems:

Computation of maximal independent sets in planar graphs. The 5-colouring produced by our algorithm easily yields a maximal independent vertex set S of the input graph. Let $S = \varnothing$ initially, and then step sequentially through the set of colours, for each colour simultaneously adding to S all vertices of that colour that have no neighbours in S. The test of whether given vertices have neighbours in S, although not trivial in the EREW PRAM model, can nevertheless be carried out optimally in logarithmic time.

Colouring graphs of bounded genus. One consequence of Euler's formula [5] for graphs with n vertices, m edges, and genus g reads $m \leqq 3n + 6(g-1)$, implying that the number of vertices of degree at most 6 is at least $(n - 12g + 12)/7$. Hence as long as n is larger than some number depending only on g, there are at least $n/8$ vertices of degree at most 6. As in the proof of Lemma 6, one may construct a conflict graph H of bounded degree on the set of these vertices, find a large independent set, and carry out reductions centered at the vertices in the independent set, thereby in $O(\log^* n)$ time reducing the size of the graph by a constant factor. If seven colours are allowed, there is never any need to identify vertices. Hence the genus does not increase, and the process may be repeated until the proportion of vertices of degree at most 6 drops below $\frac{1}{8}$. Assuming that g is bounded by a constant, the remaining graph may be coloured optimally in constant time.

We conclude that an n-vertex graph G of bounded genus can be coloured with at most $\max\{\chi(G), 7\}$ colours in $O(\log n \log^* n)$ time by $O(n/(\log n \log^* n))$ processors. Here $\chi(G)$ is the chromatic number of G, i.e., the smallest k such that G is k-colourable. Note that for all $g \geqq 1$, there are graphs of genus g that cannot be coloured with fewer than seven colours. Hence the above procedure is in a certain sense optimal with respect to the number of colours used.

Construction of search structures for triangular planar subdivisions. We are here given a planar embedding $\mathscr{E}$ of an n-vertex graph all of whose faces are triangles (i.e., the boundary of each face consists of three line segments), and the problem is to construct a data structure that will allow a single processor to determine in $O(\log n)$ time the face of $\mathscr{E}$ containing a given query point. Our solution is a straightforward parallel implementation of the sequential algorithm of [18]. As above, it has the same resource requirements as the 5-colouring algorithm, $O(\log n \log^* n)$ time with an optimal number of processors. Essentially the same algorithm was discovered independently by Dadoun and Kirkpatrick [11].

Acknowledgments. We would like to thank K. Mehlhorn, L. Banachowski, T. Radzik, and M. Yung for their encouragement and for many useful suggestions.

REFERENCES

[1] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 1-9.

[2] K. APPEL AND W. HAKEN, *Every planar map is four colorable. Part I: Discharging*, Illinois J. Math., 21 (1977), pp. 429–490.
[3] K. APPEL, W. HAKEN, AND J. KOCH, *Every planar map is four colorable. Part II: Reducibility*, Illinois J. Math., 21 (1977), pp. 491–567.
[4] F. BAUERNÖPPEL AND H. JUNG, *Fast parallel vertex colouring*, in Proc. 5th International Conference on Fundamentals of Computation Theory, Lecture Notes in Computer Science, 199, L. Budach, ed., 1985, pp. 28–35.
[5] B. BOLLOBÁS, *Extremal Graph Theory*, Academic Press, London, 1978.
[6] J. F. BOYAR AND H. J. KARLOFF, *Coloring planar graphs in parallel*, J. Algorithms, 8 (1987), pp. 470–479.
[7] N. CHIBA, T. NISHIZEKI, AND N. SAITO, *A linear 5-coloring algorithm of planar graphs*, J. Algorithms, 2 (1981), pp. 317–327.
[8] M. CHROBAK AND K. DIKS, *Two algorithms for coloring planar graphs with 5 colors*, Tech. Report, Columbia University, January 1987.
[9] R. COLE, *Parallel merge sort*, in Proc. 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 511–516.
[10] R. COLE AND U. VISHKIN, *Deterministic coin tossing with applications to optimal parallel list ranking*, Inform. and Control, 70 (1986), pp. 32–53.
[11] N. DADOUN AND D. G. KIRKPATRICK, *Parallel processing for efficient subdivision search*, in Proc. 3rd Annual ACM Symposium on Computational Geometry, 1987, pp. 205–214.
[12] K. DIKS, *A fast parallel algorithm for six-colouring of planar graphs*, in Proc. 12th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 233, J. Gruska, B. Rovan and J. Wiedermann, ed., 1986, pp. 273–282.
[13] S. EVEN, *Graph Algorithms*, Pitman, London, 1979.
[14] G. N. FREDERICKSON, *On linear-time algorithms for five-coloring planar graphs*, Inform. Process. Lett., 19 (1984), pp. 219–224.
[15] M. R. GAREY, D. S. JOHNSON, AND L. STOCKMEYER, *Some simplified* NP*-complete graph problems*, Theoret. Comput. Sci., 1 (1976), pp. 237–267.
[16] A. V. GOLDBERG, S. A. PLOTKIN, AND G. E. SHANNON, *Parallel symmetry-breaking in sparse graphs*, in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 315–324.
[17] T. HAGERUP, *Parallel 5-colouring of planar graphs*, Tech. Report 10/1986, Universität des Saarlandes, November 1986.
[18] D. KIRKPATRICK, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.
[19] R. J. LIPTON AND R. E. MILLER, *A batching method for coloring planar graphs*, Inform. Process. Lett., 7 (1978), pp. 185–188.
[20] D. W. MATULA, Y. SHILOACH, AND R. E. TARJAN, *Two linear-time algorithms for five-coloring a planar graph*, Tech. Report STAN-CS-80-830, Stanford University, November 1980.
[21] J. NAOR, *A fast parallel coloring of planar graphs with five colors*, Inform. Process. Lett., 25 (1987), pp. 51–53.
[22] M. H. WILLIAMS, *A linear algorithm for colouring planar graphs with five colours*, Comput. J., 28 (1985), pp. 78–81.

SIAM J. COMPUT
Vol. 18, No. 2, pp. 301-313, April 1989

009

NONBLOCKING MULTIRATE NETWORKS*

RICCARDO MELEN† AND JONATHAN S. TURNER‡

Abstract. An extension of the classical theory of connection networks is defined and studied. This extension models systems in which multiple connections of differing data rates share the links within a network. Conditions under which the Clos and Cantor networks are strictly nonblocking for multirate traffic are determined. The authors also determine conditions under which the Beneš network and variants of the Cantor and Clos networks are rearrangeable. It is found that strictly nonblocking operation can be obtained for multirate traffic with essentially the same complexity as in the classical context.

Key words. nonblocking networks, rearrangeable networks, multirate networks, fast packet networks

AMS(MOS) subject classification. 94A99

1. Introduction. In this paper we introduce a generalization of the classical theory of nonblocking switching networks to model communication systems designed to carry connections with a multiplicity of data rates. The theory of nonblocking networks was motivated by the problem of designing telephone switching systems capable of connecting any pair of idle terminals, under arbitrary traffic conditions. From the start, it was recognized that crossbar switches with N terminals and N^2 crosspoints could achieve nonblocking behavior, only at a prohibitive cost in large systems. In 1953, Clos [6] published a seminal paper giving constructions for a class of nonblocking networks with far fewer crosspoints, providing much of the initial impetus for the theory that has since been developed by Beneš [2], [3], Pippenger [16] and many others [1], [5], [8], [11], [12], [13], [14].

The original theory was developed to model electromechanical switching systems in which both the external links connecting switches and the internal links within them were at any one time dedicated to a single telephone conversation. During the 1960s and 1970s technological advances led to digital switching systems in which information was carried in a multiplexed format, with many conversations time-sharing a single link. While this was a major technological change, its impact on the theory of nonblocking networks was slight because the new systems could be readily cast in the existing model. The primary impact was that the traditional complexity measure of crosspoint count had a less direct relation to cost than in the older technology.

During the last 10 years, there has been a growing interest in communication systems that are capable of serving applications with widely varying characteristics. In particular, such systems are being designed to support connections with arbitrary data rates, over a range from a few bits per second to hundreds of megabits per second [7], [10], [19]. These systems also carry information in multiplexed format but, in contrast to earlier systems, each connection can consume an arbitrary fraction of the bandwidth of the link carrying it. Typically, the information is carried in the form of independent blocks, called *packets*, which contain control information identifying to which of many connections sharing a given link the packet belongs. One way to operate

* Received by the editors February 1988; accepted for publication June 21, 1988.

† Centro Stude Laboratori Telecommunicazioni (CSELT), Torino, Italy. The work of this author was supported in part by the Associazione Elettrotecnica ed Elettronica Italiana, Milano, Italy. This work was done while this author was on leave at Washington University, St. Louis, Missouri 63130.

‡ Computer Science Department, Washington University, St. Louis, Missouri 63130. The work of this author was supported by National Science Foundation grant DCI-8600947 as well as by Bell Communications Research, Bell Northern Research, Italtel SIT, and NEC.

such systems is to select for each connection a path through the switching system to be used by all packets belonging to that connection. When selecting a path it is important to ensure that the available bandwidth on all selected links is sufficient to carry the connection. This leads to a natural generalization of the classical theory of nonblocking networks, which we explore in this paper. Note that such networks can also be operated with packets from a given connection taking different paths; reference [20] analyzes the worst-case loading in networks operated in this fashion. The drawback of this approach is that it makes it possible for packets in a given connection to pass one another, causing them to arrive at their destination out of sequence.

In § 2, we define our model of nonblocking multirate networks in detail. Section 3 contains results on strictly nonblocking networks, in particular showing the conditions that must be placed on the networks of Clos and Cantor in order to obtain nonblocking operation in the presence of multirate traffic. We also describe two variants on the Clos and Cantor network that are wise-sense nonblocking in the general environment. Section 4 gives results on rearrangeably nonblocking networks, in particular, deriving conditions for which the networks of Beneš and Cantor are rearrangeable.

2. Preliminaries. We start with some definitions. We define a network as a directed graph $G=(V, E)$ with a set of distinguished 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. We consider only networks that can be divided into a sequence of *stages*. We say that the input nodes are in stage 0 and for $i>0$, a node v is in stage i if for all edges (u, v), u is in stage $i-1$. An edge (u, v) is said to be in stage i if u is in stage i. In the networks that we consider, all output nodes are in the same stage, and no other nodes are in this stage. When we refer to a k-stage network, we generally neglect the stages containing the input and output nodes. We refer to a network with n input nodes and m output nodes as an (n, m)-network. We let $X_{n,m}$ denote the network consisting of n input nodes, m output nodes, and a single internal node. In this network model, nodes correspond to the hardware devices that perform the actual switching functions and the edges to the interconnecting data paths. This differs from the graph model traditionally used in the theory of switching networks, which can be viewed as a dual to our model.

When describing particular networks we will find it convenient to use a product operation. We denote the product of two networks Y_1 and Y_2 by $Y_1 \times Y_2$. The product operation yields a new network consisting of one or more copies of Y_1 connected to one or more copies of Y_2, with an edge joining each pair of subnetworks. More precisely, if Y_1 has n_1 outputs and Y_2 has n_2 inputs, then $Y_1 \times Y_2$ is formed by taking n_2 copies of Y_1 numbered from 0 to n_2-1, followed by n_1 copies of Y_2 numbered from 0 to n_1-1. Then, for $0 \leqq i \leqq n_1-1$, $0 \leqq j \leqq n_2-1$, we join $Y_1(i)$ to $Y_2(j)$ using an edge connecting output port j of $Y_1(i)$ to input port i of $Y_2(j)$. Next, we remove the former input and output nodes that are now internal, identifying the edges incident to them, and finally, we renumber the input and output nodes of the network as follows: if u was input port i of $Y_1(j)$, it becomes input jn_1+i in the new network; similarly if v was output port i of $Y_2(j)$, it becomes output jn_2+i. We also allow the product of more than two networks, which we denote with the symbol $\bowtie$; the product $Y_1 \bowtie Y_2 \bowtie Y_3$ is obtained by letting $Z_1 = Y_1 \times Y_2$ and $Z_2 = Y_2 \times Y_3$, then identifying the copies of Y_2 in Z_1 and Z_2. This requires of course that the number of copies of Y_2 generated by the two initial products be the same.

A *connection* in a network is a triple (x, y, ω), where $x \in I$, $y \in 0$, and $0 \leqq \omega \leqq 1$. We refer to ω as the *weight* of the connection and it 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 $\leqq 1$. A *configuration* for a network G is a set of routes. The weight on an edge in a particular configuration is just the sum of the weights of all routes including that edge. A configuration is compatible if for all edges $(u, v) \in E$, the weight on (u, v) is $\leqq 1$. 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 *rearrangeably nonblocking* (or simply *rearrangeable*) if for every set C of compatible connections, there exists a compatible configuration that realizes C. A network is *strictly nonblocking* if for every 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. For strictly nonblocking networks, one can choose routes arbitrarily and always be guaranteed that any new connections can be satisfied without rearrangements. We say that a network is *wide-sense nonblocking* if there exists a routing algorithm for which the network never blocks; that is, for an arbitrary sequence of connection and disconnection requests, we can avoid blocking if routes are selected using the appropriate routing algorithm, and disconnection requests are performed by simply deleting the route.

Sometimes, improved performance can be obtained by placing constraints on the traffic imposed on a network. We will consider two such constraints. First, we restrict the weights of connections to the interval $[b, B]$. We also limit the sum of the weights of connections involving a node x in $I \cup O$ to β. Note that $0 \leqq b \leqq B \leqq \beta \leqq 1$. We say a network is strictly nonblocking for particular values of b, B, and β if for all sets of connections for which the connection weights are in $[b, B]$ and the total port weight is β, the network cannot block. The definitions of rearrangeably nonblocking and wide-sense nonblocking networks are extended similarly. The practical effect of a restriction on β is to require that a network's internal data paths operate at a higher speed than the external transmission facilities connecting switching systems, a common technique in the design of high-speed systems. The reciprocal of β is commonly referred to as the *speed advantage* for a system.

Two particular choices of parameters are of special interest. We refer to the traffic condition characterized by $B=\beta$, $b=0$ as unrestricted packet switching (UPS), and the condition $B=b=\beta=1$ as pure circuit switching (CS). Since the CS case is a special case of the multirate case, we can expect solutions to the general problem to be at least as costly as the CS case and that theorems for the general case should include known results for the CS case.

3. Strictly nonblocking networks. A three-stage Clos [6] network with N input and output nodes is denoted by $C_{N,k,m}$, where k and m are parameters, and is defined as: $C_{N,k,m} = X_{k,m} \bowtie X_{N/k,N/k} \bowtie X_{m,k}$. A Clos network is depicted in Fig. 1. The standard reasoning to determine the nonblocking condition (see [6]) can be extended in a straightforward manner, yielding the following theorem.

THEOREM 3.1. *The Clos network $C_{N,k,m}$ is strictly nonblocking if*

$$m > 2 \max_{b \leqq \omega \leqq B} \left\lfloor \frac{\beta k - \omega}{s(\omega)} \right\rfloor,$$

where $s(\omega) = \max\{1-\omega, b\}$.

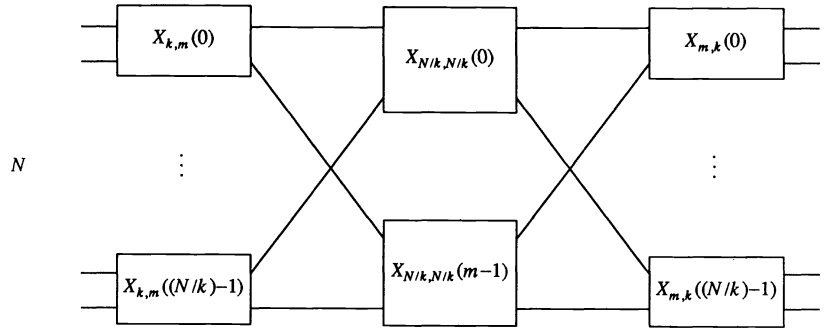


FIG. 1. *Clos network.*

Proof. Suppose we wish to add a connection (x, y, γ) to an arbitrary configuration C. Let u be the stage-1 node adjacent to x and note that the sum of the weights on all edges out of u is at most $\beta(k-1)+(\beta-\gamma)=\beta k-\gamma$. Consequently, the number of edges out of u that carry a weight of more than $(1-\gamma)$ is $\leqq \lfloor(\beta k-\gamma)/s(\gamma)\rfloor$, and hence the number of inaccessible middle stage nodes is

$$\leqq \left\lfloor \frac{\beta k-\gamma}{s(\gamma)} \right\rfloor \leqq \max_{b\leqq\omega\leqq B} \left\lfloor \frac{\beta k-\omega}{s(\omega)} \right\rfloor < m/2.$$

That is, less than half the middle stage nodes are inaccessible from x. By a similar argument, less than half the middle stage nodes are inaccessible from y, implying that there is at least one middle stage node accessible to both. □

Let us examine some special cases of interest. If we let $b=B=\beta=1$, the effect is to operate the network in CS mode. The theorem states that we get nonblocking operation when $m\geqq 2k-1$, as is well known. In the UPS case, the condition on m becomes $m>2(\beta/(1-\beta))(k-1)$. So $m=2k-1$ is sufficient here also if $\beta=\frac{1}{2}$.

Using Theorem 3.1, we can construct a wide-sense nonblocking network for unrestricted traffic by placing two Clos networks in parallel and segregating connections in the two networks based on weight. In particular if we let $m=4k-1$, the network $X_{1,2}\bowtie C_{N,k,m}\bowtie X_{2,1}$ is wide-sense nonblocking if all connections with weight $\leqq\frac{1}{2}$ are routed through one of the Clos subnetworks, and all the connections with weight $>\frac{1}{2}$ are routed through the other.

A k-ary Beneš network [2], built from $k\times k$ switching elements (where $\log_k N$ is an integer), can be defined recursively as follows: $B_{k,k}=X_{k,k}$ and $B_{N,k}=X_{k,k}\bowtie B_{N/k,k}\bowtie X_{k,k}$ (see Fig. 2). A k-ary Cantor network of multiplicity m is defined

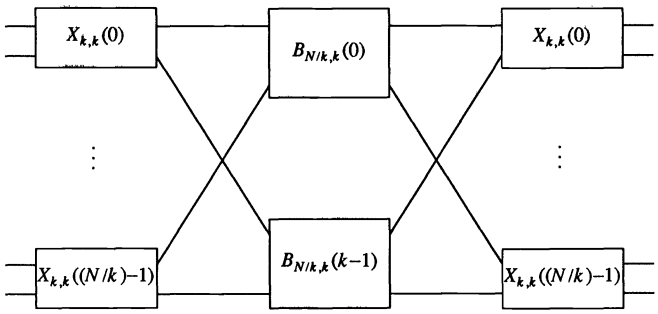


FIG. 2. *Beneš network* $(B_{N,k})$.

as $K_{N,k,m} = X_{1,m} \bowtie B_{N,k} \bowtie X_{m,1}$. Note that this definition is expressed differently from those given in [5], [13], but we find it preferable as it shows clearly the close relationship between these two structures. Figure 3 depicts a binary Cantor network of multiplicity three with one of its Beneš subnetworks highlighted. The next theorem captures the condition on m required to make the Cantor network strictly nonblocking.

THEOREM 3.2. *The Cantor network $K_{N,k,m}$ is strictly nonblocking if*

$$m \geqq \frac{2\beta}{ks(B)}(1+(k-1)\log_k (N/k)).$$

Proof. Suppose we wish to add a connection (x, y, ω) to an arbitrary configuration. Note that there are mN/k nodes in the middle stage of the network. We will show that more than half of these nodes are accessible from x if m satisfies the inequality in the statement of the theorem.

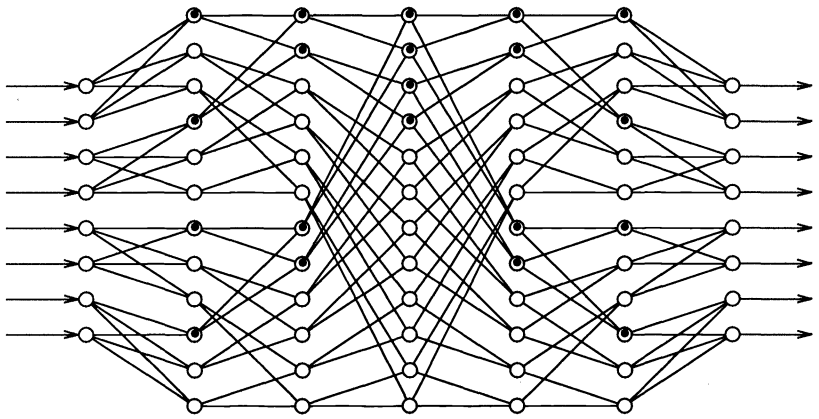


FIG. 3. *Cantor network.*

Define W_i to be the set of all edges (u, v) in stage i, for which u is accessible from x, but v is not. Define λ_i to be the sum of the weights on all edges in W_i, and note that $\lambda_i \geqq |W_i| s(\omega)$. If we let $h = \log_k N$, then the number of middle stage nodes (stage $h+1$) that are not accessible from x is given by

$$\sum_{i=2}^{h} k^{h-i}|W_i| \leqq \frac{1}{s(\omega)} \sum_{i=2}^{h} k^{h-i}\lambda_i.$$

It is easily verified that

$$\sum_{i=2}^{h} k^{h-i}\lambda_i \leqq (\beta-\omega)k^{h-2} + \sum_{i=2}^{h} k^{h-i}(k^{i-1}-k^{i-2})\beta.$$

To see this, note that each term in the summation on the left gives the weight used for blocking at stage i weighted by k^{h-i}. The terms in the summation on the right give an upper bound on the total weight of the traffic that could possibly block connections from x at stage i, similarly weighted by k^{h-i}. The initial term on the right corresponds to the weight from input port x that is available for blocking. The right side of the above inequality equals

$$\left(\frac{\beta-\omega}{k}\right)\left(\frac{N}{k}\right) + \beta\left(\frac{k-1}{k}\right)\left(\frac{N}{k}\right)\log_k(N/k) < \frac{\beta N}{k^2}(1+(k-1)\log_k (N/k)).$$

Combining this with the first inequality above, the number of inaccessible middle stage nodes is strictly less than

$$\frac{\beta N}{s(B)k^2}(1+(k-1)\log_k(N/k)) \leqq (mN/2k).$$

That is, fewer than half the middle stage nodes are inaccessible from x. By a similar argument, fewer than half of the middle stage nodes are inaccessible from y, meaning that there exists an available route from x to y. □

COROLLARY 3.1. *The Beneš network $B_{N,k}$ is strictly nonblocking if*

$$\beta \leqq \left[\frac{2}{ks(B)}(1+(k-1)\log_k(N/k))\right]^{-1}.$$

Proof. Substitute 1 for m in the statement of the theorem and solve for β. □

When we apply the theorem to the CS case for $k=2$, we find that the condition on m reduces to $m \geqq \log_2 N$, as is well known. For the UPS case with $k=2$, we have $m \geqq (\beta/(1-\beta))\log_2 N$; that is, we again need a speed advantage of two to match the value of m needed in the CS case.

We can construct wide-sense nonblocking networks for $\beta = 1$ by increasing m. We divide the connections into two subsets, with all connections of weight $\leqq \frac{1}{2}$ segregated from those with weight $>\frac{1}{2}$. Applying Theorem 3.2 we find that $m \geqq 4((k-1)/k)\log_k N$ is sufficient to carry each portion of the traffic, giving a total of $8((k-1)/k)\log_k N$ subnetworks.

4. Rearrangeably nonblocking networks. As mentioned earlier, a k-ary Beneš network [2], can be defined recursively as follows: $B_{k,k} = X_{k,k}$ and $B_{N,k} = X_{k,k} \bowtie B_{N/k,k} \bowtie X_{k,k}$. The Beneš network is rearrangeable in the CS case [2], and efficient algorithms exist to reconfigure it [12], [14]. In this section, we show that under certain conditions, the Beneš network can be rearrangeable for multirate traffic as well. We start by reviewing a proof of rearrangeability for the CS case, as we will be extending the technique for this case to the general environment.

Consider a set of connections $C = \{c_1, \cdots, c_r\}$ for $B_{N,k}$, where $c_i = \{x_i, y_i, 1\}$. There is at most one connection for each input and output port. The recursive structure of the network allows us to decompose the routing problem into a set of subproblems, corresponding to each of the stages in the recursion. The top level problem consists of selecting, for each connection, one of the k subnetworks $B_{N/k,k}$ through which to route. Given a solution to the top level problem, we can solve the routing problems for the k subnetworks independently. We can solve the top level problem most readily by reformulating it as a graph coloring problem. To do this, we define the connection graph $G_C = (V_C, E_C)$ for C as follows:

$$V_C = \{u_j, v_j | 0 \leqq j < N/k\},$$

$$E_C = \{\{u_{\lfloor x_i/k \rfloor}, v_{\lfloor y_i/k \rfloor}\} | 1 \leqq i \leqq r\}.$$

To solve the top level routing problem, we color the edges of G_C with colors $\{0, \cdots, k-1\}$ so that no two edges with a common endpoint share the same color. The colors assigned to the edges correspond to the subnetwork through which the connection must be routed. Because G_C is a bipartite multigraph with maximum vertex degree k, it is always possible to find an appropriate coloring [4], [9]. In brief, given a partial coloring of G_C, we can color an uncolored edge $\{u, v\}$ as follows. If there is a color $i \in \{0, \cdots, k-1\}$ that is not already in use at both u and v, we use it. Otherwise, we let i be any unused color at u and j be any unused color at v. We then find a maximal

alternating path from v, that is, a longest path with edges colored i or j, which has v as one of its endpoints. Because the graph is bipartite, the alternating path must end at some vertex other than u or v. Then, we interchange the colors i and j for all edges on the path and use i to color the edge $\{u, v\}$.

To prove results for rearrangeability in the presence of multirate traffic, we must generalize the graph coloring methods used in the CS case. We define a connection graph G_C for a set of connections C as previously, with the addition that each edge is assigned a weight equal to that of the corresponding connection. We say that a connection graph is (β, k)-*permissible* if the edges incident to each vertex can be partitioned into k groups whose weights sum to no more than β. A *legal* (β, m)-*coloring* of a connection graph is an assignment of colors in $\{0, \cdots, m-1\}$ to each edge so that at each vertex u, the sum of the weights of the edges of any given color is no more than β.

Now, suppose we let $Y = Y_1 \bowtie Y_2 \bowtie Y_3$, where Y_1 is a (k, m)-network, Y_2 is an $(N/k, N/k)$-network, and Y_3 is an (m, k)-network, and also let $0 \leqq \beta_1 \leqq \beta_2 \leqq 1$. Then, if Y_1, Y_2, Y_3 are rearrangeable for connection sets with $\beta \leqq \beta_2$, and every (β_1, k)-permissible connection graph for Y has a legal (β_2, m) coloring, then Y is rearrangeable for connection sets with $\beta \leqq \beta_1$.

Our first use of the coloring method is in the analysis of $B_{N,k}$. We apply it in a recursive fashion. At each stage of the recursion, the value of β may be slightly larger than at the preceding stage. The key to limiting the growth of β is the algorithm used for coloring the edges of the connection graph at each stage. We describe that algorithm next.

Let $G_C = (V_C, E_C)$ be an arbitrary connection graph. For each vertex u, let C_u be the set of edges involving u. Next, number the edges in C_u from zero, in nonincreasing order of their weight, and let $C_u^i \subseteq C_u$ comprise the edges with indices in the range $\{ik, \cdots, (i+1)k-1\}$ for $i \geqq 0$. Our coloring algorithm assigns unique colors to edges in each subset C_u^i. In particular, given a partial coloring of G_C, we color an uncolored edge $\{u, v\}$ belonging to C_u^i and C_v^j as follows. If there is a color $a \in \{0, \cdots, k-1\}$ that is not already in use within C_u^i and C_v^j, we use it. Otherwise, we let a_1 be any used color within C_u^i and a_2 be any unused color within C_v^j. We then find a maximal *constrained alternating path* from v, that is, a longest path with edges colored a_1 or a_2 with v as one of its endpoints and such that for every interior vertex w on the path, the path edges incident to w belong to a common set C_w^h. Because the graph is bipartite, the last edge cannot be a member of either C_u^i or C_v^j. Given the path, we interchange the colors a_1 and a_2 for all edges on the path, and use a_1 to color the edge $\{u, v\}$. We refer to this as the CAP (constrained alternating path) algorithm. We can route a set of connections through $B_{N,k}$ by applying CAP recursively. Our first theorem gives conditions under which this routing is guaranteed not to exceed the capacity of any edge in the network.

THEOREM 4.1. *The* CAP *algorithm successfully routes all sets of connections for* $B_{N,k}$ *for which*

$$\beta \leqq \left[1 + \frac{k-1}{k}(B/\beta)\log_k (N/k)\right]^{-1}.$$

Proof. Let G_C be any (β_1, k)-permissible connection graph with maximum edge weight B, and $\beta_1 \leqq 1 - B(k-1)/k$. We start by showing that the CAP algorithm produces a legal (β_2, k)-coloring for some $\beta_2 \leqq \beta_1 + B(k-1)/k$.

Let u be any vertex in G_C. Since each color is used at most once for each subset C_u^i of the edges at u, the largest weight that can be associated with any one color at

u is bounded by the sum of the weights of the heaviest edges in C_u^i for all i. Because the edges were assigned to the C_u^i in nonincreasing order of weight, the total weight of like-colored edges at u is at most $B+(k\beta_1-B)/k=\beta_1-B(k-1)/k$.

Given this, if we route a set of connections through $B_{N,k}$ by recursive application of the CAP algorithm, we will succeed if

$$\beta+\left(\frac{k-1}{k}\right)B\log_k(N/k)\leqq 1,$$

or equivalently, $\beta\leqq[1+((k-1)/k)(B/\beta)\log_k(N/k)]^{-1}$. □

As an example, if $N=2^{16}$, $k=4$, and $B=\beta$, it suffices to have $\beta\leqq 0.16$. We can improve on this result by modifying the CAP algorithm. Because the basic algorithm treats each stage in the recursion completely independently, it can in the worst-case concentrate traffic unnecessarily. The algorithm we consider next attempts to balance the traffic between subnetworks when constructing a coloring. We describe the algorithm only for the case of $k=2$, although extension to higher values is possible.

Let G_C be a connection graph for $B_{N,2}$. G_C comprises vertices $u_0,\cdots,u_{(N/2)-1}$ corresponding to nodes in stage one of $B_{N,2}$ and vertices $v_0,\cdots,v_{(N/2)-1}$ corresponding to nodes in stage $2(\log_2 N-1)$. We have an edge from u_1 to v_j corresponding to each connection to be routed between the corresponding nodes of $B_{N,2}$. We note that for $0\leqq i<N/4$, the nodes corresponding to u_{2i} and u_{2i+1} have the same successors in stage two of $B_{N,2}$. Similarly, the nodes in $B_{N,2}$ corresponding to v_{2i} and v_{2i+1} have common predecessors. We say such vertex pairs are *related.*

Let a and b be any pair of related vertices in G_C. The idea behind the modified coloring algorithm is to balance the coloring at a and b so that the total weight associated with each color is more balanced, thus limiting the concentration of traffic in one subnetwork. The technique used to balance the coloring is to constrain it so that when appropriate, the edges of largest weight at a and b are assigned different colors; hence the corresponding connections are routed through distinct subnetworks. For any vertex v in G_C, let $\omega_0(v)\geqq\omega_1(v)\geqq\cdots$ be the weights of the edges defined at v, let $W_0(v)=\sum_{i\geqq 0}\omega_{2i}$, $W_1(v)=\sum_{i\geqq 0}\omega_{2i+1}$, and $W(v)=W_0(v)$. Also, let $x(v)=W_0(v)-W_1(v)$.

The *modified* CAP *algorithm* proceeds as follows. For each pair of related vertices a and b in G_C, if $x(a)+x(b)>B$, add a dummy node z to G_C with edges of weight two connecting it to a and b. We then color this modified graph as in the original CAP algorithm, and on completion we simply ignore the added nodes and edges. The effect of adding the dummy node is to constrain the coloring at a and b so that the edges of maximum weight are assigned distinct colors. We apply this procedure recursively except that in the last step of the recursion we use the original CAP algorithm.

THEOREM 4.2. *The modified* CAP *algorithm successfully routes all sets of connections for* $B_{N,2}$ *for which*

$$\beta\leqq[1+\tfrac{1}{4}(B/\beta)\log_2 N]^{-1}.$$

Proof. Let a and b be related vertices with $\omega_0(a)\geqq\omega_0(b)$. Let $z_1=\max\{W(a),W(b)\}$ and let z_2 be the total weight on edges colored 0 at a and b. If $x(a)+x(b)\leqq B$, no dummy vertex is added and we have that

$$z_2\leqq W_0(a)+W_0(b)\leqq(z_1+x(a))/2+(z_1+x(b))/2\leqq z_1+B/2.$$

Similarly, if $x(a)+x(b)\geqq B$, a dummy vertex is added and we have that

$$z_2\leqq\omega_0(a)+W_1(a)+W_1(b)\leqq\omega_0(a)+(z_1-x(a))/2+(z_1-x(b))/2\leqq z_1+B/2.$$

Thus, the total weight on a node in stage i is at most $2\beta+(i-1)B/2$. In particular, this holds for $i=\log_2 N-1$. Also note that for a link (u, v) in stage $j\leqq\log_2 N-2$, the maximum weight is at most B plus half the weight on u. For a link (u, v) in stage $\log_2 N-1$, the weight is at most $B/2$ plus the maximum weight at u, since in this last step the original CAP algorithm was used. Consequently, no link carries a weight greater than $\beta+(B/4)\log_2 N$. □

Theorem 4.2 implies for example that if $\beta=B=0.2$, a binary Beneš network with 2^{16} input and output nodes is rearrangeable. Theorem 4.1, on the other hand gives rearrangeability in this case only if β is limited to about 0.118. It turns out that we can obtain a still stronger result by exploiting some additional properties of the original CAP algorithm.

THEOREM 4.3. *The* CAP *algorithm successfully routes all sets of connections for* $B_{N,k}$ *for which*

$$\beta\leqq[\max\{2, \lambda-\ln\lfloor\beta/B\rfloor\}]^{-1},$$

where $\lambda=2+\ln\log_k(N/k)$.

So, for example, if $k=4$, $N=2^{16}$, and $\beta/B=2$, we can have $\beta=0.3$. The proof of Theorem 4.3 requires the following lemmas.

LEMMA 4.1. *Let* r *be any positive integer. If a set of connections for* $B_{N,k}$ *is routed by repeated applications of the* CAP *algorithm, no link will carry more than* r *connections of weight* $>\beta/(r+1)$.

Proof. (By induction.) The condition is true by definition for the external links. If the assertion holds at a given level of recursion, the connection graph for the next stage will have at most rk edges of weight greater than $\beta/(r+1)$ at any given node u. These edges are all contained in $C_u^0\cup\cdots\cup C_u^{r-1}$, implying that the CAP algorithm will use a single color for at most r of them. □

If l is a link in $B_{N,k}$, we define S_l^j to be the set of links l' in stage j for which there is a path from l' to l. If a given set of connections uses a link l, we refer to one connection of maximum weight as the *primary connection* on l and all others as *secondary connections.* We note that if the CAP algorithm is used to route a set of connections through $B_{N,k}$, then if there are $r+1$ connections of weight $\geqq\omega$ on a link $l=(u, v)$, there are at least $1+kr$ connections of weight $\geqq\omega$ on the links entering u.

LEMMA 4.2. Let $0\leqq i\leqq\log_k(N/k)$, *let* l *be a stage* i *link in* $B_{N,k}$ *carrying connections routed by the* CAP *algorithm, and let the connection weights be* $\omega_0\geqq\omega_1\geqq\cdots\geqq\omega_h$. *For* $0\leqq t\leqq h$ *and* $0\leqq s\leqq\min\{i, t\}$, *there are at least* $(t-s+1)k^s+sk^{s-1}$ *connections of weight* $\geqq\omega_t$ *on the links in* S_l^{i-s}.

Proof. The proof is by induction on s. When $s=0$, the lemma asserts that there are $t+1$ connections of weight $\geqq\omega_t$ which is trivially true. Assume then that the lemma holds for $s-1$; that is, there exist $(t-s+2)k^{s-1}+(s-1)k^{s-2}$ connections of weight $\geqq\omega_t$ on the links in S_l^{i-s+1}. Because $|S_l^{i-s+1}|=k^{s-1}$, by the pigeon-hole principle, at least $(t-s+1)k^{s-1}+(s-1)k^{s-2}$ of these are secondary connections. This implies that there at least

$$k^{s-1}+k[(t-s+1)k^{s-1}+(s-1)k^{s-2}]=(t-s+1)k^s+sk^{s-1}$$

connections of weight $\geqq\omega_t$ in S_l^{i-s}. □

Proof of Theorem 4.3. Consider an arbitrary set of connections for $B_{N,k}$ satisying the bound on β given in the theorem, and assume that the CAP algorithm is used to route the connections. Let l be any link in stage i, where $i\leqq\log_k(N/k)$, and let the weights of the connections on l be $\omega_0\geqq\cdots\geqq\omega_h$. Let r be the positive integer defined by $\beta/(r+1)<B\leqq\beta/r$ (equivalently, $r=\lfloor\beta/B\rfloor$). By Lemma 4.2, S_l^0 carries connections

with a total weight of at least

$$\omega_0 + k\omega_1 + k^2\omega_2 + \cdots + k^{i-1}\omega_{i-1} + k^i(\omega_i + \cdots + \omega_h).$$

Since the total weight on S_l^0 is at most βk^i, we have

$$\beta k^i \geqq \sum_{j=0}^{i-1} k^j \omega_j + k^i \sum_{j=i}^{h} \omega_j.$$

From this and Lemma 4.1, we have that

$$\sum_{j=0}^{r-1} \omega_j + \sum_{j=r}^{i-1} \omega_j + \sum_{j=i}^{h} \omega_j \leqq Br + \beta \sum_{j=r}^{i-1} \frac{1}{j+1} + \beta \leqq 2\beta + \beta \sum_{j=r+1}^{\log_k (N/k)} \frac{1}{j}.$$

If $\lfloor \beta/B \rfloor \geqq \log_k (N/k)$, the summation vanishes and we have that the weight on l is $\leqq 2\beta$. Otherwise, the weight is bounded by

$$\leqq \beta(2 + \ln \log_k (N/k)/r) = \beta(\lambda - \ln \lfloor \beta/B \rfloor).$$

So, if β satisfies the bound in the statement of the theorem, the weight on l is no more than one. By a similar argument, the weight on any link in stage j for $j \geqq \log_k N$ is at most one. □

We now turn our attention to the Cantor network and give conditions for rearrangeability in that case.

THEOREM 4.4. *Let $\varepsilon > 0$ and $\lfloor \beta/B \rfloor \leqq \log_k (N/k)$. $K_{N,k,m}$ is rearrangeable if*

$$m \geqq \lceil (1+\varepsilon)(\lambda - \ln \lfloor \beta/B \rfloor) \rceil + 2(2 + \log_2 \lambda + \log_2 (B/c)),$$

where $\lambda = 2 + \ln \log_k (N/k)$ and $c = 1 - \beta\lambda/(1+\varepsilon)(\lambda - \ln \lfloor \beta/B \rfloor)$.

The proof of Theorem 4.4 requires several lemmas.

LEMMA 4.3. *Let α, r be $\geqq 1$ with r and αr integers. $B_{N,k}$ is rearrangeable for sets of connections with weights ω that satisfy $\beta/(\alpha r + 1) < \omega \leqq \beta/r$ and $\alpha \leqq e^{(1/\beta)-1}$.*

Proof. By Lemma 4.2, if $B_{N,k}$ is routed using the CAP algorithm, no link contains more than αr connections. The sum of the weights of the connections on any given link is

$$\leqq r\left(\frac{\beta}{r}\right) + \frac{\beta}{r+1} + \cdots + \frac{\beta}{\alpha r} = \beta + \beta \sum_{i=r+1}^{\alpha r} 1/i \leqq \beta(1 + \ln \alpha) \leqq 1. \qquad \square$$

LEMMA 4.4. *Let α, r be $\geqq 1$ with r and αr integers. $K_{N,k,m}$ is rearrangeable for sets of connections with weights ω that satisfy*

$$\beta/(\alpha r + 1) < \omega \leqq \beta/r \quad \text{and} \quad \alpha \leqq \exp\left[\frac{rm}{\beta(r+m-1)} - 1\right].$$

Proof. The connections can be distributed among the m Beneš subnetworks using the CAP algorithm; the resulting maximum port weight on the subnetworks is

$$\beta' \leqq \frac{\beta}{m} + \frac{(m-1)\beta}{mr} = \frac{\beta(r+m-1)}{mr}.$$

By Lemma 4.3, each subnetwork can be successfully routed if

$$\alpha \leqq \exp\left[\frac{rm}{\beta(r+m-1)} - 1\right] \leqq e^{(1/\beta')-1}. \qquad \square$$

LEMMA 4.5. *$K_{N,k,m}$ is rearrangeable if $m \geqq 2(2 + \log_2 (B/b))$.*

Proof. Define h, i by letting

$$\frac{\beta}{2^{h+1}} < B \leqq \frac{\beta}{2^h} \quad \text{and} \quad \frac{\beta}{2^{i+1}} < b \leqq \frac{\beta}{2^i}.$$

By Lemma 4.4, two of the Beneš subnetworks are sufficient to route connections with weights in the interval $(2^{-(j+1)}\beta, 2^{-j}\beta]$ for any $j \geqq 0$. For $h \leqq j \leqq i$ then, we devote two subnetworks for the connections with weights in $(2^{-(j+1)}\beta, 2^{-j}\beta]$. The total number of subnetworks required is at most

$$2(i-h+1) \leqq 2(2+\log_2(B/b)) \leqq m. \qquad \square$$

LEMMA 4.6. *$K_{N,k,m}$ is rearrangeable if*

$$\frac{(\beta - B)\lambda}{1 - B\lambda} \leqq m \leqq \frac{\beta}{B},$$

where $\lambda = 2 + \ln \log_k (N/k)$.

Proof. We distribute the traffic among the m Beneš subnetworks using the CAP algorithm. The resulting maximum port weight is β', where

$$\beta' \leqq \frac{\beta}{m} + \frac{m-1}{m} B = B + \frac{\beta - B}{m}.$$

By Theorem 4.3, the maximum weight on any link is at most

$$\beta'(\lambda - \ln \lfloor \beta'/B \rfloor) \leqq B\lambda + \frac{(\beta - B)\lambda}{m} \leqq 1. \qquad \square$$

Proof of Theorem 4.4. Let $B' = c/\lambda$. By Lemma 4.5, all the traffic with weight $> B'$ can be handled using

$$2(2 + \log_2 \lambda + \log_2 B/c)$$

of the Beneš subnetworks. By Lemma 4.6, the remaining traffic can be carried using

$$\frac{(\beta - B')\lambda}{1 - B'\lambda} = \frac{\beta\lambda - c}{1 - c} \leqq \lceil (1+\varepsilon)(\lambda - \ln \lfloor \beta/B \rfloor) \rceil$$

subnetworks. $\square$

Theorem 4.4 holds when $\lfloor \beta/B \rfloor \leqq \log_k (N/k)$. When this condition does not hold, $K_{N,k,m}$ is rearrangeable with m between one and three, depending on the value of β. In particular, if $\beta \leqq \frac{1}{2}$, $m = 1$ is sufficient using Theorem 4.1. If $(\frac{1}{2}) < \beta \leqq 1 - 1(2 \log_k (N/k))$, $m = 2$ is sufficient since in this case the traffic can be split among the two subnetworks so that each experience a maximum port weight of at most $\frac{1}{2}$.

The graph coloring methods used to route connections for $B_{N,k}$ can also be applied to networks that "expand" at each level of recursion. Let $C^*_{k,k,m} = X_{k,k}$ and for $N = k^i$, $i > 1$, let $C^*_{N,k,m} = X_{k,m} \bowtie C^*_{N/k,N/k} \bowtie X_{m,k}$. The following theorem gives conditions under which $C^*_{N,k,m}$ is rearrangeable.

THEOREM 4.5. *$C^*_{N,k,m}$ is rearrangeable if*

$$\beta \leqq \left[1/\gamma^c + \frac{m-1}{m} \frac{B}{\beta} \frac{1 - 1/\gamma^c}{1 - 1/\gamma} \right]^{-1},$$

where $\gamma = m/k$ *and* $c = \log_k (N/k)$.

Proof. We use the CAP algorithm to route the connections. If we let β_i be the largest resulting weight on a link in stage i for $1 \leqq i \leqq \log_k (N/k)$, we have

$$\beta_i \leqq B + \frac{k\beta_{i-1} - B}{m} = (\beta_{i-1}/\gamma) + \frac{m-1}{m} B \leqq (\beta_0/\gamma^i) + \frac{m-1}{m} B \frac{1-(1/\gamma)^i}{1-1/\gamma} \leqq 1. \qquad \square$$

So, for example, $C^*_{N,k,2k-1}$ is rearrangeable, if $B \leqq \frac{1}{2}$.

5. Closing remarks. In recent years, there has been a growing interest in switching systems capable of carrying general multirate traffic, in order to be able to support a wide range of applications including voice, data, and video. A variety of research teams have constructed high-speed switching systems of moderate size [7], [10], [19], [21], but little consideration has yet been given to the problem of constructing very large switching systems using such modules as building blocks. The theory we have developed here is a first step to understanding the blocking behavior of such systems.

In this paper, we have introduced what we feel is an important research topic and have given some fundamental results. There are several directions in which our work may be extended. While we have good constructions for strictly nonblocking networks, we expect that our results for rearrangeably nonblocking networks can be improved. In particular, we suspect that the Beneš network can be operated in a rearrangeable fashion with just a constant speed advantage. Another interesting topic is nonblocking networks for multipoint connections. While this has been considered for space-division networks [1], [8], [11], [17], it has not been studied for networks supporting multirate traffic. Another area to consider is determination of blocking probability for multirate networks. We expect this to be highly dependent on the particular choice of routing algorithm.

REFERENCES

[1] L. A. BASSALYGO, *Asymptotically optimal switching circuits*, Probl. Inform. Transmission, 17 (1981), pp. 206-211.

[2] V. E. BENEŠ, *Mathematical Theory of Connecting Networks and Telephone Traffic*, Academic Press, New York, 1965.

[3] ———, *Blocking states in connecting networks made of square switches arranged in stages*, Bell Systems Tech. J., 60 (1981) pp. 511-521.

[4] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, North-Holland, New York, 1976.

[5] D. G. CANTOR, *On Non-Blocking Switching Networks*, Networks, 1 (1971), pp. 367-377.

[6] C. CLOS, *A study of non-blocking switching networks*, Bell Systems Tech. J., 32 (1953), pp. 406-424.

[7] J. P. COUDREUSE AND M. SERVEL *Prelude: An Asynchronous Time-Division Switched Network, International Communications Conference*, 1987.

[8] P. FELDMAN, J. FRIEDMAN, AND N. PIPPENGER, *Non-Blocking Networks*, Proc. STOC 1986 5/86, pp. 247-254.

[9] HAROLD N. GABOW, *Using Euler partitions to edge color bipartite multigraphs*, Internat. J. Comput. and Information Sciences (1976), pp. 345-355.

[10] A. HUANG AND S. KNAUER, *Starlite: A wideband digital switch*, Proc. Globecom 84, 12 (1984), pp. 121-125.

[11] D. G. KIRKPATRICK, M. KLAWE, AND N. PIPPENGER, *Some Graph-Coloring Theorems with Applications to Generalized Connection Networks*, SIAM J. Algebraic Discrete Methods (1985), pp. 576-582.

[12] K. Y. LEE, *A New Beneš Network Control Algorithm*, IEEE Transactions on Computers, vol. C-36, 6/87, pp. 768-772.

[13] G. M. MASSON, G. C. GINGHER, AND S. NAKAMURA, *A sampler of circuit switching networks*, Computer, 6 (1979), pp. 145-161.

[14] D. C. OPFERMAN AND N. T. TSAO-WU, *On a class of rearrangeable switching networks, Part I: Control algorithm*, Bell Systems Tech. J., 50 (1971), pp. 1579-1600.

[15] J. K. PATEL, *Performance of processor-memory interconnections for multiprocessors*, IEEE Trans. Comput., C-30 (1981), pp. 301–310.

[16] N. PIPPENGER, *Telephone switching networks*, Proc. Symp. Appl. Math., (1982), pp. 101–133.

[17] G. RICHARDS AND F. K. HWANG, *A two stage rearrangeable broadcast switching network*, IEEE Transactions on Communications, 10 (1985), pp. 1025–1035.

[18] C. E. SHANNON, *Memory requirements in a telephone exchange*, Bell Systems. Tech. J., 29 (1950), pp. 343–349.

[19] J. S. TURNER, *Design of a broadcast packet network*, IEEE Transactions on Communications, 6 (1988), pp. 734–743.

[20] ———, *Fluid flow loading analysis of packet switching networks*, Report WUCS-87-16, 7/87, Washington University Computer Science Department, St. Louis, MO.

[21] Y. S. YEH, M. G. HLUCHYJ, AND A. S. ACAMPORA, *The knockout switch: A simple modular architecture for high performance packet switching*, International Switching Symposium, 3 (1987), pp. 801–808.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 314-326, April 1989

010

MINIMIZING SCHEDULE LENGTH SUBJECT TO MINIMUM FLOW TIME*

JOSEPH Y-T. LEUNG† AND GILBERT H. YOUNG†

Abstract. The problem of scheduling n independent tasks on $m \geqq 1$ identical processors, with the objective of minimizing the schedule length under the constraint that it must have minimum flow time, is considered. For nonpreemptive scheduling, it is known that the problem is NP-hard. This paper shows that the problem is a lot easier for preemptive scheduling. Specifically, an $O(n \log n)$ time algorithm to find an optimal preemptive schedule is given. This algorithm generates schedules with at most $m-1$ preemptions. It is also shown that an optimal nonpreemptive schedule can be almost twice as long as an optimal preemptive schedule. The results suggest that preemption is extremely beneficial in simultaneously minimizing the flow time and the schedule length.

Key words. schedule length, flow time, nonpreemptive scheduling, preemptive scheduling, NP-hard

AMS(MOS) subject classifications. 90B35, 68R05

1. Introduction. In deterministic scheduling theory, schedule length and flow time are two commonly studied performance measures. Minimizing the schedule length has the effect of increasing the processor utilization, while minimizing the flow time tends to decrease the average number of unfinished tasks in the system. To obtain good overall performance, a system should aim at optimizing both performance measures. Unfortunately, the two criteria seem to conflict with each other. Scheduling algorithms that do well with respect to one measure tend to do poorly for the other. For example, the LPT rule [9] is known to produce schedules that are reasonably short, yet the schedules produced tend to have large flow time. On the other hand, the SPT rule [5] is known to be an optimal algorithm for minimizing the flow time, but it is usually outperformed by the LPT rule in producing short schedules. With the exception of [4], [7], [8], most of the scheduling algorithms studied in the literature have concentrated on just one of the two criteria. In this paper, we continue the work of [4], [7], [8] by studying the scheduling problem of optimizing both performance measures simultaneously.

Formally, our problem can be stated as follows: We are given a set $TS = \{T_1, T_2, \cdots, T_n\}$ of n independent tasks with execution times $\{p_1, p_2, \cdots, p_n\}$ to be scheduled on $m \geqq 1$ identical processors. If S is a schedule of TS on the m processors, then $f_i(S)$ denotes the finishing time of T_i in S. The schedule length of S, denoted by $SL(S)$, is defined to be $SL(S) = \max_{i=1}^{n} \{f_i(S)\}$, and the flow time of S, denoted by $FT(S)$, is defined to be $FT(S) = \sum_{i=1}^{n} f_i(S)$. A minimum length schedule, S^+, is one such that $SL(S^+) \leqq SL(S)$ for all schedules S, and a minimum flow-time schedule, S^-, is one such that $FT(S^-) \leqq FT(S)$ for all schedules S. Our goal is to find a schedule, S^*, that has the minimum schedule length among all minimum flow-time schedules. That is, S^* is one such that $FT(S^*) = FT(S^-)$ and $SL(S^*) \leqq SL(S^-)$ for all schedules S^-.

The set of tasks can be scheduled nonpreemptively or preemptively on the processors. In nonpreemptive scheduling, a task, once execution has begun, must proceed to completion. In contrast, preemptive scheduling allows a task to be preempted; execution is later resumed, sometimes on a different processor. It is assumed, however, that there is no time loss in preemption. We shall use the symbols S_{NP}^+ and S_P^+ to denote a minimum-length nonpreemptive schedule and a minimum-length preemptive

* Received by the editors July 27, 1987; accepted for publication June 24, 1988. This research was supported in part by Office of Naval Research grant N00014-87-K-0833.

† Computer Science Program, University of Texas at Dallas, Richardson, Texas 75080.

schedule, respectively. Similarly, S_{NP}^{-}, S_{P}^{-}, S_{NP}^{*}, and S_{P}^{*} are defined analogously. It is known that finding a S_{NP}^{+} is strongly NP-hard [6]. This problem, known in the literature as the multiprocessor scheduling problem or the makespan minimization problem, has been studied by many researchers [2], [3], [9], [10], [12], [13], [15]. Due to the intractability of the problem, most research efforts have been directed towards designing and analyzing fast approximation algorithms [2], [9], [10], [12], or studying special cases of the problem which admit polynomial-time solutions [3], [13], [15]. Finding a S_{P}^{+} is a lot easier [14]; it can be done in $O(n)$ time. In fact, the schedule length of S_{P}^{+} is simply given by $SL(S_{P}^{+}) = \max\{\max_{i=1}^{n}\{p_i\}, (1/m)\sum_{i=1}^{n} p_i\}$. Recently, it has been shown [11] that for any given set of independent tasks to be scheduled on $m \geqq 1$ identical processors, we have $SL(S_{NP}^{+})/SL(S_{P}^{+}) \leqq 2m/(m+1)$. Furthermore, the bound is achievable for every $m \geqq 1$. Thus, when m gets large, a minimum-length non-preemptive schedule can be almost twice as long as a minimum-length preemptive schedule. Hence, preemption can reduce the length of a schedule by a significant proportion.

Finding a S_{NP}^{-} can be done in $O(n \log n)$ time. The well-known SPT rule [5] always constructs a S_{NP}^{-} and it works as follows: Add enough zero-execution-time tasks so that n becomes an integer multiple of m. Sort the tasks in nondecreasing order of their execution times. Divide the tasks into $r = n/m$ ranks, with tasks $T_{(j-1)m+1}$, $T_{(j-1)m+2}, \cdots, T_{(j-1)m+m}$ in rank j, $1 \leqq j \leqq r$. Assign the tasks to the processors rank by rank, in order of increasing rank, each task in a given rank being assigned to a different processor. A schedule constructed by the SPT rule will be called an SPT schedule. Since there are $m!$ ways of assigning the tasks in any given rank, there are $(m!)^r$ SPT schedules for a set of rm tasks to be scheduled on m processors. Figure 1 shows two SPT schedules for a particular set of tasks. Observe that all SPT schedules give the same minimum flow time, $FT(S_{NP}^{-}) = \sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m} p_{(j-1)m+k}$. It has been shown [14] that preemption cannot reduce the minimum flow time. Thus, $FT(S_{P}^{-}) = FT(S_{NP}^{-}) = \sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m} p_{(j-1)m+k}$.

The problem of finding an S_{NP}^{*} has been shown to be NP-hard [1]. In [4], several fast approximation algorithms have been proposed and their worst-case performance bounds analyzed. In this paper, we give an $O(n \log n)$ time algorithm to find an S_{P}^{*}. Our algorithm generates schedules with at most $m-1$ preemptions. The set of tasks given in Fig. 2 shows that $SL(S_{NP}^{*})/SL(S_{P}^{*}) = 2m/(m+1)$. From the result in [11], $2m/(m+1)$ is also an upper bound of $SL(S_{NP}^{*})/SL(S_{P}^{*})$ for any given set of tasks. Thus, preemption can be very beneficial in reducing the schedule length of a minimum flow-time schedule.

Gonzalez [7] has given an $O(n \log n + nm)$-time algorithm to find a preemptive schedule with the minimum flow time subject to the constraint that the length of the schedule is no more than a given parameter β. It is tempting to suggest that Gonzalez's algorithm can be used to find an S_{P}^{*} by repeatedly calling his algorithm with the parameter β, where β is the schedule length obtained in a binary search on an interval in which the minimum schedule length lies. However, the binary search may not terminate in a finite amount of time because the schedule length of S_{P}^{*} may be a real number even when the execution times of the tasks are integers. Moreover, even if the number of iterations in the binary search can be bounded by a polynomial function of the size of the input, the overall running time of the algorithm will be substantially higher than that of our algorithm.

As will be seen in the next section, our algorithm employs a subroutine to preemptively schedule a subset of tasks to meet a certain deadline. This subroutine is actually a slight modification of Sahni's algorithm [16], which has been shown to solve

T_i	T_1	T_2	T_3	T_4	T_5	T_6
p_i	2	5	6	6	8	10

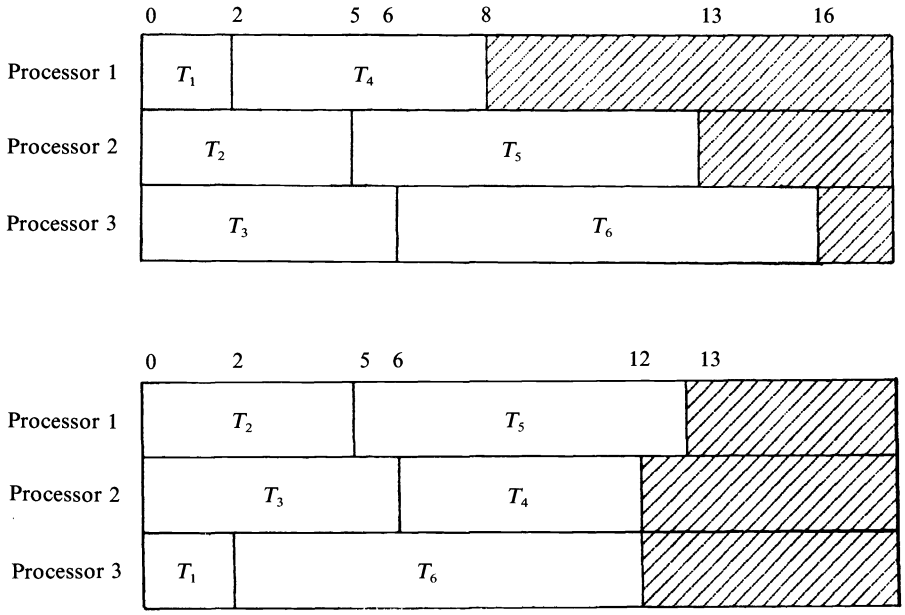


FIG. 1. *Two* SPT *schedules for a set of tasks.*

the Deadline Scheduling problem. In the Deadline Scheduling problem, we are given $m \geqq 1$ identical processors and a set of n independent tasks $TS = \{T_1, T_2, \cdots, T_n\}$ with execution times $\{p_1, p_2, \cdots, p_n\}$ and deadlines $\{d_1, d_2, \cdots, d_n\}$. The problem is to determine if there is a preemptive schedule of TS on the m processors such that all deadlines are met. Such a schedule is called a feasible schedule. The idea of Sahni's algorithm is to schedule the tasks in nondecreasing order of their deadlines. Assume the tasks have been indexed such that $d_1 \leqq d_2 \leqq \cdots \leqq d_n$, and consider the scheduling of task T_i. Let $F(k)$, $1 \leqq k \leqq m$, denote the finishing time of processor k after T_{i-1} has been scheduled. (Initially, $F(k) = 0$ for all $1 \leqq k \leqq m$.) Let $k_1, k_2, \cdots, k_{x_i}$ be the indexes of the processors such that $F(k_1) \leqq F(k_2) \leqq \cdots \leqq F(k_{x_i}) < d_i$. The scheduling of T_i is done by one of the following four cases: (1) If $p_i > d_i - F(k_1)$, then output "infeasible" (i.e., there are no feasible schedules for TS on m processors). (2) If $p_i \leqq d_i - F(k_{x_i})$, then schedule T_i completely on processor k_{x_i} and set $F(k_{x_i})$ to be $F(k_{x_i}) + p_i$. (3) If $p_i = d_i - F(k_l)$ for some $1 \leqq l < x_i$, then schedule T_i completely on processor k_l and set $F(k_l)$ to be d_i. (4) If none of the above three cases apply, let processor k_l be such that $d_i - F(k_{l+1}) < p_i < d_i - F(k_l)$. Schedule $d_i - F(k_{l+1})$ of T_i on processor k_{l+1} and the remainder of T_i on processor k_l. Set $F(k_{l+1})$ to be d_i and $F(k_l)$ to be $F(k_l) + (p_i - (d_i - F(k_{l+1})))$. The tasks are scheduled iteratively by the above procedure until

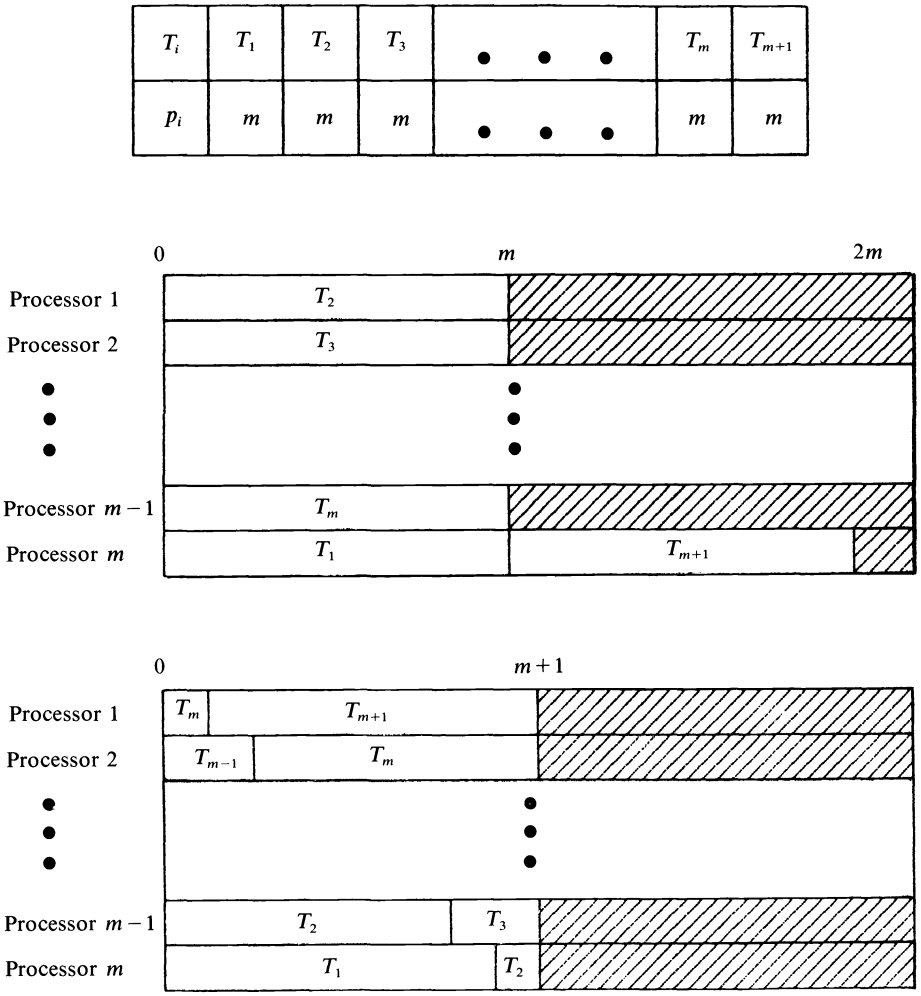


FIG. 2. *A task set such that* $SL(S^*_{NP})/SL(S^*_P) = 2m/(m+1)$.

it outputs "infeasible," or all tasks have been scheduled. In the latter case, the schedule constructed by the algorithm is a feasible schedule.

We now define some conventions and notations that will be used consistently throughout the remainder of this paper. For a given set of n tasks $TS = \{T_1, T_2, \cdots, T_n\}$, to be scheduled on m processors, we always assume that the tasks have been indexed such that $p_1 \leqq p_2 \leqq \cdots \leqq p_n$, and that $n = rm$ for some integer $r \geqq 1$. For each $1 \leqq j \leqq r$, we call the tasks $T_{(j-1)m+1}$, $T_{(j-1)m+2}, \cdots, T_{(j-1)m+m}$ the rank j tasks. Furthermore, we say that rank $(T_i) = j$ if T_i is a rank j task. We define a particular SPT schedule, denoted by $SPT^{\#}$, as follows: For each $1 \leqq k \leqq m$, assign the tasks $T_k, T_{m+k}, \cdots, T_{(r-1)m+k}$ to processor k. It is easy to see that $SPT^{\#}$ has the longest schedule length among all SPT schedules for any given set of tasks. Furthermore, if $F(k)$ denotes the finishing time of processor k in the $SPT^{\#}$ schedule, then we have $F(k) = \sum_{j=1}^{r} p_{(j-1)m+k}$ for each $1 \leqq k \leqq m$. Clearly, we have $F(k) \leqq F(k+1)$ for each $1 \leqq k < m$. Finally, an optimal schedule always means an S^*_P.

The organization of the paper is as follows. In § 2, we describe our algorithm and analyze its running time. In § 3, we prove that our algorithm always constructs an

optimal schedule for any given set of tasks. Finally, we draw some concluding remarks in the last section.

2. The algorithm. In this section, we shall describe our algorithm and analyze its running time. Given a set of n tasks, $TS=\{T_1, T_2, \cdots, T_n\}$, to be scheduled on m processors, our algorithm constructs a schedule as follows. First, we construct the $SPT^{\#}$ schedule for the tasks in the first $r-1$ ranks. Next, we compute the optimal schedule length, denoted by OSL, using the finishing times of the processors in the $SPT^{\#}$ schedule and the execution times of the rank r tasks. As will be shown later, $OSL=\max_{k=1}^{m}\{\sum_{l=1}^{k}(\mathrm{F}(l)+p_{n-l+1})/k\}$, where $\mathrm{F}(l)$, $1\leqq l\leqq m$, denotes the finishing time of processor l in the $SPT^{\#}$ schedule. Finally, we preemptively schedule the rank r tasks into the $SPT^{\#}$ schedule such that (1) no two rank r tasks finish on the same processor, (2) no processors have idle times before their finishing times, and (3) no tasks finish later than OSL. Note that the first two conditions are to ensure that the schedule has the minimum flow time. The last step of our algorithm is done by a subroutine that is a modification of Sahni's algorithm. The rank r tasks are preemptively scheduled to meet the deadline OSL in the same manner as Sahni's procedure, with the following three exceptions. First, Case 1 of Sahni's procedure no longer applies, since all tasks can meet the deadline OSL. Second, if a task T_i is scheduled on processor k_{x_i} by Case 2 of Sahni's procedure, then processor k_{x_i} will no longer be used to execute any subsequent tasks. This is to ensure that no two rank r tasks finish on the same processor. Finally, the tasks are scheduled in nonincreasing order of their execution times. A full description of our algorithm, to be called Algorithm B, is given below. A schedule produced by Algorithm B will be called a B-schedule.

ALGORITHM B

Input: A set $TS=\{T_1, T_2, \cdots, T_n\}$ of n tasks to be scheduled on $m\geqq 1$ identical processors.

Output: An optimal schedule of TS on the m processors.

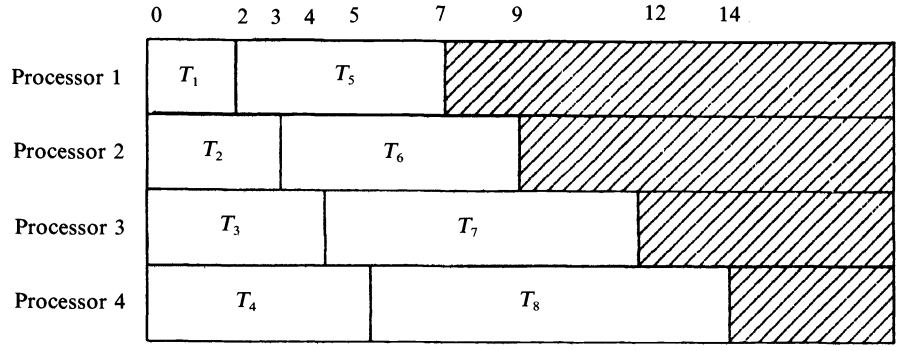
Method:

1. Construct the $SPT^{\#}$ schedule for the tasks $T_1, T_2, \cdots, T_{(r-1)m}$.
2. $\mathrm{F}(k)\leftarrow\sum_{j=1}^{r-1}p_{(j-1)m+k}$ for each $1\leqq k\leqq m$.
3. $OSL\leftarrow\max_{k=1}^{m}\{\sum_{l=1}^{k}(\mathrm{F}(l)+p_{n-l+1})/k\}$.
4. Initialize the set of available processors to be processors $1, 2, \cdots, m$. Schedule the tasks $T_n, T_{n-1}, \cdots, T_{n-m+1}$ in that order. Suppose we are scheduling task T_i. Let $k_1, k_2, \cdots, k_{x_i}$ be the indexes of the available processors such that $\mathrm{F}(k_1)\leqq\mathrm{F}(k_2)\leqq\cdots\leqq\mathrm{F}(k_{x_i})<OSL$. T_i will be scheduled by one of the following three rules: (1) If $p_i\leqq OSL-\mathrm{F}(k_{x_i})$, then schedule T_i completely on processor k_{x_i}. $\mathrm{F}(k_{x_i})\leftarrow\mathrm{F}(k_{x_i})+p_i$. Delete processor k_{x_i} from the set of available processors. (2) If $p_i=OSL-\mathrm{F}(k_l)$ for some $1\leqq l<x_i$, then schedule T_i completely on processor k_l. $\mathrm{F}(k_l)\leftarrow OSL$. Delete processor k_l from the set of available processors. (3) If the above two cases do not apply, then let l be such that $OSL-\mathrm{F}(k_{l+1})<p_i<OSL-\mathrm{F}(k_l)$. Schedule $OSL-\mathrm{F}(k_{l+1})$ of T_i on processor k_{l+1} and the remainder of T_i on processor k_l. $\mathrm{F}(k_l)\leftarrow\mathrm{F}(k_l)+(p_i-(OSL-\mathrm{F}(k_{l+1})))$. $\mathrm{F}(k_{l+1})\leftarrow OSL$. Delete processor k_{l+1} from the set of available processors. □

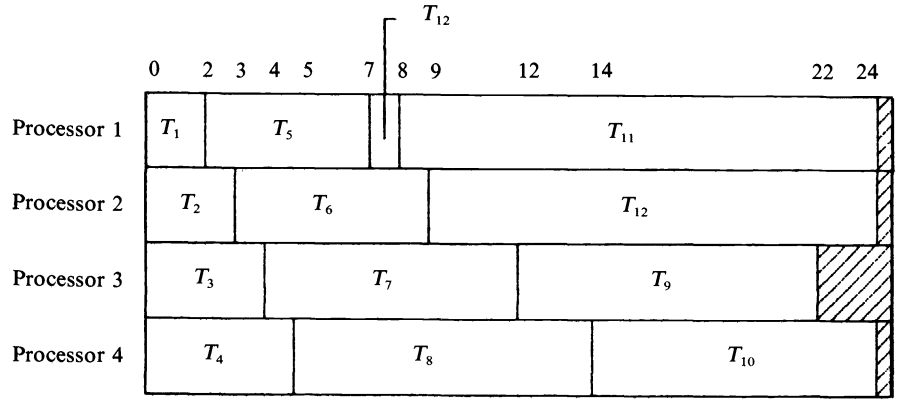
Shown in Fig. 3 is a set of 12 tasks to be scheduled on four processors. Figure 3(b) shows the $SPT^{\#}$ schedule constructed in step 1 of the algorithm. The finishing times of processors 1, 2, 3, and 4 are 7, 9, 12, and 14, respectively. Step 3 of the algorithm gives $OSL=\max\{(7+16)/1, (7+16+9+16)/2, (7+16+9+16+12+10)/3, (7+16+9+16+12+10+14+10)/4\}=24$. In step 4, T_{12} is scheduled by rule (3) on

T_i	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}	T_{11}	T_{12}
p_i	2	3	4	5	5	6	8	9	10	10	16	16

(a)



(b)



(c)

FIG. 3. *An example showing Algorithm B.*

processors 1 and 2. Then, processor 2 is deleted from the available set of processors. T_{11} is scheduled by rule (2) on processor 1, which is then deleted from the available set of processors. Finally, T_{10} and T_9 are both scheduled by rule (1) on processors 4 and 3, respectively.

The running time of Algorithm B is determined as follows. Steps 1 and 2 take $O(n)$ time. Step 3 takes $O(m)$ time. Step 4 can be implemented to run in $O(m \log m)$ time. Thus, the overall running time of Algorithm B is $O(n+m \log m)$ time. If we include the time needed to sort the tasks into nondecreasing order of their execution times, then the overall running time of the algorithm is $O(n \log n)$ time, since n is assumed to be larger than m. It is easy to see that a B-schedule has at most $m-1$ preemptions.

3. Algorithm correctness. In this section, we shall show that Algorithm B always constructs an optimal schedule. We first show that a B-schedule always gives the minimum flow time. Observe that in step 4 of Algorithm B, the rank r tasks are scheduled

in such a way that no two tasks finish on the same processor. This follows from the observation that if a task is scheduled by rules (1) or (2), then the processor to which it is assigned will not be used to schedule any subsequent tasks. If a task is scheduled by rule (3), then it finishes on processor k_{l+1} but not on processor k_l. Furthermore, processor k_{l+1} will not be used to schedule any subsequent tasks. Since a task is scheduled by one of the three rules, no two tasks can finish on the same processor. With this observation in mind, it is readily verified that the flow time of a B-schedule is $\sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m} p_{(j-1)m+k}$, which is the minimum as stated in § 1. For the remainder of this section, we shall concentrate on showing that a B-schedule has the minimum schedule length among all minimum flow-time schedules.

Let S be a schedule of $TS=\{T_1, T_2, \cdots, T_n\}$ on m processors. With respect to S, we can define an ordering of the tasks $\rho(S)=(T_{i_1}, T_{i_2}, \cdots, T_{i_n})$ such that $f_{i_1}(S) \leqq f_{i_2}(S) \leqq \cdots \leqq f_{i_n}(S)$. We call this ordering the finishing-time ordering of the tasks with respect to S. Furthermore, we call the tasks $T_{i_{(j-1)m+1}}, T_{i_{(j-1)m+2}}, \cdots, T_{i_{(j-1)m+m}}$ the rank j tasks, $1 \leqq j \leqq r$, with respect to $\rho(S)$. A schedule S is said to be in canonical form if for each $1 \leqq k \leqq m$, the tasks $T_{i_k}, T_{i_{m+k}}, \cdots, T_{i_{(r-1)m+k}}$ finish on processor k, where T_{i_l}, $1 \leqq l \leqq n$, is the lth task in $\rho(S)$. The following theorem shows that every preemptive schedule can be converted into one in canonical form with no change in schedule length and flow time.

THEOREM 1. *For every preemptive schedule S of TS on m processors, there is another schedule S' in canonical form such that $FT(S')=FT(S)$ and $SL(S')=SL(S)$.*

Proof. Let $\rho(S)=(T_{i_1}, T_{i_2}, \cdots, T_{i_n})$ be the finishing-time ordering of the tasks with respect to S. We construct a new schedule S' as follows. Suppose T_{i_1} finishes on processor $k \neq 1$ in S. We interchange a small segment, δ, of T_{i_1} executed in the time interval $[f_{i_1}(S)-\delta, f_{i_1}(S)]$ with the task executed in the same interval on processor 1. The new schedule is a valid schedule with no change in the finishing times of the tasks. Repeating this operation for $T_{i_2}, T_{i_3}, \cdots, T_{i_n}$ gives a new schedule S' such that $FT(S')=FT(S)$ and $SL(S')=SL(S)$. □

Let S be a preemptive schedule in canonical form and let $\rho(S)=(T_{i_1}, T_{i_2}, \cdots, T_{i_n})$ be the finishing-time ordering of the tasks with respect to S. We divide S into r segments $E_1, E_2, \cdots, E_r$ as follows. Segment E_1 consists of all the time intervals from time 0 until a rank 1 task (with respect to $\rho(S)$) finishes. That is, E_1 consists of the time intervals $[0, f_{i_1}(S)]$ on processor 1, $[0, f_{i_2}(S)]$ on processor 2, $\cdots$, $[0, f_{i_m}(S)]$ on processor m. For each $2 \leqq j \leqq r$, E_j consists of the time intervals $[f_{i_{(j-2)m+1}}(S), f_{i_{(j-1)m+1}}(S)]$ on processor 1, $[f_{i_{(j-2)m+2}}(S), f_{i_{(j-1)m+2}}(S)]$ on processor 2, $\cdots$, $[f_{i_{(j-2)m+m}}(S), f_{i_{(j-1)m+m}}(S)]$ on processor m. We use ES_j to denote the total length of all time intervals in segment E_j. Thus, $ES_1=\sum_{k=1}^{m} f_{i_k}(S)$, and for each $2 \leqq j \leqq r$, $ES_j=\sum_{k=1}^{m}(f_{i_{(j-1)m+k}}(S)-f_{i_{(j-2)m+k}}(S))$. With these definitions in mind, we can prove the following theorem, which characterizes the structure of a minimum flow-time preemptive schedule.

THEOREM 2. *Let S be a minimum flow-time preemptive schedule in canonical form, and let $\rho(S)=(T_{i_1}, T_{i_2}, \cdots, T_{i_n})$ be the finishing-time ordering of the tasks with respect to S. For each segment E_j, $1 \leqq j \leqq r$, we have: (1) there are no idle processor times in E_j, and (2) the tasks that are executed in E_j are rank j tasks (with respect to $\rho(S)$) only. Furthermore, every rank j task (with respect to $\rho(S)$) has execution time no larger than any rank $j+1$ task, $1 \leqq j < r$.*

Proof. Let S, $\rho(S)$, and E_j be as stated in the theorem. Let ES_j denote the total length of all time intervals in E_j, and let R_j denote the total execution time of all tasks in rank j (with respect to $\rho(S)$), $1 \leqq j \leqq r$. It is easy to see that $FT(S)=\sum_{j=1}^{r}(r-j+1)*ES_j$. Since S has the minimum flow time, we also have $FT(S)=$

$\sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m}p_{(j-1)m+k}$. Thus, we have

$$\sum_{j=1}^{r}(r-j+1)ES_j=\sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m}p_{(j-1)m+k}. \tag{1}$$

For each $1\leqq j\leqq r$, let $X_{i,j}$ denote the total execution time of all rank i tasks (with respect to $\rho(S)$) executed in E_j, $i>j$, and let ϕ_j denote the total idle processor time in E_j. Clearly, we have

$$ES_j=R_j+\phi_j+\sum_{i>j}X_{i,j}-\sum_{j>i}X_{j,i},$$

and hence

$$\sum_{j=1}^{r}(r-j+1)ES_j=\sum_{j=1}^{r}(r-j+1)R_j+\sum_{j=1}^{r}(r-j+1)\left(\phi_j+\sum_{i>j}X_{i,j}-\sum_{j>i}X_{j,i}\right). \tag{2}$$

Since $\sum_{j=1}^{r}R_j=\sum_{i=1}^{n}p_i$, we have

$$\sum_{j=1}^{r}(r-j+1)R_j\geqq\sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m}p_{(j-1)m+k}. \tag{3}$$

From (1), (2), and (3), we have

$$\sum_{j=1}^{r}(r-j+1)\left(\phi_j+\sum_{i>j}X_{i,j}-\sum_{j>i}X_{j,i}\right)\leqq 0.$$

Observing that $\phi_j\geqq 0$ for each $1\leqq j\leqq r$, and

$$\sum_{j=1}^{r}(r-j+1)\left(\sum_{i>j}X_{i,j}-\sum_{j>i}X_{j,i}\right)=\sum_{j=1}^{r-1}\sum_{i>j}(i-j)X_{i,j}\geqq 0,$$

we have $\phi_j=0$ and $X_{i,j}=0$, $i>j$, for each $1\leqq j\leqq r$. Therefore, each segment E_j can only have rank j tasks (with respect to $\rho(S)$) executing in it, and it cannot have any idle processor time. Finally, from (1) and (2), we have

$$\sum_{j=1}^{r}(r-j+1)R_j=\sum_{j=1}^{r}(r-j+1)\sum_{k=1}^{m}p_{(j-1)m+k}. \tag{4}$$

From (4), we observe that no rank j tasks (with respect to $\rho(S)$) can have execution time larger than any rank $j+1$ tasks; otherwise, (4) cannot hold. □

From Theorem 2, we see that a minimum flow-time preemptive schedule, S, in canonical form is actually very similar to a B-schedule. The rank j tasks with respect to $\rho(S)$ are the same as the rank j tasks of the task system, $1\leqq j\leqq r$. The tasks are executed rank by rank in S, and there are no idle processor times before the finishing time of a processor. If S is a schedule, we let $P(S)$ denote the partial schedule obtained from S by deleting all rank r tasks.

LEMMA 1. *Let S be a minimum flow-time preemptive schedule in canonical form and let S' be a B-schedule. Let* $\mathrm{F}(k, S)$ *and* $\mathrm{F}(k, S')$ *denote the finishing time of processor k in $P(S)$ and $P(S')$, respectively, $1\leqq k\leqq m$. Then, we have* $\sum_{k=l}^{m}(\mathrm{F}(k, S')-\mathrm{F}(k, S))\geqq 0$ *for all $1\leqq l\leqq m$.*

Proof. Both $P(S)$ and $P(S')$ consist of the same set of tasks and both schedule tasks rank by rank. The lemma will be proved if we can show that $\sum_{k=1}^{l}(\mathrm{F}(k, S')-\mathrm{F}(k, S))\geqq 0$ for all $1\leqq l\leqq m$. But this follows immediately from the observation that $P(S')$ is an $SPT^{\#}$ schedule. □

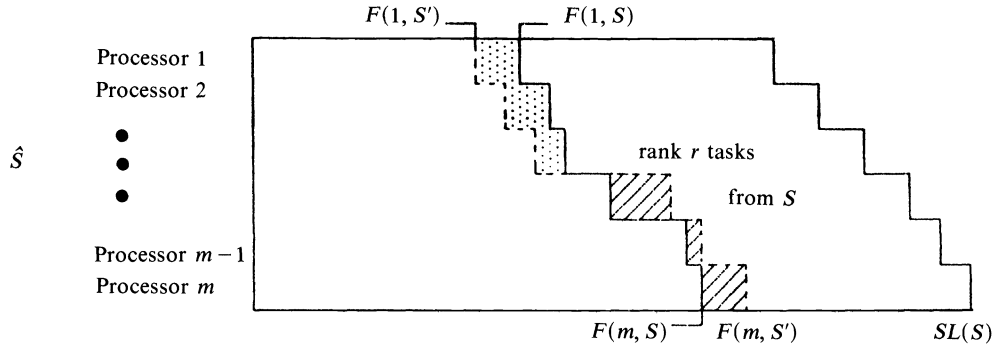
THEOREM 3. *Let S be a minimum flow-time preemptive schedule in canonical form and let S' be a B-schedule. Then, there is another schedule $\hat{S}$ such that $FT(\hat{S}) = FT(S)$, $SL(\hat{S}) = SL(S)$, and $\mathrm{F}(k, \hat{S}) = \mathrm{F}(k, S')$, $1 \leqq k \leqq m$, where $\mathrm{F}(k, \hat{S})$ and $\mathrm{F}(k, S')$ denote the finishing time of processor k in $P(\hat{S})$ and $P(S')$, respectively.*

Proof. Let S be a minimum flow-time preemptive schedule in canonical form and let S' be a B-schedule. We construct $\hat{S}$ using the following three steps: (1) Copy the rank r tasks from S to $\hat{S}$. As shown in Fig. 4(a), the solid lines enclose the rank r tasks. The solid staircase on the left is given by $\mathrm{F}(k, S)$, $1 \leqq k \leqq m$, where $\mathrm{F}(k, S)$ denotes the finishing time of processor k in $P(S)$. The dotted staircase is given by $\mathrm{F}(k, S')$, $1 \leqq k \leqq m$. Processor k is said to be a cross-hatched processor if $\mathrm{F}(k, S') > \mathrm{F}(k, S)$; the time interval $[\mathrm{F}(k, S), \mathrm{F}(k, S')]$ is said to be a cross-hatched interval. Processor k is said to be a dotted processor if $\mathrm{F}(k, S) > \mathrm{F}(k, S')$; the time interval $[\mathrm{F}(k, S'), \mathrm{F}(k, S)]$ is said to be a dotted interval. As shown in Fig. 4(a), processor 1 is a dotted processor, whereas processor m is a cross-hatched processor. (2) Move all portions of the rank r tasks executed in a cross-hatched interval on a cross-hatched processor to a dotted interval on a dotted processor. The moving is done systematically by starting from the highest indexed cross-hatched (dotted) processor to the lowest indexed cross-hatched (dotted) processor. From Lemma 1, we know that for any $1 \leqq l \leqq m$, the total length of all cross-hatched intervals is no less than the total length of all dotted intervals on processors l to m. Therefore, we will always be moving from a higher indexed processor to a lower indexed processor. We need to show that this transformation gives a valid schedule with no change in the finishing times of the rank r tasks. Since each rank r task has execution time larger than $\mathrm{F}(m, S') - \mathrm{F}(1, S')$, no rank r task can finish by $\mathrm{F}(m, S')$ in S. Thus, the transformation will not change the finishing times of rank r tasks. When we move portions of a rank r task, say T_i, executed in a cross-hatched interval $[t_1, t_2]$ to a dotted interval $[t_1', t_2']$, we need to ensure that T_i is not executed in $[t_1', t_2']$ on some other processor. This can be done by swapping tasks among processors in $[t_1, t_2]$ before the move to avoid possible conflicts. Since we are always moving from a higher indexed processor to a lower indexed processor, we can always swap tasks among processors in $[t_1, t_2]$ to avoid conflicts. Thus, the schedule after the transformation is still a valid schedule. Figure 4(b) shows the schedule of the rank r tasks after the transformation. (3) Construct the SPT^* schedule for the tasks in the first $r-1$ ranks. Figure 4(c) shows the completed schedule $\hat{S}$.

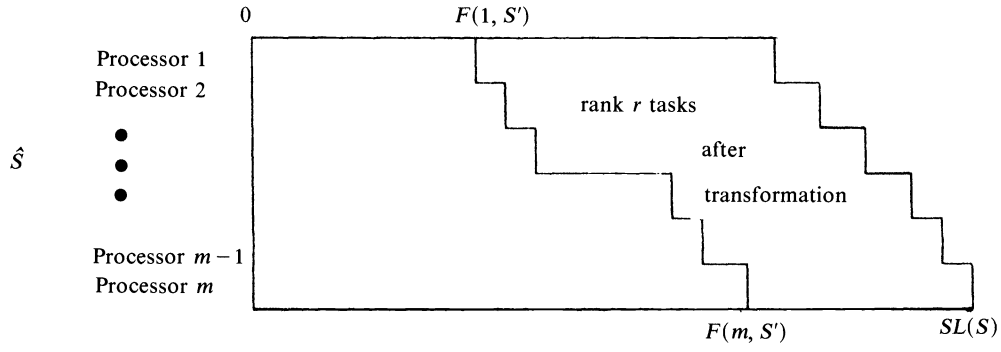
From the construction, it is easy to see that $FT(\hat{S}) = FT(S') = FT(S)$, $SL(\hat{S}) = SL(S)$, and $\mathrm{F}(k, \hat{S}) = \mathrm{F}(k, S')$ for each $1 \leqq k \leqq m$. □

By Theorems 1 and 3, an optimal schedule can always be found such that the tasks in the first $r-1$ ranks are scheduled in the same manner that they are scheduled by Algorithm B. Thus, to show that a B-schedule has the minimum schedule length among all minimum flow-time schedules, all we need to show is that Algorithm B schedules the rank r tasks in such a way that the resulting schedule length is minimum. In the following, we shall show that: (I) An optimal schedule must schedule the rank r tasks such that the resulting schedule length is at least *OSL*, which is computed in step 3 of Algorithm B and (II) Algorithm B can always schedule the rank r tasks such that no rank r tasks finish later than *OSL*.

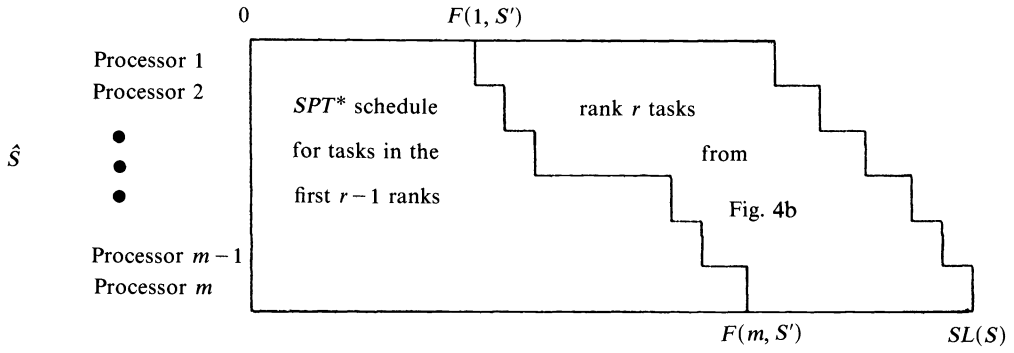
Before we show the above, we need to introduce the following notations. Let $\mathrm{F}(k)$, $1 \leqq k \leqq m$, denote the finishing time of processor k of a partial schedule S such that $\mathrm{F}(1) \leqq \mathrm{F}(2) \leqq \cdots \leqq \mathrm{F}(m)$. The m-tuple $\mathrm{PF} = (\mathrm{F}(1), \mathrm{F}(2), \cdots, \mathrm{F}(m))$ is called the processor profile of S. For a given $D > \mathrm{F}(m)$, the ordered pair (PF, D) is called a block and it consists of the time intervals $[\mathrm{F}(1), D]$ on processor 1, $[\mathrm{F}(2), D]$ on processor 2, $\cdots$, $[\mathrm{F}(m), D]$ on processor m.



(a) *Step 1 in the construction.*



(b) *Step 2 in the construction.*



(c) *Step 3 in the construction.*

FIG. 4. *Illustrating the proof of Lemma* 3.

THEOREM 4. *Let* $\mathrm{PF} = (\mathrm{F}(1), \mathrm{F}(2), \cdots, \mathrm{F}(m))$ *be the processor profile of a partial schedule S, and let* $TS' = \{T'_1, T'_2, \cdots, T'_m\}$ *be a set of m tasks such that* $p'_1 \geqq p'_2 \geqq \cdots \geqq p'_m$. *Then, TS' can be preemptively scheduled into the block* (PF, D) *only if* $D \geqq \max_{k=1}^{m} \{\sum_{l=1}^{k} (\mathrm{F}(l) + p'_l)/k\}$.

Proof. Let j be such that $\sum_{l=1}^{j}(\mathrm{F}(l)+p'_l)/j = \max_{k=1}^{m}\{\sum_{l=1}^{k}(\mathrm{F}(l)+p'_l)/k\}$. Since $D-\mathrm{F}(1)\geqq D-\mathrm{F}(2)\geqq\cdots\geqq D-\mathrm{F}(m)$, the processors are in nonincreasing order of processing capacity. Therefore, if TS' can be scheduled into the block (PF, D), then we must be able to schedule $T'_1, T'_2, \cdots, T'_j$ completely on processors $1, 2, \cdots, j$ of the block. Thus, $D\geqq\sum_{l=1}^{j}(\mathrm{F}(l)+p'_l)/j = \max_{k=1}^{m}\{\sum_{l=1}^{k}(\mathrm{F}(l)+p'_l)/k\}$. □

Theorem 4 shows that *OSL* computed in step 3 of Algorithm B is a lower bound of the schedule length of an optimal schedule. We shall show in Theorem 5 that step 4 of Algorithm B can always schedule the rank r tasks into the block (PF, *OSL*), where PF is the processor profile of the SPT^{*} schedule constructed in step 1 of Algorithm B. Note that the rank r tasks are scheduled in nonincreasing order of their execution times. Task T_{n-l+1} is scheduled at the lth iteration of step 4 of Algorithm B, $1\leqq l\leqq m$. The next three lemmas are instrumental in proving Theorem 5.

LEMMA 2. *In step* 4 *of Algorithm B, the number of available processors remaining after the* lth *iteration is* $m-l$, $1\leqq l\leqq m$.

Proof. The lemma follows from the observations that we start out with m processors, and that each iteration removes one processor from the set of available processors. □

LEMMA 3. *Let* $\mathrm{F}(k, l)$, $1\leqq k\leqq m$, $1\leqq l\leqq m$, *denote the finishing time of processor* k *before the* lth *iteration in step* 4 *of Algorithm B. Let processors* $k_1<k_2<\cdots<k_{m-l+1}$ *be the available processors before the* lth *iteration. Then, we have* $\mathrm{F}(k_1, l)\leqq\mathrm{F}(k_2, l)\leqq\cdots\leqq\mathrm{F}(k_{m-l+1}, l)$.

Proof. We shall prove the lemma by induction on l. The lemma is clearly true for $l=1$, since $\mathrm{F}(1, 1)\leqq\mathrm{F}(2, 1)\leqq\cdots\leqq\mathrm{F}(m, 1)$. Assume the lemma is true for $l=j$, we shall prove it is also true for $l=j+1$. In the jth iteration, task T_{n-j+1} is being scheduled. Let $k_1<k_2<\cdots<k_{m-j+1}$ be the indexes of the available processors before the jth iteration. By the inductive hypothesis, we have $\mathrm{F}(k_1, j)\leqq\mathrm{F}(k_2, j)\leqq\cdots\leqq\mathrm{F}(k_{m-j+1}, j)$. If T_{n-j+1} is scheduled by rules (1) or (2) in step 4 of Algorithm B, then the lemma is clearly true for $l=j+1$. Thus, we assume that T_{n-j+1} is scheduled by rule (3) of step 4. Let processors k_c and k_{c+1} be used in the scheduling of T_{n-j+1}. Then, we have $\mathrm{F}(k_c, j+1)\leqq\mathrm{F}(k_{c+1}, j)\leqq\mathrm{F}(k_{c+2}, j)$, and $\mathrm{F}(k_i, j+1)=\mathrm{F}(k_i, j)$ for $i\neq c$ and $c+1$. Thus, we have $\mathrm{F}(k_1, j+1)\leqq\mathrm{F}(k_2, j+1)\leqq\cdots\leqq\mathrm{F}(k_c, j+1)\leqq\mathrm{F}(k_{c+2}, j+1)\leqq\cdots\leqq\mathrm{F}(k_{m-j+1}, j+1)$. Hence, the lemma is also true for $l=j+1$. □

LEMMA 4. *Let processors* $k_1<k_2<\cdots<k_{m-l+1}$ *be the available processors before the* lth *iteration in step* 4 *of Algorithm B. If task* T_{n-l+1} *is scheduled by rules* (1) *or* (2) *in the* lth *iteration on processor* k_c, *or if* T_{n-l+1} *is scheduled by rule* (3) *on processors* k_c *and* k_{c+1}, *then after* T_{n-l+1} *is scheduled there are at least* $c-1$ *available processors that have processing capacity at least* p_{n-l+1}.

Proof. Let $\mathrm{F}(k, l)$ denote the finishing time of processor k before the lth iteration, and let $k_1<k_2<\cdots<k_{m-l+1}$ be the indexes of the available processors. If T_{n-l+1} is scheduled by rule (1) or rule (2) on processor k_c, or if T_{n-l+1} is scheduled by rule (3) on processors k_c and k_{c+1}, then we must have $OSL-\mathrm{F}(k_c, l)\geqq p_{n-l+1}$. Since $OSL-\mathrm{F}(k_i, l)\geqq OSL-\mathrm{F}(k_c, l)$ and $\mathrm{F}(k_i, l+1)=\mathrm{F}(k_i, l)$ for each $1\leqq i<c$, we have $OSL-\mathrm{F}(k_i, l+1)\geqq p_{n-l+1}$ for each $1\leqq i<c$. Thus, we have at least $c-1$ processors that have processing capacity no smaller than p_{n-l+1}. □

THEOREM 5. *Step* 4 *of Algorithm B can always schedule the tasks* $T_n, T_{n-1}, \cdots, T_{n-m+1}$ *into the block* (PF, *OSL*), *where* PF *is the processor profile of the* SPT^{*} *schedule constructed in step* 1 *and OSL is computed in step* 3 *of Algorithm B.*

Proof. We prove the theorem by contradiction and assume that at the lth iteration T_{n-l+1} is the first task that cannot be scheduled into the block. We first observe that none of the tasks $T_n, T_{n-1}, \cdots, T_{n-l+2}$ were scheduled by rule (1) of step 4 of Algorithm

B. For otherwise, since the tasks are in nonincreasing order of their execution times, all tasks can be scheduled into the block. Let $F(k, l)$, $1 \leqq k \leqq m$, denote the finishing time of processor k at the time T_{n-l+1} is scheduled, and let processors $k_1 < k_2 < \cdots < k_{m-l+1}$ be the available processors. By Lemma 3, we have $F(k_1, l) \leqq F(k_2, l) \leqq \cdots \leqq F(k_{m-l+1}, l)$, and hence $OSL - F(k_1, l) \geqq OSL - F(k_2, l) \geqq \cdots \geqq OSL - F(k_{m-l+1}, l)$. Since T_{n-l+1} cannot be scheduled, we have $p_{n-l+1} > OSL - F(k_1, l)$.

By Lemma 2, $k_1 \leqq l$. We want to show that $k_2 > l$. If $k_2 \leqq l$, then, by Lemma 2, there is a processor $x > l$ that has been deleted as a result of scheduling some task T_{n-y+1}, $y < l$. By Lemmas 4 and 2, there are at least $(x-2)-(y-1) = x-y-1 \geqq l-y$ available processors among processors $1, 2, \cdots, l$ with processing capacity no smaller than p_{n-y+1}. Since the tasks are in nonincreasing order of their execution times, we must be able to schedule the tasks $T_{n-y+2}, T_{n-y+3}, \cdots, T_{n-l+1}$ into one of these processors. But this contradicts our assumption that T_{n-l+1} cannot be scheduled. Thus, $k_2 > l$.

From the above discussions, there is only one available processor among processors $1, 2, \cdots, l$ at the time T_{n-l+1} is scheduled. Since $T_n, T_{n-1}, \cdots, T_{n-l+2}$ were all scheduled by rules (2) and (3) of step 4 of Algorithm B, and since T_{n-l+1} cannot be scheduled, we must have $\sum_{k=1}^{l} (F(k) + p_{n-k+1})/l > OSL$, where $F(k)$ is the finishing time of processor k in the $SPT^{\#}$ schedule. But this contradicts the definition of OSL. □

From the above theorems, we have the main result of this paper, which is stated in the following theorem.

THEOREM 6. *Algorithm B always generates an optimal schedule.*

4. Concluding remarks. In this paper, we have given an $O(n \log n)$-time algorithm to find a minimum-length preemptive schedule subject to the constraint that it must have the minimum flow time. As a generalization of our problem, it will be interesting to find a minimum-length preemptive schedule subject to the constraint that the flow time is not larger than a given parameter α. One can also generalize this problem to the case of uniform processors. We note that, for the case of uniform processors, Gonzalez [8] has given an $O(nm)$-time algorithm to find a minimum flow-time preemptive schedule subject to the constraint that the schedule length is not larger than a given parameter β.

REFERENCES

[1] J. L. BRUNO, E. G. COFFMAN, JR., AND R. SETHI, *Algorithms for minimizing mean flow time*, Inform. Process. (1974), pp. 504–510.

[2] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *An application of bin packing to multiprocessor scheduling*, SIAM J. Comput., 17 (1978), pp. 1–17.

[3] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Bin packing with divisible item sizes*, J. Complexity, to appear.

[4] E. G. COFFMAN, JR. AND R. SETHI, *Algorithm minimizing mean flow time: schedule-length properties*, Acta Inform., 6 (1976), pp. 1–14.

[5] R. W. CONWAY, W. L. MAXWELL, AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

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

[7] T. GONZALEZ, *Minimizing the mean and maximum finishing time on identical processors*, Tech. Report CS-78-15, Computer Science Dept., Pennsylvania State University, University Park, PA, 1978.

[8] ———, *Minimizing the mean and maximum finishing time on uniform processors*, Tech. Report CS-78-22, Computer Science Dept., Pennsylvania State University, University Park, PA, 1978.

[9] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[10] D. S. HOCHBAUM AND D. B. SHMOYS, *Using dual approximation algorithms for scheduling problems: Theoretical and practical results*, J. ACM, 34 (1987), pp. 144–162.
[11] K. S. HONG AND J. Y-T. LEUNG, unpublished results.
[12] M. A. LANGSTON, *Processor Scheduling with Improved Heuristic Algorithms*, Ph.D. thesis, Texas A&M University, College Station, TX, 1981.
[13] J. Y-T. LEUNG, *On scheduling independent tasks with restricted execution times*, Oper. Res., 30 (1982), pp. 163–171.
[14] R. MCNAUGHTON, *Scheduling with deadlines and loss functions*, Management Sci., 12 (1959), pp. 1–12.
[15] S. SAHNI, *Algorithms for scheduling independent tasks*, J. ACM, 23 (1976), pp. 116–127.
[16] ———, *Preemptive scheduling with due dates*, Oper. Res., 27 (1979), pp. 925–934.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 327-349, April 1989

011

FAST PARALLEL ALGORITHMS FOR CHORDAL GRAPHS*

JOSEPH NAOR†, MONI NAOR‡, AND ALEJANDRO A. SCHÄFFER§

Abstract. Techniques for parallel algorithms on chordal graphs are developed. An NC algorithm for recognizing chordal graphs is developed, as are NC algorithms for finding the following objects in chordal graphs: all maximal cliques, an intersection graph representation, an optimal coloring, a perfect elimination scheme, a weighted maximum independent set, and a minimum clique cover. The recognition algorithm presented in this paper is simpler than previous algorithms given by Edenbrandt and by Chandrasekharan and Iyengar; the other problems were apparently open. The known polynomial-time algorithms for these problems seem highly sequential, and therefore a different approach to find parallel algorithms is used.

Key words. parallel algorithms, chordal graphs, perfect elimination scheme, perfect graphs, maximum independent set, chromatic number

AMS(MOS) subject classifications. 05C05, 05C15, 05C70, 05C75, 68Q20, 68R10

1. Introduction.

An undirected graph is *chordal* if every cycle of length at least four contains a chord, that is, an edge between two vertices that are not consecutive in the cycle. Chordal graphs arise in the study of sparse matrices [Ro70], [Ro72] and in the study of acyclic relational database schemes [Be83]. Many NP-complete problems can be solved in polynomial time if the input graph is chordal, but the standard algorithms seem highly sequential [Jo85]. We give fast parallel algorithms for several problems on chordal graphs.

Chordal graphs comprise an important subclass of perfect graphs. Perfect graphs are characterized by either of two min-max identities:

1) the chromatic number equals the maximum clique size for every induced subgraph

2) the maximum independent set and the minimum clique cover are of the same size for every induced subgraph.

These two characterizations were developed somewhat independently, but Lovász [Lo72] showed that they describe precisely the same graphs.

The problem of recognizing perfect graphs is open, but chordal graphs can be recognized in polynomial time. Rose, Tarjan, and Lueker [Ro76] and Tarjan and Yannakakis [Ta84] gave linear-time sequential algorithms for this problem based on an alternative characterization of chordal graphs. To understand this characterization, we need some definitions: a vertex, v, is *simplicial* if the graph induced by v and its neighbors is a clique. We use the term *clique* to mean a completely connected subgraph; we use the term *maximal clique* to mean a clique such that no vertex outside the clique is adjacent to every vertex in the clique. An ordering of the vertices $v_1, v_2, \cdots, v_n$

* Received by the editors April 13, 1987; accepted for publication (in revised form) July 19, 1988. Most of these results were presented in preliminary form at the 19th Annual Symposium on Theory of Computing in May 1987.

† Department of Computer Science, Hebrew University, Jerusalem 91904, Israel. Part of the work was done while this author was visiting the University of California at Berkeley. Present address, Computer Science Department, Stanford University, Stanford, California 94305.

‡ Computer Science Division, University of California at Berkeley, Berkeley, California 94720. The work of this author was supported by National Science Foundation grants DCR85-13926 and CCR88-13632. Present address, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

§ Computer Science Department, Stanford University, Stanford, California 94305. This work was done while this author was a Research Student Associate at IBM Almaden Research Center. During the academic year, the work of this author is primarily supported by a Fannie and John Hertz Foundation Fellowship and is supported in part by Office of Naval Research contract N00014-85-C-0731.

with v_i simplicial in the graph induced by $\{v_i, v_{i+1}, \cdots, v_n\}$ for all i is called a *perfect elimination scheme* or PES, for short. A graph is chordal if and only if it has a perfect elimination scheme [Fu65], [Go80].

Interval graphs, which arise in a variety of scheduling and resource-allocation problems [Go80], form an important subclass of chordal graphs. An *interval graph* is the intersection graph of intervals on the real line. There is a bijection between the vertex set of the graph and a set of intervals such that two vertices are adjacent if and only if the corresponding intervals overlap.

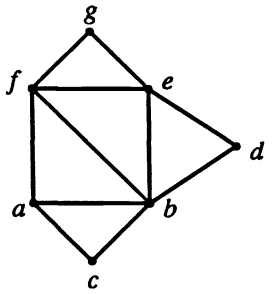


FIG. 1. *A chordal graph; one perfect elimination scheme is cadbefg.*

Our parallel algorithms use an intersection graph characterization of chordal graphs due to Buneman [Bu74] and Gavril [Ga74]. A graph G is chordal if and only if it is the intersection graph of a family of subtrees of a tree. That is to say, there is an undirected tree T, a family of subtrees $\mathcal{S} = \langle S_1, S_2, \cdots, S_{|V|} \rangle$, and a bijection from $V(G)$ to the multiset $\mathcal{S}$ with the property that two vertices in G are adjacent if and only if the corresponding subtrees share a node in T. We use the convention that an element of $V(G)$ is called a *vertex*, and if T is fixed, an element of $V(T)$ is called a *node* or a *tree node*. Buneman [Bu74] and Gavril [Ga74] gave different polynomial-time algorithms for building a tree-and-subtrees representation of a chordal graph.

A PES can be used to give polynomial-time algorithms on chordal graphs for many graph problems that are NP-complete on general inputs [Jo85]. Algorithms to build a PES and use it, appear hard to parallelize because they choose one vertex to process at a time based on previous choices. Algorithms that build and use the tree representation appear hard to parallelize because the tree may have long paths.

Our main results are:

THEOREM 1.1. *Chordal graphs can be recognized in polylogarithmic time on a* PRAM (*parallel random access machine*) *having polynomially many processors. Given a chordal graph, it is possible to find the following objects in polylogarithmic time on a* PRAM *having polynomially many processors*:

— *all maximal cliques and the chromatic number* (§ 3),
— *a tree-and-subtrees representation* (§ 4),
— *an explicit optimal coloring* (§ 4),
— *a perfect elimination scheme* (§ 5),
— *unweighted and weighted maximum independent sets* (§ 5), *and*
— *a minimum clique cover* (§ 5).

Next to each item is the section in which we give its algorithm.

Sequential algorithms for all these problems, except building a tree representation, follow directly from the PES, and to build a tree, the most important step is to find

all maximal cliques from the PES. Our parallel algorithms that recognize chordal graphs and find all maximal cliques use neither a tree-and-subtrees representation nor a PES. The maximal cliques are used to build the tree and to find a coloring; we use the tree to find a PES and a maximum independent set. The coloring algorithm uses only the existence of the tree to guide the recursion, but does not require an explicit tree or a PES. We present algorithms for both the unweighted and weighted independent set problems because the algorithm for the unweighted case runs faster and uses fewer processors in the worst case. In a longer technical report [Na87], we extend our results to obtain simple parallel algorithms for three sample problems related to chordal graphs: computing the chromatic polynomial of a chordal graph, deciding whether a database scheme is acyclic [Be83], and finding a largest k-colorable subgraph of a chordal graph [Ya87].

We take our processors to be Concurrent-Read Exclusive-Write (CREW) PRAMs, and all our algorithms run within the bounds required for the complexity class commonly known as NC, which means they use polynomially many processors, and their runtime is polylogarithmic. All our algorithms except those for weighted independent set and maximum k-colorable subgraph run in $O(\log^2 n)$ time, where n is the number of vertices.

Edenbrandt [Ed85], [Ed87] and Chandrasekharan and Iyengar [Ch86] gave NC algorithms to recognize chordal graphs. Our recognition algorithm uses fewer processors on sparse graphs, but its main advantage over the previous algorithms is that its correctness proof is much simpler. The other seven problems appear to have been open. Coincidentally with the preparation of this paper and independent from it, Dahlhaus and Karpinski [Da86] discovered NC algorithms very different from ours for finding a PES, all maximal cliques, and a tree representation; their algorithms are sketched in [Da86]. They present their algorithms as polynomial-time algorithms on a LOGSPACE-bounded alternating Turing machine. To make the algorithms deterministic and suitable for a PRAM, they rely on the simulation of Ruzzo [Ru80]. The resulting deterministic algorithms require at least $\Omega(n^9)$ processors [Da87] and run in $O(\log^2 n)$ time, while our algorithms for all the problems we solve, except weighted independent set and maximum k-colorable subgraph, use at most $O(n^5)$ processors to run in $O(\log^2 n)$ time or $O(n^4)$ processors to run in $O(\log^3 n)$ time.

After the publication of [Na87], Dahlhaus and Karpinski found a new maximal cliques algorithm that simultaneously achieves $O(\log^2 n)$ time and $O(n^4)$ processors [DK87]. Klein has announced more efficient NC algorithms for all the problems mentioned in Theorem 1.1, except maximum weight independent set. His algorithms run in $O(\log^2 n)$ time using $O(m+n)$ processors on the more powerful Concurrent-Read Concurrent-Write (CRCW)PRAM model [Kl88].

Finding a perfect elimination scheme in NC is explicitly left open in [Ed85], [Ed87], and [Ch86]. The problems of finding maximal cliques and a maximum independent set are mentioned in [Ch86].

Our algorithms for finding a PES and maximum independent set use a new recursion technique on trees: at every stage we process and prune all terminal branches of the tree that represents the chordal graph. A terminal branch is a path containing only nodes of degree two and a leaf. A variety of parallel algorithms for graph problems, expression evaluation, and language recognition use bottom-up recursion on rooted trees [Mi85]. However, these algorithms change the edge structure of the tree during the computation, so as to shorten long paths. Our algorithms, which process terminal branches, do the computations without changing the tree structure (except for deleting vertices already processed).

To build the tree representation and to give an optimal coloring, we use recursion around a centroid. This method has been used to give parallel algorithms for other tree problems where the tree structure is completely known [Me83]. However, we use recursion around a centroid to build the tree when we know only its nodes (and what they represent in the graph), but not its edges.

2. Recognizing chordal graphs. We first consider the problem of recognizing a chordal graph. The fastest sequential solutions rely on the existence of a PES [Ro76], [Ta84].

Edenbrandt gives an NC-recognition algorithm that searches for certain types of clique separators [Ed85], [Ed87]. Chandrasekharan and Iyengar give a recognition algorithm similar to Edenbrandt's [Ch86]. The main advantage of our algorithm is that its correctness follows easily from the definition of chordal graphs as graphs that contain no induced chordless cycles of length greater than 3. A secondary advantage is that on sparse graphs it uses asymptotically fewer processors.

Let $N(v)$ denote the set of vertices adjacent to a vertex v. Let $G\backslash v$ be the graph induced by $V(G)\backslash\{v\}$; if W is a subset of $V(G)$, let $G\backslash W$ be the graph induced by $V(G)\backslash W$. The following lemma suggests a recognition algorithm.

LEMMA 2.1. *The graph G is not chordal if and only if it contains a vertex v with the property that a connected component of $(G\backslash v)\backslash N(v)$ is adjacent to two vertices v_i, $v_j \in N(v)$, and the vertices v_i and v_j are not adjacent.*

Proof. If such a vertex v exists, then G has a chordless cycle including v_1, v, v_j, and vertices in the connected component of $(G\backslash v)\backslash N(v)$ adjacent to v_i and v_j. To prove the converse, suppose that $v_1 - v_2 - \cdots - v_p - v_1$ with $p \geqq 4$ is a shortest chordless cycle of length at least 4 in G. In $(G\backslash v_1)\backslash N(v_1)$, vertices $v_3, \cdots, v_{p-1}$ are in the same connected component, which is adjacent to both v_2 and v_p in G. Letting $v = v_1$, $v_i = v_2$, and $v_j = v_p$ satisfies the conditions of the Lemma. □

ALGORITHM TO RECOGNIZE A CHORDAL GRAPH
Input: A simple undirected graph $G = (V, E)$.
Output: Is G chordal?

1. For every vertex $v \in V$ *in parallel*: Find all connected components of $(G\backslash v)\backslash N(v)$ and number them.
2. For every vertex $V \in V$ *in parallel*: Compute the set of pairs $\langle v_i, v_j \rangle$, such that v_i and v_j are distinct neighbors of v, but v_i and v_j are not adjacent.
3. For every vertex $v \in V$ *in parallel*: For every neighbor v_i of v, compute a sorted list of the connected components in $(G\backslash v)\backslash N(v)$ to which v_i is adjacent.
4. For every vertex $v \in V$ *in parallel*: Test if there exist vertices v_i and v_j in $N(v)$ such that v_i and v_j are not adjacent to each other, but there exists a connected component A in $(G\backslash v)\backslash N(v)$, such that both v_i and v_j are adjacent to A.
5. If the test in step 4 is not satisfied for every vertex $v \in V$, then G is chordal; if the test is satisfied for some vertex, then G is not chordal.

Correctness of the algorithm follows immediately from Lemma 2.1.

LEMMA 2.2. *The above recognition algorithm can be implemented to run in $O(\log^2 n)$ time using $O(mn^2)$ processors, where m is the number of edges.*

Proof. In step 1 we can use the connected-components algorithm of Hirschberg, Chandra, and Sarwate [Hi79], which requires $O(\log^2 n)$ time and $O(n^2)$ processors. We do n connected components computations in parallel, so we need $O(n^3)$ processors for step 1.

Steps 2 and 3 can be implemented $O(\log n)$ and $O(\log^2 n)$ time, respectively, both using $O(mn)$ processors.

Step 4 can be implemented with $O(mn^2)$ processors in $O(\log n)$ time. To every triple $\langle v, v_i, v_j\rangle$ such that v is adjacent to v_i, v is adjacent to v_j, v_i and v_j are not adjacent, and $v_i < v_j$, assign one processor for every component on the list of components which we computed for the pair $\langle v, v_i\rangle$ in step 3. The total length of the component lists for all pairs $\langle v, v_i\rangle$ is $O(mn)$. To each entry on such a list assign at most $n-2$ processors, one for each choice of v_j. The total number of processors is $O(mn^2)$. A processor corresponding to entry C for the triple $\{v, v_i, v_j\}$ does a binary search for C on the component list of $\langle v, v_j\rangle$ and records the result (1 if it finds C, 0 otherwise). For each fixed value of v, all processors corresponding to triples with v as the first component do a boolean OR of their results. This leaves n boolean values for step 5.

Step 5 can be implemented with $O(n)$ processors in $O(\log n)$ time by doing a boolean OR of the n values computed in step 4. □

3. Maximal cliques. We now develop an algorithm to compute all maximal cliques in a chordal graph. A chordal graph can have at most n maximal cliques [Go80]. Define a graph G to be a *bi-clique* if its vertex set can be divided into two disjoint sets P and Q such that the induced graphs G_P and G_Q are cliques. We reduce computing all maximal cliques in any chordal graph to computing all maximal cliques in a chordal bi-clique using divide and conquer.

ALGORITHM TO COMPUTE ALL MAXIMAL CLIQUES (SUMMARY)
Input: A chordal graph G.
Output: A list containing each maximal clique of G exactly once.

1. Divide the vertex set of G arbitrarily into two disjoint sets A and B of equal size. (If n is odd, then $|A|-|B|=1$.)
2. Recursively compute all maximal cliques in the induced graphs G_A and G_B.
3. For every pair of maximal cliques $P \subset A$ and $Q \subset B$, compute all maximal cliques of $A \cup B$ in the induced subgraph $G_{P\cup Q}$, which is a bi-clique.
4. Eliminate duplicates occurring in more than one bi-clique.

LEMMA 3.1. *The above algorithm to compute all maximal cliques in a chordal graph G is correct.*

Proof. Every maximal clique of G not contained entirely in either G_A or G_B is contained in the union of two maximal cliques—one from A and the other from B. □

We now show how to compute all maximal cliques in a bi-clique. Let the vertex set of the chordal bi-clique be $CA \cup CB$, where $CA \subset A$, $CB \subset B$, and CA and CB induce cliques. If $v \in CA$ and $w \in CB$, then let $N_{CB}(v)$ be the set of neighbors of v in CB, and let $N_{CA}(w)$ be the set of neighbors of w in CA.

LEMMA 3.2. *The $|CA|$ sets $\{N_{CB}(v) \mid v \in CA\}$ are ordered by inclusion. Similarly, the $|CB|$ sets $\{N_{CA}(x) \mid x \in CB\}$ are ordered by inclusion.*

Proof. Suppose that there were four vertices $v', v'' \in CA$ and $w', w'' \in CB$ such that $w' \in N_{CB}(v')$ and $w'' \in N_{CB}(v'')$ but, $w' \notin N_{CB}(v'')$ and $w'' \notin N_{CB}(v')$. In this case, there would be a chordless cycle $v' - w' - w'' - v'' - v'$. □

LEMMA 3.3. *Let $Q = QA \cup QB$ be any maximal clique of the chordal bi-clique $G_{CA\cup CB}$. Suppose that $QA \subset CA$, $QB \subset CB$, and both are nonempty. Then QA is the smallest set in the set of sets $\{N_{CA}(x) \mid x \in QB\}$, and QB is the smallest set in $\{N_{CB}(v) \mid v \in QA\}$.*

Proof. By Lemma 3.2, the sets $\{N_{CA}(x) \mid x \in QB\}$ are ordered by inclusion. If one of these sets were a proper subset of QA, then Q would not be a clique. (There would be a vertex in QA that is not adjacent to our vertex in QB.) If QA were not one of the sets in $\{N_{CA}(x) \mid x \in QB\}$, then Q would not be maximal. □

ALGORITHM TO GENERATE ALL MAXIMAL CLIQUES OF G OCCURRING IN A BI-CLIQUE
Input: Vertex sets CA, CB, and $G_{A\cup B}$.
Output: A list of cliques that occur in $G_{CA\cup CB}$ and are maximal cliques in $G_{A\cup B}$.

1. Compute $N_{CB}(v)$ for every $v \in CA$. Sort these sets by inclusion and sort the vertices of CA so that vertices with larger neighborhoods in CB are higher. Do the analogous computation for every $v \in CB$. We treat CA and CB as both sets and ordered lists.
2. Find those vertices in the list for CA such that the successor (and therefore every vertex lower in the list) has a strictly smaller neighborhood in CB.
3. Each vertex v chosen in step 2 yields a different maximal clique $Q(v) = QA(v) \cup QB(v)$. $QA(v)$ consists of v and every vertex in CA with the same or a larger neighbor set in CB. Given $QA(v)$, $QB(v)$ is defined by Lemma 3.3. We may also need to include the cliques CA and CB.
4. Decide which maximal cliques of the bi-clique $G_{CA\cup CB}$ are maximal in $G_{A\cup B}$. See the expanded version of this step below.

If $G_{CA\cup CB}$ is a bi-clique having Q as a maximal clique, Q need not be a maximal clique in the graph $G_{A\cup B}$. There may be a vertex $v \notin CA \cup CB$ adjacent to all the vertices of Q but not to all the vertices of CA or CB. To eliminate these nonmaximal cliques, in step 4 of the bi-clique subroutine, we use the following algorithm.

ALGORITHM TO ELIMINATE LOCALLY MAXIMAL CLIQUES THAT ARE NOT GLOBALLY MAXIMAL
Input: A bi-clique $G_{CA\cup CB}$, a list of the maximal cliques in $G_{CA\cup CB}$.
Output: A list of the cliques that are maximal in both $G_{CA\cup CB}$ and $G = G_{A\cup B}$.

4a. For each vertex $v \notin CA \cup CB$, find its neighbors in $G_{CA\cup CB}$.
4b. Try to find the highest vertex $u(CA, v)$ in the sorted list CA such that $N_{CB}(u(CA, v)) \subset N_{CB}(v)$; there may be no vertex in CA with this subset property, in which case $u(CA, v)$ does not exist. Similarly try to find $u(CB, v)$.
4c. For every clique Q that is maximal in $G_{CA\cup CB}$ and has a nonempty intersection with both CA and CB, test if for every $v \notin CA \cup CB$, Q contains at most one of $u(CA, v)$ or $u(CB, v)$. If either $u(CA, v)$ or $u(CB, v)$ does not exist, then v automatically passes the test.
4d. Put every clique Q that passes the test in step 4c on the output list.
4e. If either CA or CB is an input clique, test it for maximality directly. That is, test if any vertex not in the clique is adjacent to all the clique's members. If no vertex can be added, then copy the clique (CA or CB) to the output.

LEMMA 3.4. *The bi-clique algorithm correctly identifies every maximal clique in $G_{CA\cup CB}$ that is also maximal in G.*

Proof. By Lemma 3.3 each maximal clique of $G_{CA\cup CB}$ consists of a (possibly empty) prefix of the sorted list CA and a (possibly empty) prefix of the sorted list CB. One maximal clique contains the entire set CA, and one maximal clique contains the entire set CB. A prefix $P \neq CA$ of CA generates a maximal clique (i.e., is the intersection of a maximal clique with CA) if and only if no vertex of $CA \setminus P$ has as neighborhood in CB the intersection

$$\bigcap_{v \in P} N_{CB}(v).$$

This intersection is precisely the neighborhood in CB of the lowest vertex in P. Moreover, the neighborhood in CB of the first vertex below P contains the neighborhood in CB of all vertices below P. Thus, P generates a maximal clique if and only

if the neighborhood in CB of the lowest vertex in P is different from the neighborhood in CB of the highest vertex not in P. This is the test we implement in steps 1, 2, and 3.

It remains to show that in step 4 we properly eliminate cliques that are not maximal in $G_{A\cup B}$. The test in step 4e for the cliques CA and CB is the standard maximality test. For the other cliques, we use a different test for the sake of efficiency.

Let Q be a maximal clique in $G_{CA\cup CB}$ that has a nonempty intersection with both CA and CB. Let x be the last vertex in the sorted list CA that belongs to Q, and let y be the last vertex in the sorted list CB that belongs to Q. The clique Q is not maximal in $G_{A\cup B}$ if and only if there exists $v\notin CA\cup CB$ such that $Q\subset N(v)$. Since CA and CB are sorted by their neighborhoods, an equivalent rule is that Q is *not* maximal if and only if there exists $v\notin CA\cup CB$ such that

$$N_{CA}(y)\cup N_{CB}(x)=Q\subset N_{CA}(v)\cup N_{CB}(v)\subset N(v).$$

This is equivalent to the condition that

$$N_{CA}(y)\subset N_{CA}(v) \text{ and } N_{CB}(x)\subset N_{CB}(v).$$

Taking the contrapositive, Q is maximal in $G_{A\cup B}$ if and only if for every $v\notin CA\cup CB$,

$$N_{CA}(y)\not\subset N_{CA}(v) \text{ or } N_{CB}(x)\not\subset N_{CB}(v).$$

The condition $N_{CA}(y)\notin N_{CA}(v)$ is satisfied if and only if $u(CB, v)$ does not exist, or $u(CB, v)$ occurs strictly lower than y in the sorted list CB. Similarly, the condition $N_{CB}(x)\notin N_{CB}(v)$ is satisfied if and only if $u(CA, v)$ does not exist, or $u(CA, v)$ occurs strictly lower than y in the sorted list CA. If $u(CB, v)$ occurs strictly lower than y, then Q does not contain $u(CB, v)$; symmetric conditions hold for $u(CA, v)$. Thus Q is maximal in $G_{A\cup B}$ if and only if for every $v\notin CA\cup CB$, one of $u(CA, v)$ and $u(CB, v)$ does not exist or they both exist, but Q contains at most one. This is precisely the test that we implement in step 4c. □

LEMMA 3.5. *The bi-clique algorithm can be implemented to run in $O(\log n)$ time using $O(n^2)$ processors.*

Proof. Step 1 takes $O(\log n)$ time and step 2 takes constant time. Both use $O(n^2)$ processors.

In each clique $Q(v)$ obtained in step 3, $QA(v)$ is a prefix of CA. The size of $QA(v)$ is the rank of v in CA. The set $QB(v)$ is the same as $N_{CB}(v)$. Thus for each clique $Q(v)$, we can compute its size in $O(\log n)$ time with one processor. For each member of $Q(v)$, we can then assign one processor responsible for listing that member in the output. It follows from the existence of a PES that the sum of the sizes of all maximal cliques in a chordal graph with m edges and n vertices is at most $m+n$ [Go80]. Therefore $O(m+n)$ processors suffice for step 3.

Step 4a can be implemented in $O(\log n)$ time using $O(n^2)$ processors by assigning one processor to each pair $\langle v, w\rangle$ such that $v\notin CA\cup CB$ and $w\in CA\cup CB$. Step 4b (for finding $u(CA, v)$) can be implemented as follows. If $N_{CB}(v)=CB$, then $u(CA, v)$ is the first vertex of CA. Otherwise, let $h(CB, v)$ be the highest vertex in CB that is not adjacent to v. If every vertex in CA is adjacent to $h(CB, v)$, then $u(CA, v)$ does not exist. If there is a vertex in CA that is not adjacent to $h(CB, v)$, then $u(CA, v)$ is the highest such vertex in the sorted list CA. The computation of $u(CB, v)$ is analogous. Testing if h and u exist can be done with $O(n)$ processors in $O(\log n)$ time. Finding their values can be done in $O(\log n)$ time with one binary search for each. Since there are $O(n)$ choices for v, the total processor requirement for step 4b is $O(n^2)$. Steps 4c and 4d require $O(\log n)$ time and $O(n)$ processors for each choice of Q; there are at

most n choices of Q. Step 4e requires $O(\log n)$ time and $O(n)$ processors for each vertex not in $CA \cup CB$; there are at most n such vertices. □

Each of the $O(n^2)$ parallel subroutine calls may produce $O(n)$ maximal cliques, although there can be at most n distinct maximal cliques. One way to eliminate duplicates is to sort the clique copies lexicographically. However, this may take $\Omega(\log^2 n)$ time, increasing the total running time to $O(\log^3 n)$. We can eliminate duplicates in $O(\log n)$ time using the following algorithm.

ALGORITHM TO ELIMINATE DUPLICATE MAXIMAL CLIQUES

Input: G, A, B, all maximal cliques in A and B without duplicates, all maximal cliques in G with duplicates.

Output: A list of all maximal cliques of G with no duplicates.

Definitions/Assumptions: Let the maximal cliques in A be numbered $CA_1, CA_2, \cdots, CA_s$. Let the maximal cliques in B be numbered $CB_1, CB_2, \cdots, CB_t$. Assume that each entry on the list of maximal cliques in G comes with a pair of indices indicating which bi-clique generated it. Every step is done in parallel for every (copy of a) maximal clique Q in G.

0. Assume that Q was generated from the bi-clique induced by $CA_i \cup CB_j$.
1. For each $k < i$ test if $Q \cap CA_k$ is a subset of CA_i.
2. For each $l < j$ test if $Q \cap CB_l$ is a subset of CB_j.
3. If no successful k is found in step 1, and no successful l is found in step 2, then output Q.

LEMMA 3.6. *The above algorithm to eliminate duplicate maximal cliques is correct.*

Proof. For each maximal clique R in G, the algorithm implicitly finds the lexicographically smallest pair $\langle k, l \rangle$ such that R is contained in the bi-clique induced by $CA_k \cup CB_l$. If $R \cap A$ is contained in both CA_i and CA_k, then R is generated by the bi-clique induced by $CA_i \cup CB_l$ as well as by the bi-clique induced by $CA_k \cup CB_l$. Thus we can eliminate copies generated by bi-cliques with first part CA_i, for $i > k$; this is the test in step 1. Similarly, if $R \cap B$ is contained in both CB_j and CB_l, then R is generated by the bi-clique induced by $CA_k \cup CB_j$ as well as by the bi-clique induced by $CA_k \cup CB_l$. We eliminate copies generated by bi-cliques with second part CB_j, for $j > l$ in step 2. □

LEMMA 3.7. *It is possible to eliminate duplicate maximal cliques in $O(\log^2 n)$ time using $O(n^4)$ processors or in $O(\log n)$ time using $O(n^5)$ processors.*

Proof. The first pair of resource bounds can be obtained by sorting the cliques lexicographically [Co86] and eliminating any clique that is identical to its predecessor. The second pair of resource bounds come from the algorithm given above. There are at most $O(n^3)$ copies of maximal cliques in the input. Both A and B induce chordal graphs and therefore have at most n (distinct) maximal cliques. Thus in each of steps 1 and 2, we compute at most n set intersections and subset tests for each input clique. Each intersection and subset test can be done in $O(\log n)$ time using n processors. This yields the bounds of $O(\log n)$ time and $O(n^5)$ processors. □

If we allowed our processors to write concurrently, then a subset test could be done with $O(n)$ processors in constant time, and we could eliminate duplicates in $O(\log n)$ time using only $O(n^4)$ processors as follows. If there are more than $2n$ total copies of cliques to consider, partition the cliques into sets of size between $2n$ and $4n$. Within each set, compare all cliques against each other, and within any identical pair, eliminate the copy with the higher number. As there are at most n distinct maximal cliques, these comparisons eliminate at least half of the $O(n^3)$ candidates with which we started. At most $O(\log n)$ rounds of partitioning and comparing leave fewer than

$4n$ candidates that might be distinct. Each of these can then be compared against all the others to eliminate the remaining duplicates.

LEMMA 3.8. *The algorithm to compute all maximal cliques of a chordal graph can be implemented to run in* $O(\log^3 n)$ *time using* $O(n^4)$ *processors or* $O(\log^2 n)$ *time using* $O(n^5)$ *processors. The bounds* $O(\log^2 n)$ *and* $O(n^4)$ *can be achieved simultaneously if the processors are allowed to write concurrently.*

Proof. There are at most $O(\log n)$ levels of recursive calls. At each fixed level, no vertex occurs in the input graph G in more than one call. All the calls together at any level produce at most n cliques, and generate at most $O(n^2)$ calls to the bi-clique subroutine. By Lemma 3.5, all these calls (at step 3) taken together can be implemented to run in $O(\log n)$ time using $O(n^2)$ processors per call, for a total of $O(n^4)$ processors. By Lemma 3.7, duplicate cliques can be eliminated in $O(\log^2 n)$ time using $O(n^4)$ processors or in $O(\log n)$ time using $O(n^5)$ processors. As noted after Lemma 3.7, the bounds $O(\log n)$ time and $O(n^4)$ processors can be achieved simultaneously if processors can write concurrently. Multiplying these time bounds, for a single recursion level, by $\log n$ yields the desired results. □

Since chordal graphs are perfect, the size of the largest clique is the chromatic number.

4. Optimal coloring and representing chordal graphs by trees. Although we just showed how to compute the chromatic number of a chordal graph, obtaining an explicit coloring in NC seems to be nontrivial. In order to do that, we represent a chordal graph as an intersection graph of a family of subtrees. A graph G is the intersection graph of a family $\mathcal{S} = \langle S_1, S_2, \cdots, S_{|V(G)|} \rangle$ of subtrees in a tree T if:

(i) there is a one-to-one correspondence between the subtrees $\mathcal{S}$ and the vertices in G.

(ii) two vertices in G are adjacent if and only if their corresponding subtrees share a node in T.

The choice of T and $\mathcal{S}$ is not unique in general. Two members of $\mathcal{S}$ may be identical.

THEOREM 4.1 ([Go80], [Ga74], [Bu74]). *The following statements are equivalent*:

(i) G is a chordal graph.

(ii) G is the intersection graph of a family of subtrees of a tree.

(iii) There exists a tree $T = (K, E)$ whose vertex set K is the set of maximal cliques of G such that for each $v \in V$ the subgraph induced in T by the set K_v (consisting of all maximal cliques that contain v) is connected, and hence, a subtree.

From now on, we assume that G is connected; otherwise, each component can be processed separately.

Two members of $\mathcal{S}$ corresponding to vertices v and w will be precisely the same when v and w are in exactly the same maximal cliques in G. Even with the three restrictions given in the theorem, the tree T may not be unique. We adopt the convention

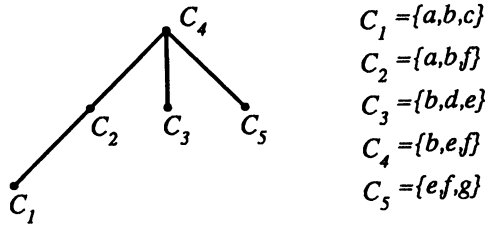


FIG. 2. *One tree representation for the graph of Fig.* 1.

that if a tree node is denoted by a lowercase letter, then its corresponding maximal clique is denoted by the corresponding upper case letter.

The sequential algorithms in [Bu74], [Ga74] that compute a tree representation extend it node by node, but we use a different approach. Mark Novick recently pointed out that a paper by Bernstein and Goodman [Be81] implicitly contains a different algorithm to build a clique tree representation (given all the maximal cliques) that is parallelizable. One advantage of our algorithm is that from it we can easily derive a coloring algorithm.

Our tree-construction algorithm is based on the following idea. Deleting an internal node of a tree makes the tree into a forest with trees of various sizes. A node whose deletion results in a forest where the size of the *largest* component is *minimized* (over all possible choices of one node to delete) is called a *centroid.* Removing a centroid disconnects a tree into components, each including at most half of the nodes [Jo69]. Using properties of G, we decide if a node can be a centroid *without* knowing the exact structure of the tree. We compute a tree representation using a divide-and-conquer method that in each recursive call, finds a centroid of the tree we eventually construct for the induced subgraph of G passed to that call. In each recursion step, a connected input graph is divided into components such that in each component, the number of maximal cliques decreases to at most a constant fraction of the number of maximal cliques in the input. The recursion is based on the following lemmas and observation regarding a tree T representing a chordal graph G.

LEMMA 4.2. *Let C, C_1, and C_2 be maximal cliques of the graph G with corresponding nodes c, c_1, and c_2 in the tree T. If the nodes c_1 and c_2 are in different connected components of $T\backslash c$, and the cliques C_1 and C_2 share a vertex v, then $v \in C$.*

Proof. The subtree corresponding to v consists of all tree nodes that correspond to cliques containing v. Since T is a tree, there is a unique path between nodes c_1 and c_2, and it passes through c. □

LEMMA 4.3. *The removal of any maximal clique C that does not correspond to a leaf in T disconnects the remainder of G.*

Proof. The proof is by contradiction. The removal of the node c disconnects the tree T. Let $T_1, T_2, \cdots, T_l$ denote the connected components (trees) of $T\backslash c$. For each j such that $1 \leqq j \leqq l$, let W_j be the set of vertices of G occurring in a maximal clique represented by a node of T_j. Let $X_j = W_j \backslash C$.

Suppose there were an edge $v - w$ with $v \in X_j$, $w \in X_k$, and $j \neq k$. The edge $v - w$ must belong to some maximal clique. Thus there is an i such that $v, w \in X_i$. If $i = j$, then by Lemma 4.2, w must belong to C, contradicting the choice of w. Similarly if $i = k$, then v must belong to C, contradicting the choice of v. If $i \notin \{j, k\}$, then both v and w must belong to C, a contradiction. This shows that no two of the sets $X_1, X_2, \cdots, X_l$ have adjacent vertices. Since c is an internal node with degree at least 2, $l \geqq 2$, and G must be disconnected. □

Observation 4.4. If removing the maximal clique Q from G leaves $G\backslash Q$ with components $C_1, C_2, \cdots, C_l$, then it is possible to build a tree T so that for each i, with $1 \leqq i \leqq l$, the maximal cliques in $C_i \cup Q$ correspond to the nodes in one component of the forest $T\backslash q$ plus one node for the clique Q.

Our recursion divides the problem into subproblems, each of which includes, at most, half of the maximal cliques (plus one) of the original problem, by finding a centroid node in the tree. At the bottom level we have a graph with exactly two overlapping maximal cliques whose tree is simply two adjacent nodes. We compute a centroid c as follows: for every maximal clique Q, count in parallel how many maximal cliques are in each connected component of $G\backslash Q$ with a copy of Q added to each

component. Let $p(Q)$ be the maximum number of maximal cliques in any of these components. We choose c to be that q such that $p(Q)$ is minimized over all choices of maximal clique Q.

To bound the depth of the recursion, we need to observe that every maximal clique in each connected component of $G \backslash Q$ is either a maximal clique of G or is contained in a maximal clique of G. Thus the number of maximal cliques in each connected component is cut by at least half not counting the centroid clique.

ALGORITHM TO COMPUTE A TREE-AND-SUBTREES REPRESENTATION (SUMMARY)
Input: A connected chordal graph G and the list of maximal cliques of G.
Output: A tree representation for G.

1. If G has more than two maximal cliques, then find a maximal clique that corresponds to a centroid c as explained above.
2. Let the connected components of $G \backslash C$ be $C_1, C_2, \cdots, C_l$. Compute recursively the tree representation of $C_i \cup C$ for every i.
3. Let $T_1, T_2, \cdots, T_l$ be the trees that result from the recursion. $T = \cup_{i=1}^{l} T_i$, where the T_i are disjoint except for c. The subtree corresponding to each vertex is defined in Theorem 4.1(iii).

In the base case of the recursion, we have two maximal cliques whose intersection is not empty. Their corresponding tree consists of two adjacent nodes. The correctness of the algorithm follows from Lemmas 4.2 and 4.3 and Observation 4.4.

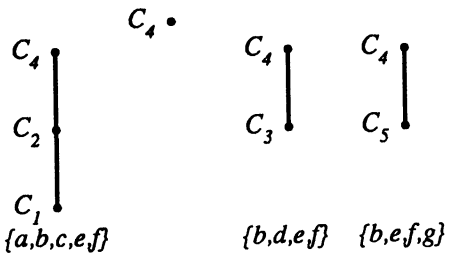


FIG. 3. *Building the tree of Fig. 2 by using C_4 as the centroid clique.*

To keep the running time and number of processors down, we do not recompute the maximal cliques and connected components from scratch in every recursive call. Before making the recursive call in step 2, we compute the maximal cliques in each graph of the form $C_i \cup C$ with the following algorithm.

ALGORITHM TO UPDATE LISTS OF MAXIMAL CLIQUES
DEFINITIONS. Let G_j be the graph considered in the current call. Let c be the centroid we chose in step 1 above. Let the connected components of $G_j \backslash C$ be $C_1, C_2, \cdots, C_l$.
Input: G_j, C, $C_1, C_2, \cdots, C_l$, a list of maximal cliques in G_j.
Output: For each component C_i a list of maximal cliques in $C \cup C_i$.

1. Put C on each output list.
2. For each maximal clique $Q \neq C$ of G_j *in parallel*: find a vertex $v(Q)$ that does not belong to C.
3. For each maximal clique Q, let $C_i(Q)$ be the unique connected component among $C_1, C_2, \cdots, C_l$ that contains $v(Q)$.
4. For each maximal clique $Q \neq C$ of G_j *in parallel*: put Q on the list for $C_i(Q)$ (and on no other list).

LEMMA 4.5. *The algorithm to update the lists of maximal cliques is correct.*

Proof. Let $Q \neq C$ be a maximal clique in G_j. Because C separates G_j, the vertices in Q/C must belong to exactly one connected component in $G_j \backslash C$. That component, $C_i(Q)$, is identified in step 3, and Q must be a clique in $C_i(Q) \cup C$.

If there were a vertex $w \in (C_i(Q) \cup C) \backslash Q$ adjacent to every vertex of Q, then Q would not be maximal in G_j. This shows that Q must be maximal in $C_i(Q) \cup C$. Since Q is arbitrary, every maximal clique output is legitimate. We now show that no maximal clique is omitted. Suppose, seeking to establish a contradiction, that there is a maximal clique $R \neq C$ in some $C_k \cup C$ such that R is *not* maximal in G_j. Since R is not maximal in G_j, there must be a vertex $x \notin C_k \cup C$ such that every vertex is R is adjacent to x. Since $R \neq C$, the graph induced by $R \cap C_k$ is nonempty. The previous two assertions contradict the assumption that C_k is an entire connected component of $G_j \backslash C$. □

In step 1 of the main algorithm, we compute for each clique Q the connected components of $G \backslash Q$. In the first call, we use the algorithm in [Hi79]. In subsequent calls, we compute the connected components more efficiently by reusing previous computations.

ALGORITHM TO UPDATE CONNECTED COMPONENTS IN STEP 1.

Definitions/Assumptions. Let G_j be the current input graph. Let G_{j-1}, a supergraph of G_j, be the input graph for the recursive call that preceded this one. Let C_{j-1} be the centroid clique that we chose for G_{j-1}. Each component can be represented as a sorted subset of the vertices of v. For each graph G_j, it is also useful to have a table showing to which component each vertex belongs.

Input: The graphs G_j and G_{j-1}; for each maximal clique Q in G_{j-1}, a list of connected components of $G_{j-1} \backslash Q$ (this was computed in the previous recursive call); a list of maximal cliques in G_j, and the centroid clique C for G_{j-1}.

Output: For each maximal clique $R \neq C$ in G_j, a list of connected components of $G_j \backslash R$.

1. For each maximal clique $R \neq C$ in G_j, let $L_{j-1}(R)$ be the list of components of $G_{j-1} \backslash R$.
2. For each R in parallel; let $w(R)$ be a vertex in $C \backslash R$.
3. For each R *in parallel*: let $H_{j-1}(R)$ be the component in $L_{j-1}(R)$ that contains $w(R)$.
4. For each R *in parallel*: put every component on the list $L_{j-1}(R)$ except $H_{j-1}(R)$ on the output list for R.
5. For each R *in parallel*: compute $H_{j-1}(R) \cap G_j$ and add it to the output list for R.

LEMMA 4.6. *The algorithm to update connected components is correct.*

Proof. Let T be a tree for G_{j-1} satisfying the conditions of Observation 4.4. Remove a node c from T and reconnect c to each tree in the resulting forest. This yields a forest of trees $T_1, T_2, \cdots$ that are disjoint except for the common node c. Using Observation 4.4, define a bijection ϕ_c from the set of trees in the forest $T \backslash c$ to the set of connected components $G_{j-1} \backslash C$. Let T_i be one of the trees, and let W_i be the union of maximal cliques whose corresponding nodes lie in W_i. Then $\phi_c(T_i) := W_i \backslash C$ is a connected component of $G_{j-1} \backslash C$.

Let c be the centroid we chose for G_{j-1}. Let a maximal clique $R \neq C$ in G_j be given. Assume, without loss of generality, that the node r lies in tree T_1 defined above. Let $d(r)$ be the degree of node r in T; r has the same degree in T_1. Thus removing r from either T or T_1 would leave a forest of $d(r)$ trees. Moreover, the only difference between these two forests is in the tree containing c. Thus the bijection ϕ_r, which defines the connected components of $G_{j-1} \backslash R$, also yields $d(r) - 1$ of the connected components of $G_j \backslash R$. In each graph, there is one component left over. The one

component that changes is found in step 3; the components that remain the same are reported in step 4.

The component that changes in going from $G_{j-1}\backslash R$ to $G_j\backslash R$ is the one containing $C\backslash R$. In going from T to T_1, we deleted all nodes representing cliques that did not intersect $G_j\backslash C$. Thus the component in $G_{j-1}\backslash R$ that included $C\backslash R$, loses all vertices not in G_j when converting to G_j and keeps the rest. This is what we compute in step 5.

Since $R \neq C$ was an arbitrary maximal clique of G_j, the algorithm works correctly for each candidate separating clique in G_j. □

LEMMA 4.7. *The algorithm to build a tree representation can be implemented to run in time* $O(\log^2 n)$ *using* $O(n^3)$ *processors (assuming that all the maximal cliques are part of the input).*

Proof. To update the maximal cliques, we use at most one processor per vertex occurring in a clique. As remarked in the proof of Lemma 3.5, the sum of all the maximal clique sizes is bounded by $m+n$. Each step of the maximal clique updating algorithm takes $O(\log n)$ time.

The algorithm to update connected components can be analyzed as follows. Step 1 can be done in constant time by table lookup. In step 2, we may need as many as $m+n$ processors to find all the values $w(R)$ in $O(\log n)$ time. Step 3 can be done in constant time by table lookup. In step 4, we just copy part of the input to the output. The size of the output may be $O(n)$ for each R, so we need $O(n^2)$ processors and $O(\log n)$ time. In step 5 we compute the intersection of two sets (represented by sorted lists or bitmaps) of size at most n. This can be done in $O(\log n)$ time using n processors for each R, for a total of $O(n^2)$ processors.

Finally, we analyze the main algorithm. The first time we do the connected components computations in step 1, we use the algorithm in [Hi79]. Since we do at most n such computations in parallel, $O(n^3)$ processors suffice.

The maximum depth of recursion is $O(\log n)$. Assume that the recursive calls are synchronized by level in the sense that for any level i, all calls at level i are started simultaneously, and no call at level i is resumed (after its recursive calls) until all calls one level deeper are complete. We show that for any recursion level i, $O(\log n)$ time and $O(n^3)$ processors suffice for all the calls at level i together. Each input graph at level i that has more than two maximal cliques includes a maximal clique that belongs to no other input graph at level i. Since a chordal graph can have at most n maximal cliques, the number of calls at level i in which the input graph has three or more maximal cliques is bounded by n. Each call in which the input contains exactly two maximal cliques corresponds to a *distinct* edge in the final; thus, there can be at most $n-1$ such calls on any level. In all, there can be at most $2n-1$ parallel calls at any recursion level.

We showed above that updating the maximal cliques and connected components can be done in $O(\log n)$ time and $O(n^2)$ processors per call. Thus the total cost per level of these subroutines is $O(\log n)$ time and $O(n^3)$ processors. The only remaining steps to analyze are the selection of the centroid and the union of the trees.

Once a maximal clique has been chosen as a centroid, it cannot be chosen again because its removal cannot disconnect graphs deeper in the recursion. Thus for any level i, the sets of candidate centroids in any two calls at that level are disjoint. Since there are at most n maximal cliques, there are at most n candidate centroids among all calls at any recursion level. For each candidate centroid q and each maximal clique $R \neq Q$, we have to decide which component of $G\backslash Q$ contains $R\backslash Q$. This can be done in $O(\log n)$ time with $O(n)$ processors by finding one vertex of $R\backslash Q$ and then finding out which component of $G\backslash Q$ contains it. Altogether, at level i of recursion, there are

at most n candidate centroids and at most n cliques per candidate. Thus we need $O(n^3)$ processors to find out which clique belongs to which component. In order to sum the results and find the minima, we again need $O(n^3)$ processors.

To combine the trees, we identify the copy of c in each tree and append the adjacency lists for copies of c together. The adjacency lists for the tree nodes need not be sorted, so the copying and appending can be done in $O(\log n)$ time using $O(n)$ processors. □

The coloring algorithm proceeds using exactly the same structure of recursive calls as the tree-construction algorithm. For coloring, the tree is primarily a conceptual tool used to design the algorithm, and there is no need to actually build a tree. The coloring algorithm uses the same subroutines to update maximal cliques and connected components as the tree-construction algorithm.

SUMMARY OF ALGORITHM TO COMPUTE AN OPTIMAL COLORING

1. Find a maximal clique Q that has the centroid property.
2. Let the connected components of $G \setminus Q$ be $C_1, C_2, \cdots, C_l$. Compute recursively an optimal coloring for $C_i \cup Q$ for every i.
3. Merge the colorings so that each copy of Q is colored consistently. Since Q is a clique, each of its vertices must have a different color, and we need only to rename the color classes relative to one of $C_i \cup Q$.

The base case of the recursion is the same as in constructing the tree. We color one clique first and then the other.

LEMMA 4.8. *The coloring algorithm is correct.*

Proof. Assume as inductive hypothesis that the colorings of $C_1 \cup Q$, $C_2 \cup Q, \cdots, C_l \cup Q$ are legal and optimal. By Lemma 4.3, the only edges in both $C_i \cup Q$ and $C_j \cup Q$, for $i \neq j$ have an endpoint in Q. Therefore Q can always be colored the same way, and the combined coloring is legal and optimal. □

LEMMA 4.9. *The coloring algorithm can be implemented in $O(\log^2 n)$ time using $O(n^3)$ processors if the maximal cliques are included as part of the input.*

Proof. The analysis of the subroutines and steps 1 and 2 is identical to that in the proof of Lemma 4.7 analyzing the tree-construction algorithm.

We can implement step 3 as follows. Suppose that the graphs $C_i \cup Q$ are numbered in such a way that $C_1 \cup Q$ has the largest chromatic number. Take the coloring of $C_1 \cup Q$ to be fixed. For each other $C_i \cup Q$, compute a mapping Γ_i, represented as a table, from colors to colors. Every vertex of Q must get a different color from all the other vertices of Q. If $v \in Q$ has color a_1 in $C_1 \cup Q$ and color a_2 in $C_i \cup Q$, then in the table for $C_i \cup Q$, set $\Gamma_i(a_2) := a_1$; any vertex that used to be colored a_2 will be colored a_1. Let A_i be the set of colors used in $C_i \cup Q$ that are not used to color vertices of Q. Define Γ_i on A_i so that it maps A_i injectively to (arbitrary colors in) A_1. A satisfactory mapping Γ_i can be constructed in $O(\log n)$ time with $O(n)$ processors for each graph $C_i \cup Q$. Finally, recolor each vertex in $C_i \cup Q$ by applying the mapping Γ_i. Recoloring can be done in constant time using at most $O(n)$ processors per graph. Since there are at most $O(n)$ parallel calls at any given level of recursion, the total processor requirement for step 3 is $O(n^2)$. □

5. Perfect elimination scheme, independent set, clique cover. We use the tree-and-subtrees representation constructed in § 4 to give NC algorithms for finding a perfect elimination scheme, a maximum unweighted or weighted independent set, and a minimum clique cover in a chordal graph. The clique cover follows directly from the PES and the unweighted independent set. The basic idea for the PES and the independent sets is to process the tree in stages: in each stage, we first process and then delete

all terminal branches of the tree in parallel. A *terminal branch* is any path consisting of tree nodes of degree two or less and containing a leaf. After deleting the terminal branches, the new tree has at most half as many leaves as the old tree; therefore, there are at most $O(\log n)$ stages.

To process the terminal branches, we first need to identify them and number the nodes on them. On each terminal branch we number the nodes 1, 2, $\cdots$ starting at one leaf. Terminal branches can be identified and numbered with several applications of path doubling. Details are given in the technical report [Na87].

LEMMA 5.1 [Na87]. *The terminal branches can be identified and numbered in* $O(\log n)$ *time using* $O(n^2)$ *processors.*

To process one terminal branch in polylogarithmic time, we use the following observation.

Observation 5.2 [Gi64], [Go80]. If a chordal graph G has a tree-and-subtrees representation T, $\mathcal{S}$ in which every tree node corresponds to a maximal clique, every subtree in $\mathcal{S}$ corresponds to the set of maximal cliques containing a particular vertex, and T is a *path*, then G is an interval graph. In an interval graph the vertex corresponding to the interval with the leftmost right endpoint is simplicial.

Thus the cliques represented by nodes on any particular terminal branch induce an interval graph. Kozen, Vazirani, and Vazirani explain how to build an interval model for an interval graph [Ko85]. However, we want to build the models in such a way that the partial perfect elimination schemes for each of the interval graphs induced by the terminal branches fit together in a PES for the entire graph. The PES algorithm implicitly relies on the fact (proved below) that the following algorithm yields a representation of the interval graph corresponding to a terminal branch.

ALGORITHM TO CONSTRUCT AN INTERVAL MODEL FOR THE VERTICES ON A TERMINAL BRANCH

Input: A chordal graph G, a tree representation T, a terminal branch B of T, the set $\mathcal{C}(B)$ of maximal cliques corresponding to nodes on B.

Definition: Let $V(B)$ be the set of vertices of G occurring in some member of $\mathcal{C}(B)$.

Output: An interval model for the interval graph $G_{V(B)}$.

1. Let the size of $\mathcal{C}(B)$ be p_B. Index the members of $\mathcal{C}(B)$, by 1, 2, $\cdots$, p_B in the order in which the corresponding nodes appear on B starting at a leaf clique.
2. For each $v \in V(B)$, let $J(v)$ be the interval corresponding to v. Let the left endpoint of $J(v)$ be the lowest index of a maximal clique containing v. If v appears in a maximal clique of G that is not in $\mathcal{C}(B)$, then let the right endpoint of $J(v)$ be $p_B + 1$. Otherwise, let the right endpoint of $J(v)$ be the highest index of a maximal clique containing v.

LEMMA 5.3. *The algorithm to construct an interval model for* $G_V(B)$ *is correct.*

Proof. By Theorem 4.1, the nodes of B corresponding to maximal cliques containing a given vertex v induce a path. Therefore the interval $J(v)$ contains the index of a maximal clique $K \in \mathcal{C}(B)$ if and only if $v \in K$. Let distinct vertices v and $w \in V(B)$ be given. If v is adjacent to w, then they must both occur in some maximal clique $K \in \mathcal{C}(B)$. The index of K must be in both $J(v)$ and $J(w)$, so the intervals overlap. If v and w are not adjacent, they do not occur together in any maximal cliques, and the intervals $J(v)$ and $J(w)$ are disjoint. □

By Observation 5.2, the set $V(B)$ sorted according to the right endpoints of the intervals $J(v)$ is a PES for $G_{V(B)}$. For any terminal branch B, let $U(B)$ be the set of vertices that are in no maximal cliques represented by tree nodes outside B.

Observation 5.4. A vertex in $U(B_i)$ can never be adjacent to a vertex in $U(B_j)$ if B_i and B_j are different terminal branches. □

Since the vertices in $U(B)$ occur in no cliques corresponding to tree nodes outside B, a vertex in $U(B)$ is simplicial in G if and only if it is simplicial in $G_V(B)$. The PES defined by sorting the intervals produced by the algorithm above puts all vertices of $U(B)$ first. Thus there must be a PES for all of G that puts $U(B)$ first.

PERFECT ELIMINATION SCHEME ALGORITHM

Input: A chordal graph G, the maximal cliques of G, a tree representation T in which every node corresponds to a maximal clique.

Assumption: The cliques are represented by a vertex versus a maximal clique incidence matrix.

Output: A PES for $V(G)$.

1. In parallel, find all the terminal branches and number them arbitrarily, B_1, $B_2, \cdots, B_k$.
2. In each B_i, index the nodes (maximal cliques) 1, 2, $\cdots$ starting at a leaf.
3. For each B_i *in parallel*: Find the set $U(B_i)$.
4. For each B_i *in parallel*: Sort the vertices in $U(B_i)$ according to the highest index of a maximal clique in which they occur. Vertices whose highest indexed clique is lower come first. This sorted order is a PES for $G_{U(B_i)}$.
5. Append the vertices in the perfect elimination schemes for $U(B_1)$, $U(B_2), \cdots, U(B_k)$.
6. Delete every node in a terminal branch from T to get a new tree T'. Delete every vertex in every $U(B_i)$ to get a new graph G'. If T' is not empty, compute recursively a PES for the remaining nodes and append it to the partial elimination scheme computed so far.

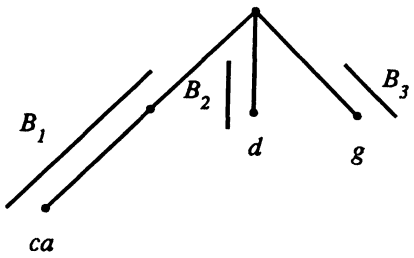


FIG. 4. *Terminal branches B_1, B_2, B_3 for the tree of Fig. 2 and their perfect elimination schemes.*

LEMMA 5.5. *The perfect elimination scheme algorithm is correct.*

Proof. As remarked after Observation 5.4 the sorted order on $U(B_i)$ produced in step 4 is not only a PES for $G_{U(B_i)}$, but it is also the beginning of a PES for all of G. From Observation 5.4, it follows that the perfect elimination schemes for the different branches can be put together in any order. The tree obtained by deleting all tree nodes in $B_1, B_2, \cdots, B_k$ is a tree representation having the desired clique properties for the graph induced by the vertices $V(G)\backslash(U(B_1) \cup U(B_2) \cup \cdots \cup U(B_K))$. □

Each ordering of the branches $B_1, B_2, \cdots, B_k$ gives a different PES. For example in Fig. 4, the perfect elimination schemes for the three branches are *ca*, *d*, and *g*. When the branches are numbered as shown, the complete PES is *cadgbef*. A different order of branches would result in a different, equally correct scheme.

LEMMA 5.6. *The perfect elimination scheme algorithm can be implemented to run in* $O(\log^2 n)$ *time using* $O(n^2)$ *processors (given all the maximal cliques and a tree representation).*

Proof. Steps 1 and 2 can be implemented in $O(\log n)$ time using $O(n^2)$ processors by Lemma 5.1.

If the entire tree consists of one branch B, then $U(B) = V(G)$. Otherwise, step 3 can be implemented in $O(\log n)$ time using $O(m)$ processors by assigning one processor to every pair $\langle v, Q\rangle$, where v is in the maximal clique Q, and the node q is on a terminal branch (B_i). The processor tests whether v also occurs in the maximal clique whose node is one further away from the leaf. If v does not occcur in that clique, then it is part of $U(B_i)$.

If v is put in $U(B_i)$ by the processor assigned to $\langle v, Q\rangle$ in step 3, then Q is the highest indexed clique in which it occurs. The clique indices can be sorted in step 4 in $O(\log n)$ time using $O(n)$ processors [Co86].

For step 5, we need to compute prefix sums of the sizes of $U(B_i)$ in order to decide where the PES for each branch starts in the final output sequence. This can be done in $O(\log n)$ time using $O(n)$ processors.

Step 6 requires copying every part of T and G and every maximal clique that remains for the recursive call. To copy T, assign one processor to each entry on an adjacency list for T. Keep the entry if and only if it is not a node on a terminal branch. Ignore all adjacency lists for nodes on terminal branches. The surviving part of the graph G can be copied similarly. If we represent G by an adjacency matrix, this operation requires $O(n^2)$ processors. The maximal cliques that survive are precisely those represented by nodes that are not on terminal branches.

As remarked at the beginning of § 5 an algorithm that in each stage prunes all terminal branches in a tree with at most n nodes is guaranteed to terminate after $O(\log n)$ stages. Therefore the PES algorithm requires $O(\log^2 n)$ time and $O(n^2)$ processors. □

Next we turn to maximum independent sets. To find an unweighted maximum independent set, we rely on the result of Gavril [Ga72] that the lexicographically first maximal independent set, using the vertex numbering given by a PES, is a maximum independent set. Our parallel algorithm computes a lexicographically first maximal independent set with respect to the PES computed by the previous algorithm.

The special case of maximum independent sets in interval graphs, even if the nodes have weights, can be solved by an algorithm of Helmbold and Mayr [He86] (or a similar algorithm of Bertossi and Bonuccelli [Be87]), which we discuss in more detail after showing how to use it for the unweighted independent set. In the unweighted case, the algorithms in [He86], [Be87] find the lexicographically first maximal independent set, which will be exactly what we need to make the independent set for the entire chordal graph lexicographically maximal, that is, maximum by Gavril's result.

If I is an independent set in G, and B is a branch of T, let $R(B, I)$ be $U(B)\backslash\{v \in V(G) \mid v \in N(I)\}$, where $N(I)$ is the set of neighbors of vertices in I.

UNWEIGHTED MAXIMUM INDEPENDENT SET ALGORITHM

Input: A chordal graph G, the set of maximal cliques of G, a tree representation T for G.

Output: An unweighted maximum independent set for G.

1. Set $I := \varnothing$.
2. In parallel, find all the terminal branches and number them arbitrarily, B_1, $B_2, \cdots, B_k$.

3. In each B_i, index the nodes (maximal cliques) 1, 2, $\cdots$ starting at a leaf.
4. For each B_i *in parallel*: Find the set $U(B_i)$.
5. For each B_i *in parallel*: Find the set $R(B_i, I)$.
6. For each branch B_i, use the algorithm in [He86] to compute a lexicographically first maximal independent set on $R(B_i, I)$ with respect to the lexicographic ordering given by the PES on $U(B_i)$.
7. Add the independent sets computed in step 6 to I.
8. Delete every node in a terminal branch from T to get a new tree T'. Delete every vertex in every $U(B_i)$ to get a new graph G'. Set $T := T'$. Do not overwrite the input copy of G, since we need it to compute the sets $R(B_i, I)$ in step 5.
9. If the new T is empty, return I; otherwise, go back to step 2 with the new values of T and I.

LEMMA 5.7. *The unweighted maximum independent set algorithm is correct.*

Proof. Let $v_1, v_2, \cdots, v_n$ be the PES we would get ordering the terminal branches the same way throughout both the PES and independent set algorithms. Partition $V(G)$ into sets $U_1, U_2, \cdots, U_l$ of vertices that occur consecutively in the PES so that each set U_j consists of precisely the vertices coming from one terminal branch. That is, $U_1 = U(B_1)$, $U_2 = U(B_2)$, and so on.

We prove by induction on j that the unweighted maximum independent set algorithm chooses a lexicographically first maximal independent set among the vertices in $U_1 \cup U_2 \cup \cdots \cup U_j \cup \cdots \cup U_l$. It then follows from Gavril's [Ga72] observation mentioned above that the algorithm computes a maximum independent set.

For the base case, consider $U_1 = U(B_1)$. The algorithm in [He86] yields a lexicographically first maximal independent set on the interval graph G_{U_1}. Let $I_j := I \cap (U_1 \cup U_2 \cup \cdots \cup U_j)$. Assume as the inductive hypothesis that I_{j-1} is the lexicographically first maximal independent set for the vertices in $U_1 \cup U_2 \cup \cdots \cup U_{j-1}$. Any vertex $v \in U(B_j)$ that is adjacent to a vertex w in I_{j-1} should not occur in the output. By Observation 5.4, any such vertex w must be in some $U(B_k)$, where B_k is a terminal branch that is processed before B_j and *not* in parallel with B_j. Thus the set $R(B_j, I)$ computed in step 5 contains precisely the vertices in $U(B_j)$ that are not adjacent to any vertex in I_{j-1}. The Helmbold and Mayr algorithm computes a lexicographically first maximal independent set for $R(B_j, I)$. Adding this set to I_{j-1} to obtain I_j ensures that I_j is indeed the lexicographically maximal independent set for $U_1 \cup U_2 \cup \cdots \cup U_j$. It follows by induction on j that the final independent set $I_l = I$ is the lexicographically first maximal independent set for the entire graph G. □

Before analyzing the resource requirements of our unweighted independent set algorithm, we describe the Helmbold and Mayr algorithm in more detail for both the unweighted and weighted cases. Let H be an interval graph with vertices $u_1, u_2, \cdots, u_p$ ordered by the left to right order of the right endpoints of the corresponding intervals, $i_1, i_2, \cdots, i_p$. Construct a directed graph $D(H)$ on the same vertex set as follows. Put in the arc $u_j \to u_k$ if $j < k$ and the right endpoint of i_j comes before the left endpoint of i_k, so that the intervals i_j and i_k do not overlap. The edge set of $D(H)$ is a partial order, often called the *interval order*, associated with $\{i_1, i_2, \cdots, i_p\}$ [Go80].

Helmbold and Mayr observed that the lexicographically first maximal unweighted independent set of H contains exactly the vertices on the lexicographically first maximal path in $D(H)$. At most one outgoing arc from each vertex can possibly occur in the maximal path, namely the arc to the vertex with the lowest index (subscript). Thus the computation can be restricted to a subgraph $D'(H)$ of $D(H)$ where every vertex has outdegree at most 1; the *outdegree* of a vertex is the number of outgoing arcs

incident to it. The lexicographically first maximal paths of $D'(H)$ and of $D(H)$ coincide. The subgraph $D'(H)$ can be computed in $O(\log n)$ time using $O(V(D(H))^2)$ processors. Since each vertex of $D'(H)$ has outdegree 1, one can find the lexicographically first maximal path by path doubling in $O(\log n)$ time using a linear number of processors.

In the weighted independent set algorithm, we will need to allow H to be weighted, and to find the heaviest independent set containing only vertices in $\{u_1, u_2, \cdots u_r\}$ for different values of $r \leqq p$. We build $D(H)$ as before with two minor differences. First, we add a source vertex u_0 and an arc $u_0 \to u_j$ for each j, $1 \leqq j \leqq p$, having as its weight the weight of u_j in H. Second, we give arc $u_j \to u_k$ the weight that u_k has in H.

A maximum weight independent set in H corresponds to a maximum weight path in $D(H)$. Helmbold and Mayr do the computation by applying a *max-plus closure* to the weighted adjacency matrix for $D(H)$. Max-plus closure produces a $(p+1) \times (p+1)$ matrix, and is algebraically analogous to multiplying p copies of the weighted adjacency matrix for $D(H)$ together, except that the usual operations of addition and multiplication are replaced by max and addition, respectively. The weight of a heaviest path will be the largest number in the final result. The weight of the heaviest path using only vertices in $\{u_0, u_1, \cdots, u_r\}$ for some $r < p$, is given by the maximum among the entries in the submatrix of the final result induced by rows $0, 1, \cdots, r$ and columns $0, 1, \cdots, r$.

LEMMA 5.8. *The unweighted independent set algorithm for chordal graphs can be implemented to run in $O(\log^2 n)$ time using $O(n^2)$ processors (if the maximal cliques and a tree representation are included in the input).*

Proof. Step 1 can be implemented in constant time using $O(n)$ processors. Steps 2, 3, and 4 are identical to steps 1, 2, and 3 of the PES algorithm. In step 5, we compute $R(B_i, I)$. This can be done by checking all the neighbors of each vertex in $U(B_i)$, assigning one processor per edge. If $v \in U(B_i)$ has a neighbor w that is already in I, then v should not be in $R(B_i, I)$. This computation takes $O(\log n)$ time and $O(m)$ processors. In the unweighted case, the Helmbold and Mayr algorithm used in step 6 takes $O(\log n)$ time and use $O(m)$ processors. The augmentation of I in step 7 can be done in constant time using $O(n)$ processors. Step 8 is identical to step 6 of the PES algorithm, except that in this case we do not overwrite the original copy of G.

In each iteration of the loop in steps 2 through 9, we process and delete all terminal branches. Thus, there are $O(\log n)$ iterations, and the total running time is $O(\log^2 n)$. □

Our algorithm for maximum weighted independent sets in a chordal graph is a parallelization of the following sequential algorithm of Frank [Fr75] (see also [Lo85]). Let $w(v)$ be the weight of vertex v.

SEQUENTIAL MAXIMUM WEIGHTED INDEPENDENT SET ALGORITHM
Input: A chordal graph G with vertex weights and numbered as in a PES.
Output: A maximum weight independent set for G.

1. Let v be the first vertex in the PES.
2. Delete all vertices $u \in N(v)$ such that $w(u) \leqq w(v)$.
3. For all remaining vertices $u \in N(v)$, set $w(u) := w(u) - w(v)$.
4. Recursively find a maximum weight independent set in the remaining graph with the updated weights.
5. If no vertex in $N(v)$ is in the set computed in step 4, then add v to the weighted independent set.

To implement this algorithm in parallel, we again process all terminal branches of the tree in parallel, but the computation for each branch is more complicated than

in the unweighted case.

SKETCH OF A PARALLELIZATION OF FRANK'S WEIGHTED INDEPENDENT SET ALGORITHM

Input: G, all maximal cliques, a tree representation, a weight function w on $V(G)$.
Output: The maximum weight independent set that Frank's sequential algorithm would compute, assuming the vertices are numbered according to the PES that our parallel algorithm would compute.

1. If the tree is a path, use the algorithm of Helmbold and Mayr to compute a maximum weighted independent set on the interval graph G. Otherwise, proceed to the following steps.
2. Identify and number the terminal branches $B_1, B_2, \cdots, B_k$.
3. Index the nodes on each terminal branch as in the previous algorithms.
4. For each branch B_i *in parallel*: Compute the set $U(B_i)$.
5. For each $U(B_i)$ *in parallel*: Find its subset $S_2(B_i)$ of vertices that would survive step 2 of the sequential algorithm if it were to process the vertices in $U(B_i)$ in the same order as in the PES. Ignore any vertices of weight less than or equal to zero.
6. Update the weights of all vertices, as in step 3 of the sequential algorithm.
7. Delete the terminal branches and recursively compute the maximum weight independent set of what remains.
8. For each $S_2(B_i)$ computed in step 5, decide which vertices can be added to the independent set, as in step 5 of the sequential algorithm.

We now elaborate a bit on steps 5, 6, and 8 that are especially sketchy. Suppose $U(B_i)$ contains the vertices $v_{i_1}, v_{i_2}, \cdots, v_{i_s}$, which are consecutive in the PES we construct. We need to decide for each of those vertices whether it would get deleted in step 2 of Frank's sequential algorithm. To do this, we use the weighted version of the Helmbold and Mayr algorithm. Recall that we mentioned that the Helmbold and Mayr algorithm also allows us to determine the maximum weight independent set of any of the subgraphs H_j induced by $\{v_{i_1}, v_{i_2}, \cdots, v_{i_j}\}$ for each j, $1 \leqq j \leqq s$. The vertex v_{i_j} is eliminated in step 2 of Frank's algorithm if and only if there is a maximum weight independent set in H_j that does not contain v_{i_j}. This can be determined by comparing the maximum weight for H_{j-1} and for H_j. An analogous computation works for all terminal branches.

In step 6, we update the weights of all the vertices as in step 4 of Frank's algorithm. For each B_i and each vertex $v \in S_2(B_i)$, the fact that v survives step 5 forces us to update the weights of its neighbors not yet processed. If u is one of its neighbors, we want to change the "current" weight of u by the following assignment: $w(u) := w(u) - w(v)$. Each assignment is an arithmetic expression using one subtraction and no additions, divisions, or multiplications. We can compute all the assignments symbolically and order them as we would have done them in the sequential algorithm. All the assignments required by vertices in $S_2(B_i)$ are listed before those for $S_2(B_{i+1})$. The list of assignments is a straight-line program for the weights involving no multiplications or divisions. By evaluating this program, we can compute the weight of each vertex after all vertices in $U(B_1), U(B_2), \cdots, U(B_k)$ have been processed as far as step 7. The program can be evaluated using the methods of Miller and Reif [Mi85]. In their framework, our straight-line program becomes an expression tree. One subtlety is that we may need $O(n)$ processors at each of the $O(n)$ nodes in the expression tree because each time we consider one vertex, we may change the weight of any of its neighbors.

To do step 8 for $S_2(B_i)$, we first delete any vertex in $S_2(B_i)$ adjacent to any vertex in the maximum weight independent set returned by the next deepest recursive call.

What remains is a subset of $S_2(B_i)$ that again induces an interval graph. We number the vertices in the reverse order to that in which they were processed. This order corresponds to the fact that in the sequential algorithm the vertices are chosen in the PES order at step 1, and considered in reverse PES order at step 5. Once we have the remaining vertices in reverse order, we run the unweighted version of the Helmbold and Mayr algorithm to find the lexicographically maximal independent set among them. This set contains precisely those vertices that would be chosen in step 5 of the sequential algorithm.

Summarizing the above discussion, gives us Lemma 5.9.

LEMMA 5.9. *The parallel algorithm to compute a maximum weight independent set is correct.*

Proof. We process the vertices in the order of a PES as is done in Frank's sequential algorithm. Vertices that come from two different terminal branches that are processed in parallel have no effect on each other in the sequential algorithm. The fact that step 5 of the parallel algorithm correctly simulates the test in step 2 of the sequential algorithm follows from the correctness of the algorithm in [He86]. In step 6, we do the same weight updates as in step 3 of the sequential algorithm. In step 8, we consider adding vertices in reverse order exactly as in the sequential algorithm. The correctness of our implementation of step 8 follows from correctness of the algorithm in [He86]. □

LEMMA 5.10. *The maximum weight independent set algorithm can be implemented to run in $O(\log^3 n)$ time using $O(n^4)$ processors (given the maximal cliques and a tree representation).*

Proof. Step 1 can be implemented in $O(\log^2 n)$ time using $O(n^3)$ processors which are the bounds for computing the necessary max-plus closure. Steps 2, 3, and 4 are identical to steps 2, 3, and 4 in the unweighted case. In step 5, we instantiate one copy of the Helmbold and Mayr algorithm for each vertex in $U(B_i)$, where B_i is some terminal branch that we are processing. This takes $O(\log^2 n)$ time and $O(n^3)$ processors per instance. The total processor requirement is $O(n^4)$ for all the instances that we run in parallel.

In step 6, we evaluate the straight-line program containing the weight updates. The program can be evaluated in $O(\log n)$ time with $O(n^2)$ processors using the methods of Miller and Reif [Mi85]. Step 7 is similar to step 8 in the unweighted independent set algorithm. In step 8, we instantiate one copy of the unweighted version of the Helmbold and Mayr algorithm for each terminal branch. No vertex occurs in more than one of the parallel instances, so they can be implemented in $O(\log n)$ time using $O(n^2)$ processors altogether.

Multiplying the time requirements for each step by $O(\log n)$ to account for the recursion depth yields the desired results. □

To conclude this section we explain how to compute a minimum clique cover for a chordal graph.

LEMMA 5.11. *Given a perfect elimination scheme for a chordal graph G and the corresponding lexicographically first maximal independent set, it is possible to compute a minimum clique cover for G in $O(\log n)$ time using $O(m)$ processors.*

Proof. Given a perfect elimination scheme $v_1, v_2, \cdots, v_n$, and any vertex $v \in V(G)$, let $X(v)$ be the neighbors of v that are listed after v in the scheme. Gavril [Ga72] observed that if $\{w_1, w_2, \cdots, w_k\}$ is the lexicographically first maximal independent set corresponding to the given elimination scheme, then the sets $\{w_1\} \cup X(w_1)$, $\{w_2\} \cup X(w_2), \cdots, \{w_k\} \cup X(w_k)$ form a minimum clique cover for G.

These k cliques can be computed in parallel by assigning one processor to every

edge $v-w$. If v comes before w in the PES, then $w \in X(v)$; otherwise $v \in X(w)$. □

This concludes the proof of Theorem 1.1. Combining Lemmas 2.2, 3.8, 4.7, 4.9, 5.6, 5.8, 5.10, and 5.11 yields:

COROLLARY 5.12. *All the objects listed in Theorem* 1.1, *except a maximum weighted independent set, can be computed in* $O(\log^2 n)$ *time using* $O(n^5)$ *processors. All the objects can be found in* $O(\log^3 n)$ *time using* $O(n^4)$ *processors.* □

Acknowledgments. Danny Soroker suggested the current formulation of Lemma 3.2 which led us to find a simplification for the bi-clique subroutine. Professor Richard Anderson, Dr. Ronald Fagin, Professor Richard Karp, Professor Ernst Mayr, Professor Christos Papadimitriou, Dr. David Peleg, and Dr. Larry Stockmeyer, and two referees made helpful comments on various earlier drafts. Professor D. Shier told us about the work in [Ch86]. Mark Novick told us about the tree-construction algorithm in [Be81]. Professors Elias Dahlhaus, Marek Karpinski, and Ernst Mayr helped us understand the algorithms of Dahlhaus and Karpinski sketched in [Da86].

REFERENCES

[Be83] C. BEERI, R. FAGIN, D. MAIER, AND M. YANNAKAKIS, *On the desirability of acyclic database schemes*, J. Assoc. Comput. Mach., 30 (1983), pp. 479–513.

[Be81] P. A. BERNSTEIN AND N. GOODMAN, *Power of natural semijoins*, SIAM J. Comput., 10 (1981), pp. 751–771.

[Be87] A. A. BERTOSSI AND M. A. BONUCCELLI, *Some parallel algorithms on interval graphs*, Discrete Appl. Math., 16 (1987), pp. 101–111.

[Bu74] P. BUNEMAN, *A characterization of rigid circuit graphs*, Discrete Math., 9 (1974), pp. 205–212.

[Ch86] N. CHANDRASEKHARAN AND S. SITHARAMA IYENGAR, NC *algorithms for recognizing chordal graphs and k-trees*, Tech. Report #86-020, Department of Computer Science, Louisiana State University, 1986.

[Co86] R. COLE, *Parallel merge sort*, Proc. 1986 IEEE symposium on Foundations of Computer Science, pp. 511–516.

[Da86] E. DAHLHAUS AND M. KARPINSKI, *The matching problem for strongly chordal graphs is in* NC, Tech. Report 855-CS, Institut für Informatik, Universität Bonn, December 1986.

[Da87] E. DAHLHAUS, M. KARPINSKI, AND E. MAYR, personal communications concerning [Da86], 1987.

[DK87] E. DAHLHAUS AND M. KARPINSKI, *Fast parallel computation of perfect and strongly perfect elimination schemes*, Tech. Report 8519-CS, Institut für Informatik, Universität Bonn, 1987; Proc. Scandinavian Workshop on Algorithms Theory 1988, Springer-Verlag, to appear.

[Ed85] A. EDENBRANDT, *Combinatorial Problems in Matrix Computation*, TR-85-695 (Ph.D. thesis), Department of Computer Science, Cornell University, Ithaca, NY, 1985.

[Ed87] ———, *Chordal graph recognition is in* NC, Inform. Process. Lett., 24 (1987), pp. 239–241.

[Fr75] A. FRANK, *Some polynomial algorithms for certain graphs and hypergraphs*, Proc. 1975 British Combinatorial Conference, Congressus Numerantium XV, Utilitas Mathematica Publishing, pp. 211–226.

[Fu65] D. R. FULKERSON AND O. A. GROSS, *Incidence matrices and interval graphs*, Pacific J. Math., 15 (1965), pp. 835–855.

[Ga72] F. GAVRIL, *Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph*, SIAM J. Comput., 1 (1982), pp. 180–187.

[Ga74] ———, *The intersection graphs of subtrees in trees are exactly the chordal graphs*, J. Combin. Theory B, 16 (1974), pp. 47–56.

[Gi64] P. C. GILMORE AND A. J. HOFFMAN, *A characterization of comparability graphs and of interval graphs*, Canad. J. Math., 16 (1964), pp. 539–548.

[Go80] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[He86] D. HELMBOLD AND E. MAYR, *Perfect graphs and parallel algorithms*, Proc. 1986 IEEE International Conference on Parallel Processing, pp. 853–860, 1986.

[Hi79] D. S. HIRSCHBERG, A. K. CHANDRA, AND D. V. SARWATE, *Computing connected components on parallel computers*, Comm. ACM, 22 (1979), pp. 461-464.

[Jo85] D. S. JOHNSON, *The* NP*-completeness column: an ongoing guide*, J. Algorithms, 6 (1985), pp. 434-451.

[Jo69] C. JORDAN, *Sur les assemblages des lignes*, J. Reine Angew. Math., 70 (1869), pp. 185-190.

[Kl88] P. N. KLEIN, *Efficient parallel algorithms for chordal graphs*, extended abstract presented at the 4th SIAM Conference on Discrete Mathematics, San Francisco, CA, 1988, Proc. 1988 IEEE Symposium on Foundations of Computer Science, to appear.

[Ko85] D. KOZEN, U. V. VAZIRANI, AND V. V. VAZIRANI, NC *Algorithms for comparability graphs, interval graphs and testing for unique perfect matching*, Proc. Fifth Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 206, Springer-Verlag, New York, Berlin, 1985, pp. 496-503.

[Lo72] L. LOVÁSZ, *Normal hypergraphs and the perfect graph conjecture*, Discrete Math., 2 (1972), pp. 253-267.

[Lo85] ———, *Vertex packing algorithms*, Proc. 12th Coll. Aut. Lang. Prog., Lecture Notes in Computer Science 194, Springer-Verlag, New York, Berlin, 1985, pp. 1-15.

[Me83] N. MEGIDDO, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852-865.

[Mi85] G. L. MILLER AND J. H. REIF, *Parallel tree contraction and its application*, Proc. 1985 IEEE Symposium on Foundations of Computer Science, pp. 478-489.

[Na87] J. NAOR, M. NAOR, AND A. A. SCHÄFFER, *Fast parallel algorithms for chordal graphs (extended abstract)*, Proc. 1987 ACM Symposium on Theory of Computing, pp. 355-364; IBM Res. Report RJ629.

[Ro70] D. J. ROSE, *Triangulated graphs and the elimination process*, J. Math. Anal. Appl., 32 (1970), pp. 597-609.

[Ro72] D. J. ROSE, *A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations*, in Graph Theory and Computing, R. C. Read, ed., Academic Press, New York, 1972, pp. 183-217.

[Ro76] D. J. ROSE, R. E. TARJAN, AND G. S. LUEKER, *Algorithmic aspects of vertex elimination on graphs*, SIAM J. Comput., 5 (1976), pp. 266-283.

[Ru80] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. Syst. Sci., 21 (1980), pp. 218-235.

[Ta84] R. E. TARJAN AND M. YANNAKAKIS, *Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs*, SIAM J. Comput., 13 (1984), pp. 566-579.

[Ya87] M. YANNAKAKIS AND F. GAVRIL, *The maximum k-colorable subgraph problem for chordal graphs*, Inform. Process. Lett., 24 (1987), pp. 133-137.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 350-370, April 1989

012

ON THE WORST-CASE ARITHMETIC COMPLEXITY OF APPROXIMATING ZEROS OF SYSTEMS OF POLYNOMIALS*

JAMES RENEGAR†

Abstract. Let $d_1, \cdots, d_n$ be positive integers. Let $\mathcal{P}$ denote the set of systems of polynomials $f: \mathbb{C}^n \to \mathbb{C}^n$ that have only finitely many zeros, including those "at infinity," and that satisfy degree $(f_i) = d_i$ for all i. Let $0 < \varepsilon \leqq R$. It is shown for fixed $d_1, \cdots, d_n$, that with respect to a certain model of computation, the worst-case computational complexity of obtaining ε-approximations to at least those zeros ξ satisfying $|\xi| \leqq R$ for arbitrary $f \in \mathcal{P}$, is $\Theta(\log\log(R/\varepsilon))$; that is to say, both upper and lower bounds are proved. An algorithm for proving the upper bound is introduced. The number of operations required by this algorithm is

$$O\left[n\mathcal{D}^4(\log \mathcal{D})(\log\log(R/\varepsilon)) + n^2\mathcal{D}^4\binom{1+\sum d_i}{n}^4\right], \quad \text{where } \mathcal{D} = \prod_{i=1}^{n} d_i.$$

Key words. polynomials, computational complexity, resultants

AMS(MOS) subject classifications. 65, 68, 90

1. Introduction. Let $P_d(R)$ denote the set of degree-d univariate complex polynomials with all zeros ξ satisfying $|\xi| \leqq R$. It was shown in [10], for fixed $d \geqq 2$, that with respect to a certain model of computation, the worst-case arithmetic complexity of obtaining ε-approximations to either one, or to each, zero of arbitrary $f \in P_d(R)$ is $\Theta(\log\log(R/\varepsilon))$. More specifically, in terms of d as well, a lower bound of $\Omega(\log\log(R/\varepsilon)) - O(\log d)$ operations was proven, and a new algorithm, requiring $O(d^2(\log d)(\log\log(R/\varepsilon)) + d^3 \log d)$ operations, was introduced for the problem of obtaining ε-approximations to all of the zeros. We refer the reader to Renegar [10] for the general model of "computation tree" used to prove the lower bound, but we remark that it encompasses algebraic RAMs (random access machines) whose operations are $+, -, \times, \div$, complex conjugation, and inequality comparison. (See Borodin and Munro [1] as a reference.) Arithmetic operations are assumed to be performed with infinite precision over the complex numbers. The coefficients of the polynomials are *not* assumed to be rationals; therefore, in terms of the "length" of the coefficients, simplifying properties such as lower bounds on the distance between distinct zeros cannot be used. For rational coefficients of fixed length, a uniform $\log\log(R/\varepsilon)$ upper bound is fairly straightforward to prove, but for arbitrary complex coefficients it is not.

The purpose of this paper is to present appropriate generalizations of the above results to the several-variable setting.

Of course, systems of polynomials are not nearly as simple as univariate polynomials. For example, univariate polynomials have finitely many zeros but polynomial systems $f: \mathbb{C}^n \to \mathbb{C}^n$ can have infinitely many zeros, so we cannot hope to approximate all of the zeros unless we restrict attention to "nice" systems. The nice systems to which we restrict our attention in this paper are those systems having only finitely many zeros, including the zeros "at infinity." Formally, if $F: \mathbb{C}^{n+1} \to \mathbb{C}^n$ is the

* Received by the editors August 18, 1987; accepted for publication (in revised form) July 19, 1988. This work was supported by a National Science Foundation Mathematical Sciences Postdoctoral Research Fellowship and was performed while the author was visiting the Department of Operations Research at Stanford University.

† School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853.

homogenization of f (i.e., if degree $(f_i) = d_i$, then the terms of F_i are obtained by multiplying the terms of f_i by the appropriate powers of z_{n+1} so as all to become a degree d_i), then f is said to have finitely many zeros, including those at infinity, if the zero set of F is the union of finitely many complex lines through the origin in $\mathbb{C}^{n+1}$. We refer to these lines as the "zero lines" of F; in the literature they are often referred to as the "solution rays" of F. The zero lines of F that are in $\mathbb{C}^n \times \{0\}$ correspond to the zeros of f at infinity. There is an obvious correspondence between the other zero lines of F and the zeros of f in $\mathbb{C}^n$.

Let $d_1, \cdots, d_n$ be positive integers. Let $\mathscr{P}$ denote the set of systems $f = (f_1, \cdots, f_n): \mathbb{C}^n \to \mathbb{C}^n$ with only finitely many zeros, including those at infinity, and satisfying degree $(f_i) = d_i$ for all i. (Of course $\mathscr{P}$ depends on the specific values of $d_1, \cdots, d_n$.)

We consider the following problem. Let $R \geqq \varepsilon > 0$ and assume $f: \mathbb{C}^n \to \mathbb{C}^n$ is a polynomial system satisfying degree $(f_i) = d_i$ for all i. First determine if $f \in \mathscr{P}$. If so, determine ε-approximations to a subset of the zeros of f containing at least all zeros ξ satisfying $|\xi| \leqq R$. (An ε-approximation of a zero is a point within Euclidean distance ε of the zero.) More specifically, determine points $\mathbb{X}^{(1)}, \cdots, \mathbb{X}^{(m)} \in \mathbb{C}^n$ for which there exist zeros $\xi^{(1)}, \cdots, \xi^{(m)}$ of f with $\|\mathbb{X}^{(i)} - \xi^{(i)}\| \leqq \varepsilon$, where each zero ξ of f satisfying $|\xi| \leqq R$ is listed among the $\xi^{(i)}$ a number of times exactly equal to its multiplicity, and where no zero of f is listed among the $\xi^{(i)}$ a greater number of times than its multiplicity. (Thus, if each zero ξ of f satisfies $|\xi| \leqq R$, then each zero can be considered as being approximated by several points $\mathbb{X}^{(i)}$, the number of such points being equal to its multiplicity.)

We refer to the above approximation problem as "the (ε, R)-approximation problem for f."

We present a test involving

$$O\left[n\mathscr{D}^2 \binom{1+\sum d_i}{n}^4 \right] \quad \text{operations,} \quad \text{where } D = \prod_{i=1}^{n} d_i$$

for determining if $f \in \mathscr{P}$. The validity of this test follows straightforwardly from well-known facts regarding resultants. (Resultants are discussed in § 2.) For those f's that pass this test, that is, for $f \in \mathscr{P}$, we also present an algorithm for solving the (ε, R)-approximation problem for f. The operation count for this algorithm is

$$O\left[n\mathscr{D}^4(\log \mathscr{D})(\log\log(R/\varepsilon)) + n^2\mathscr{D}^4 \binom{I+\sum d_i}{n}^4 \right].$$

Note that coefficient "sizes" do not enter into this bound in any way.

A significant fact about the bound is how it depends on ε and R, that is, the $\log\log(R/\varepsilon)$ term. The lower bound in Renegar [10] showed that in the univariate setting this is the best possible dependence on R and ε that can be obtained. However, that lower bound implies the same lower bound for the several-variable setting. Assume that one of the $d_i \geqq 2$, say, $d_1 \geqq 2$. If $g \in P_{d_1}(R)$ (i.e., a degree-d_1 univariate polynomial with all zeros ξ satisfying $|\xi| \leqq R$), then $f: \mathbb{C}^n \to \mathbb{C}^n$ defined by $f_1(z) = g(z_1)$, $f_2(z) = z_2^{d_2}, \cdots, f_n(z) = z_n^{d_n}$ satisfies $f \in \mathscr{P}$. Any ε-approximation to a zero of f easily gives an ε-approximation to a zero of g. Hence, the (ε, R)-approximation problem for arbitrary $f \in \mathscr{P}$ is at least as hard as the ε-approximation problem for arbitrary $g \in P_{d_1}(R)$, and thus, in the worst case, requires $\Omega(\log\log(R/\varepsilon)) - O(n + \log d_1)$ operations. (The "n" occurs to account for the cost of conversion to the several-variable problem.)

Together, our upper and lower bounds give the main theorem.

MAIN THEOREM. *Fix* $d_1, \cdots, d_n$ *and assume* $\mathscr{D} = \prod_{i=1}^n d_i \geqq 2$. *Let* $0 < \varepsilon \leqq R$. *The arithmetic complexity of obtaining* ε*-approximations to at least those zeros* ξ *of arbitrary* $f \in \mathscr{P}$ *satisfying* $|\xi| \leqq R$ *is* $\Theta(\log\log(R/\varepsilon))$.

Another noteworthy fact about the upper bound is that it is not doubly exponential in n, in contrast to bounds for classical approaches using elimination theory (e.g., see the sections on elimination theory that appear in Van der Waerden [11]—these sections do not appear in newer editions of the book).

The algorithm for obtaining approximations to the zeros of $f \in \mathscr{P}$ is actually an algorithm for obtaining approximations to all of the zero lines of the homogenization $F: \mathbb{C}^{n+1} \to \mathbb{C}^n$ of f, along with a few operations to transform the approximations for F to those for f. By "ε-approximations to all of the zero lines of F" we mean nonzero vectors $\mathbb{X}^{(i)} \in \mathbb{C}^{n+1}$, $i = 1, \cdots, \mathscr{D}$, for which there exists a one-to-one correspondence with nonzero vectors $\xi^{(i)} \in \mathbb{C}^{n+1}$, $i = 1, \cdots, \mathscr{D}$ (say, $\mathbb{X}^{(i)}$ corresponds to $\xi^{(i)}$), where the zero lines of F are precisely the lines $\{\lambda\xi^{(i)};\ \lambda \in \mathbb{C}\}$, $i = 1, \cdots, \mathscr{D}$, each occurring according to its multiplicity, and where

$$\left\| \frac{\mathbb{X}^{(i)}}{\|\mathbb{X}^{(i)}\|} - \frac{\xi^{(i)}}{\|\xi^{(i)}\|} \right\| \leqq \varepsilon.$$

$\| \ \|$ denoting the Euclidean norm on $\mathbb{C}^{n+1}$. (The fact that F has $\mathscr{D}$ zero lines, counting multiplicities, is immediate from Theorem 2.3.)

Assuming $\varepsilon \leqq R$, in the appendix we show that if $\mathbb{X}^{(i)}$, $i = 1, \cdots, \mathscr{D}$, are $\varepsilon/4(R+1)^2$-approximations to all of the zero lines of F, then the set of vectors

$$\left\{ \left(\frac{\mathbb{X}_1^{(i)}}{\mathbb{X}_{n+1}^{(i)}}, \cdots, \frac{\mathbb{X}_n^{(i)}}{\mathbb{X}_{n+1}^{(i)}} \right);\ |\mathbb{X}_{n+1}^{(i)}| / \|\mathbb{X}^{(i)}\| \geqq \frac{3}{4}(R+1) \right\} \tag{1.1}$$

is a solution for the (ε, R)-approximation problem for f.

Let $d_1, \cdots, d_n$ be positive integers. Let $\mathscr{H}$ denote the set of systems of homogeneous polynomials $F: \mathbb{C}^{n+1} \to \mathbb{C}^n$ that have only finitely many zero lines and that satisfy degree $(F_i) = d_i$ for all i.

Henceforth, we focus on the following problem. Given a system of homogeneous polynomials $F: \mathbb{C}^{n+1} \to \mathbb{C}^n$, where degree $(F_i) = d_i$ for all i, is $F \in \mathscr{H}$? If so, determine ε-approximations to all of the zero lines of F.

We present a test involving $O\left[n\mathscr{D}^2\binom{1+\sum d_i}{n}\right]$ operations for determining if $F \in \mathscr{H}$, where $\mathscr{D} = \prod_{i=1}^n d_i$. For $F \in \mathscr{H}$, we present an algorithm for obtaining ε-approximations to all of the zero lines of F. The operation count for this algorithm is

$$O\left[n\mathscr{D}^4(\log \mathscr{D})(\log\log(1/\varepsilon)) + n^2\mathscr{D}^4\binom{1+\sum d_i}{n}^4 \right]. \tag{1.2}$$

From these bounds and (1.1), follow the upper bounds stated earlier to determine if $f \in \mathscr{P}$ and to solve the (ε, R)-approximation problem for arbitrary $f \in \mathscr{P}$.

Using the lower bound of Renegar [10], one can prove that as regards ε, the term $\log\log(1/\varepsilon)$ occurring in (1.2) is the best possible.

Our algorithm is similar in spirit to the algorithm of Lazard [6]; both algorithms work by factoring the "u-resultant." Lazard only sketches a complexity analysis, avoiding degenerate situations and implicitly assuming that the zeros of a single-variable polynomial can be calculated exactly. It is not very difficult to determine "reasonable" complexity bounds for his algorithm if one is only concerned with rational coefficients and is satisfied with a bound on the number of arithmetic operations that grows with the coefficient "lengths." However, his algorithm and analysis are far from providing a uniform bound on arithmetic operations that is independent of the coefficients.

It should be mentioned that Lazard does not restrict attention to the field of complex numbers. Chistov and Grigor'ev [3], [4] extended Lazard's analysis to the problem of approximating a point in each component of the zero set of an arbitrary system of polynomials $f:\mathbb{C}^n \to \mathbb{C}^m$ with rational coefficients. Their bound on the required number of arithmetic operations has the maximal coefficient length as a factor. Similarly, this length appears as a factor in the arithmetic operation bound for the recent algorithm of Canny [2]. Canny's algorithm approximates zeros of systems $f:\mathbb{C}^n \to \mathbb{C}^n$ also via the u-resultant.

Of related interest is Grigor'ev and Vorobjov [5], where the problem of constructing approximate solutions to systems of real polynomial inequalities with rational coefficients is considered. Their arithmetic operation bound has the maximal coefficient length as a factor.

It is certainly not the case that our upper bound can be obtained by rounding coefficients to rationals (where the rounding is a function of R and ε) and then applying the results mentioned above. Because those results have arithmetic operation bounds (not just bit operation bounds) with the maximal coefficient bit length as a fundamental factor, proving our $\log\log(R/\varepsilon)$ result in this manner would require showing that every polynomial system can be perturbed to one with rational coefficients of length bounded by $\log(R/\varepsilon)$, where each zero ξ, $\|\xi\| \leqq R$ of the original system is approximated within distance ε by a zero of the perturbed system. For example, assuming $R=1$ and $\varepsilon=(1/2)^L$, in the univariate case of degree d one would at least need each point in the unit disk in $\mathbb{C}$ to be within distance $(1/2)^L$ of a root of one of the $[O(L)]^{2(d+1)}$ polynomials $\sum_{i=0}^{d} a_i z^i$ with a_i complex rationals of bit length $O(L)$. Of course this is not possible for L large compared to d.

In Renegar [9], a probabilistic analysis of an algorithm for approximating all of the zeros of systems of polynomials is given. The bounds are polynomial in n, $\mathscr{D}$, and L, where L is the number of nonzero coefficients in the considered systems. Hence, by focusing on "sparse" systems, a probabilistic bound independent of $\binom{1+\sum d_i}{n}$ can be obtained. (Note that $\binom{1+\sum d_i}{n}$ grows like $\mathscr{D}^n$ when, for example, all polynomials except one are linear.)

Our algorithm relies on the univariate algorithm of Renegar [10]. The reliance on that particular algorithm is not crucial. What is needed is an algorithm for approximating all zeros of arbitrary $f \in P_d(R)$ with worst-case operation count growing only like $\log\log(R/\varepsilon)$ with respect to ε and R.

2. A few facts about resultants. Let $\mathscr{H}^n_{d_1,\cdots,d_n}$ denote the set of all homogeneous polynomial systems $G:\mathbb{C}^n \to \mathbb{C}^n$ satisfying degree $(G_i) = d_i$. The resultant R for systems in $\mathscr{H}^n_{d_1,\cdots,d_n}$ is a homogeneous polynomial in the coefficients of these systems. It has the property that $R(G)=0$ if and only if G has a nontrivial zero, i.e., $G(x)=0$ for some $x \neq 0$. In this section we state a few facts regarding the resultant that are crucial for our algorithm.

Let $\bar{d} = 1 - n + \sum_i d_i$ and consider $\mathscr{H}^n_{\bar{d}}$, the vector space consisting of all homogeneous polynomials $g:\mathbb{C}^n \to \mathbb{C}$ of degree $\bar{d}$, along with the zero map. A basis for this space is easily seen to be given by the set of terms

$$B = \left\{ z_1^{i_1} z_2^{i_2} \cdots z_n^{i_n}, \qquad \sum_j i_j = \bar{d}, \text{ each } i_j \text{ is a nonnegative integer} \right\}.$$

It is easily shown by the definition of $\bar{d}$ that each of the terms in B satisfies $i_j \geqq d_j$ for at least one j. Partition B into the disjoint union $\cup_{j=1}^{n} B_j$, where B_j contains all terms $z_1^{i_1} \cdots z_n^{i_n}$ satisfying $i_1 < d_1, \cdots, i_{j-1} < d_{j-1}, i_j \geqq d_j$.

To each system $G \in \mathscr{H}^n_{d_1,\cdots,d_n}$ we can associate a linear map from $\mathscr{H}^n_{\bar{d}}$ to itself, defined for the basis terms in B_j by

$$z_1^{i_1} \cdots z_n^{i_n} \mapsto z_1^{i_1} \cdots z_j^{i_j - d_j} \cdots z_n^{i_n} \cdot G_j(z_1, \cdots, z_n).$$

Let $a_j(i_1, \cdots, i_n)$ denote the coefficient in G_j of the term $z_1^{i_1} \cdots z_n^{i_n}$. In terms of the basis B, the matrix corresponding to the above linear map is simply the following: the entry in the intersection of the column and row corresponding to $z_1^{i_1} \cdots z_n^{i_n} \in B_j$ and $z_1^{k_1} \cdots z_n^{k_n}$, respectively, is $a_j(k_1 - i_1, \cdots, k_j - i_j + d_j, \cdots, k_n - i_n)$ if $i_1 \leqq k_1, \cdots, i_j \leqq k_j + d_j, \cdots, i_n \leqq k_n$, and equals zero otherwise. Let $D(G)$ denote the determinant of this matrix. Then $D(G)$ is a homogeneous polynomial in the coefficients of $G \in \mathscr{H}^n_{d_1,\cdots,d_n}$. In fact, it is homogeneous in the coefficients of G_i and has degree, in those coefficients, equal to the number of terms in B_i. Its total degree equals the number of terms in B, that is, $\binom{\bar{d}+n-1}{n-1} = \binom{\sum d_i}{n-1}$.

Assume that the linear map associated with $G \in \mathscr{H}^n_{d_1,\cdots,d_n}$ is nonsingular, so that for each $i = 1, \cdots, n$ some polynomial in $\mathscr{H}^n_{\bar{d}}$ is mapped to the term $z_i^{\bar{d}}$. Hence, for each i,

$$z_i^{\bar{d}} = \sum_{j=1}^{n} p_j(z_1, \cdots, z_n) G_j(z_1, \cdots, z_n)$$

for some polynomials $p_1, \cdots, p_n$ (dependent on i). It easily follows that $G(x) \neq 0$ if $x \neq 0$. Thus $D(G) = 0$ is a necessary condition for these to exist a nontrivial zero for G. However, it is not a sufficient condition, but we do have the following remarkable theorem.

THEOREM 2.1 (Macaulay, 1902, Theorem 6). *Let $M(G)$ denote the determinant of the submatrix (of the matrix corresponding to the linear map induced by G) consisting of entries for which both the row and column correspond to terms in B of the form $z_1^{i_1} \cdots z_n^{i_n}$, with at least two i_j and i_k satisfying $i_j \geqq d_j$, $i_k \geqq d_k$. Then $M(G)$, a polynomial in the coefficients of $G \in \mathscr{H}^n_{d_1,\cdots,d_n}$, is a factor of the polynomial $D(G)$. Moreover, letting $R(G)$ be the polynomial satisfying $D(G) = M(G)R(G)$, then having $R(G) = 0$ is a necessary and sufficient condition for G to have a nontrivial zero.* □

Remark. Macaulay's Theorem 6 actually does not state the last conclusion of Theorem 2.1. This was well known to him, and is stated in the introduction to his paper. A proof of the last conclusion is given in Van der Waerden [11], § 82. (Beware that in some editions of Van der Waerden's book this section on elimination theory has been eliminated!)

The polynomial $R(G)$ is the "resultant." Both it and $M(G)$ are homogeneous in the coefficients of each G_i. The degree of $R(G)$ in the coefficients of G_i is $\prod_{j \neq i} d_j$. Also, $M(G)$ is independent of the coefficients of G_n.

We now turn our attention to systems of homogeneous polynomials $F: \mathbb{C}^{n+1} \to \mathbb{C}^n$ satisfying degree $(F_i) = d_i$. Let $\mathscr{H}^{n+1}_{d_1,\cdots,d_n}$ denote the set of these systems. Here we are concerned with the question, "Does $F \in \mathscr{H}^{n+1}_{d_1,\cdots,d_n}$ have only finitely many zero lines?"

Let $u_1, \cdots, u_{n+1}$ denote variables. For specified values of these variables, consider the system $z \mapsto (F(z), u \cdot z)$, where $u \cdot z = \sum_i u_i z_i$. This is a system in $\mathscr{H}^{n+1}_{d_1,\cdots,d_n,1}$. Let $R(F, u)$ denote the resultant of this system. For D and M as in Theorem 2.1, define $D(F, u)$ and $M(F, u)$ analogously. These are polynomials in the coefficients of $F \in \mathscr{H}^{n+1}_{d_1,\cdots,d_n}$ and the variables u. In the literature, $R(F, u)$ is sometimes referred to as "the u-resultant of F."

We remark, for future reference, that the determinant $M(F, u)$ is independent of the variables u. Also, $D(F, u)$ is the determinant of a $\binom{1+\sum d_i}{n} \times \binom{1+\sum d_i}{n}$ matrix, and $M(F, u)$ is the determinant of a smaller matrix.

PROPOSITION 2.2. *Fix* $F \in \mathscr{H}^{n+1}_{d_1,\cdots,d_n}$. *Then* $R(F, u) = 0$ *for all* u *if and only if* F *has infinitely many zero lines.*

Proof. This is well known. Here is a short proof.

Assume that F has finitely many solution lines. For each of these lines, choose a nonzero vector on that line. Assume $\alpha^{(1)}, \cdots, \alpha^{(m)}$ are the chosen vectors. There exists $x \in \mathbb{C}^{n+1}$ such that $x \cdot \alpha^{(i)} \neq 0$ for all i. Then the system $z \mapsto (F(z), x \cdot z)$ has no nontrivial zero and hence $R(F, x) \neq 0$.

Now assume that F has infinitely many solution lines. Choose nonzero vectors $\alpha^{(1)}, \alpha^{(2)}, \cdots$, on each line in an infinite, but countable, subset of the zero lines. Fix $x \in \mathbb{C}^{n+1}$ satisfying $x \notin \bigcup_i \{y;\ \alpha^{(i)} \cdot y = 0\}$. There exists a complex line L containing x that has infinitely many intersection points with $\bigcup_i \{y;\ \alpha^{(i)} \cdot y = 0\}$. However, for each of these intersection points y, the map $z \mapsto (F(z), y \cdot z)$ has a nontrivial solution so that $R(F, y) = 0$. Hence, the univariate polynomial obtained by restricting $R(F, u)$ to $u \in L$ has infinitely many zeros, and thus must be the zero polynomial. Consequently, $R(F, x) = 0$. Finally, if $x \in \bigcup_i \{y;\ \alpha^{(i)} \cdot y = 0\}$ then it is easy to prove that $R(F, x) = 0$. □

The following theorem is the cornerstone for our algorithm.

THEOREM 2.3. *Assume that* $F \in \mathscr{H}^{n+1}_{d_1,\cdots,d_n}$ *has only finitely many zero lines. For* $u \in \mathbb{C}^{n+1}$, *let* $R(u) = R(F, u)$. *Then* $R(u)$ *has factorization*

$$R(u) = \prod_{l=1}^{\mathscr{D}} (\xi^{(l)} \cdot u),$$

where $\xi^{(l)} \cdot u = \sum_i \xi_i^{(l)} u_i$, $\mathscr{D} = \prod_{i=1}^n d_i$, *and each* $\xi^{(l)}$ *is a nontrivial zero of* F. *Moreover, for each zero line of* F, *the number of the* $\xi^{(l)}$ *that are contained in that zero line equals the multiplicity of that zero line.*

Proof. A proof can be found in § 83 of Van der Waerden [11]. (Again, be careful to choose an edition of Van der Waerden's book containing the section on elimination theory.) □

3. Computing $R(u)$. Using the notation of the previous section, assume that $F \in \mathscr{H}^{n+1}_{d_1,\cdots,d_n}$ has only finitely many solution lines and let $R(u) \doteq R(F, u)$, where $R(F, u)$ is the u-resultant of F. Our algorithm depends on being able to compute $R(u)$ and some of its derivatives along certain complex lines. In this section we discuss procedures for doing this.

Let $\alpha, \beta \in \mathbb{C}^{n+1}$, where $\alpha \neq 0$. We first discuss a procedure for obtaining an expansion of the single-variable polynomial $\lambda \mapsto R(\lambda\alpha + \beta)$.

Referring again to the notation of the previous section, begin by computing the determinant $M(F, u)$—this determinant is independent of the variables u, depending only on the fixed coefficients of F. It can be computed with $O\left[\binom{\bar{d}+n}{n}^3\right]$ operations, where $\bar{d} = 1 - n + \sum_i d_i$.

If $M(F, u) \neq 0$, then by Theorem 2.1, $R(u) = D(F, u)/M(F, u)$. Noting that $D(\lambda\alpha + \beta) \doteq D(F, \lambda\alpha + \beta)$ is defined as the determinant of a certain $\binom{1+\sum d_i}{n} \times \binom{1+\sum d_i}{n}$ matrix, and is of degree $\mathscr{D}$ in the variable λ, compute the coefficients of the polynomial $D(\lambda\alpha + \beta)$ by evaluating this determinant at $\mathscr{D}+1$ distinct values of λ, and then interpolating. Thus, assuming $M(F, u) \neq 0$, we can obtain the coefficients of a nonzero multiple of $R(\lambda\alpha + \beta)$ with $O\left[\mathscr{D}\binom{1+\sum d_i}{n}^3\right]$ operations (using $\mathscr{D} < \binom{1+\sum d_i}{n}$).

Now assume $M(F, u) = 0$. For $t \in \mathbb{R}$, let $F_i^t(z) = tz_i^{d_i} + (1-t)F_i(z)$ for $i = 1, \cdots, n$. Then $M(F^t, u)$ is a polynomial in t alone, and is of degree not exceeding $\binom{1+\sum d_i}{n}$. It is nonconstant since $M(F^0, u) = 0$ and $M(F^1, u) = 1$. Determine the coefficients of $M(F^t, u)$ by evaluating the corresponding determinant for $\binom{1+\sum d_i}{n} + 1$ distinct values of t, and then interpolating.

Determine *the least* integer $0 \leqq k \leqq \binom{1+\sum d_i}{n}$ such that

$$\left.\frac{d^k}{dt^k} M(F^t, u)\right|_{t=0} \neq 0.$$

Then, using Theorem 2.1 and the product rule for differentiation,

$$\left.\frac{d^k D(F^t, u)}{dt^k}\right|_{t=0} = R(F, u) \cdot \left.\frac{d^k M(F^t, u)}{dt^k}\right|_{t=0}$$

as polynomials in u. Hence, $R(\lambda\alpha+\beta)$ equals

$$\left.\frac{d^k D(F^t, \lambda\alpha+\beta)}{dt^k}\right|_{t=0}$$

divided by the already computed nonzero constant

$$\left.\frac{d^k M(F^t, u)}{dt^k}\right|_{t=0}.$$

Finally, we examine the computation of

$$\left.\frac{d^k D(F^t, \lambda\alpha+\beta)}{dt^k}\right|_{t=0}.$$

Since $D(F^t, \lambda\alpha+\beta)$ is a polynomial in the variables λ and t, of degree not exceeding $\mathscr{D}$ in λ and of degree not exceeding $\binom{1+\sum d_i}{n}$ in t, $D(F^t, \lambda\alpha+\beta)$ can be expanded as follows:

$$D(F^t, \lambda\alpha+\beta) = \sum_{i=1}^{\binom{1+\sum d_i}{n}} \left(\sum_{j=0}^{\mathscr{D}} a_{ij}\lambda^j \right) t^i.$$

We wish to determine $\sum_{j=0}^{\mathscr{D}} a_{kj}\lambda^j$ for k, as defined earlier.

Choose $\mathscr{D}+1$ distinct values $\lambda_l \in \mathbb{C}$ and $\binom{1+\sum d_i}{n}+1$ distinct values t_m. For fixed λ_l, evaluate the determinant $D(F^{t_m}, \lambda_l\alpha+\beta)$ for all pairs $\{(\lambda_l, t_m)\}_m$. Interpolate in t to determine the expansion of the single-variable polynomial $D(F^t, \lambda_l\alpha+\beta)$, thereby obtaining the value $\sum_{j=0}^{\mathscr{D}} a_{kj}\lambda_l^j$. Do this for each λ_l. Then interpolate in λ to obtain the expansion $\sum_{j=0}^{\mathscr{D}} a_{kj}\lambda^j$.

Thus, we have a method for computing the coefficients of $R(\lambda\alpha+\beta)$. The total operation count is dominated by the operation required to compute the $[\mathscr{D}+1]\left[\binom{1+\sum d_i}{n}+1\right]$ determinants $D(F^{t_m}, \lambda_l\alpha+\beta)$. Hence, the total operation count is $O\left[\mathscr{D}\binom{1+\sum d_i}{n}^4\right]$.

Besides an expansion for $R(\lambda\alpha+\beta)$, we will also need expansions for the multivariable polynomials $R(\lambda\alpha+\rho_1 e_i+\rho_2 e_j+\beta)$, where ρ_1 and ρ_2 are variables over $\mathbb{C}$ and where e_i and e_j are ith and jth unit vectors. Defining F^t as before, and writing

$$D(F^t, \lambda\alpha+\rho_1 e_i+\rho_2 e_j+\beta) = \sum_{m_1=1}^{\binom{1+\sum d_i}{n}} \left(\sum_{m_2=0}^{\mathscr{D}} \left(\sum_{m_3=0}^{\mathscr{D}} \left(\sum_{m_4=0}^{\mathscr{D}} a_{m_1 m_2 m_3 m_4} \rho_2^{m_4} \right) \rho_1^{m_3} \right) \lambda^{m_2} \right) t^{m_1},$$

we wish to obtain the triple summation which is a multiple of t^k, where k is the smallest integer such that $d^k M(F^t, u)/dt^k|_{t=0} \neq 0$. Dividing that triple summation by the constant $d^k M(F^t, u)/dt^k|_{t=0}$ gives $R(\lambda\alpha+\rho_1 e_i+\rho_2 e_j+\beta)$. However, to obtain the triple summation we can use the generalization of the procedure we used for computing $d^k D(F^t, \lambda\alpha+\beta)/dt^k|_{t=0}$. First, interpolate in t to obtain

$$\sum_{m_2=0}^{\mathscr{D}} \left(\sum_{m_3=0}^{\mathscr{D}} \left(\sum_{m_4=0}^{\mathscr{D}} a_{m_1 m_2 m_3 k} \rho_2^{m_4} \right) \rho_1^{m_3} \right) \lambda^{m_2}$$

for *fixed* values of the parameters ρ_1, ρ_2, and λ. Then interpolate in λ to obtain

$$\sum_{m_3=0}^{\mathscr{D}} \left(\sum_{m_4=0}^{\mathscr{D}} a_{m_1 m_2 m_3 k} \rho_2^{m_4} \right) \rho_1^{m_3}, \qquad m_2 = 1, \cdots, \mathscr{D}$$

for fixed values of the parameters ρ_1 and ρ_2. For each m_2, interpolate in ρ_1 to obtain

$$\sum_{m_4}^{\mathscr{D}} a_{m_1 m_2 m_3 k} \rho_2^{m_4} \qquad m_2, m_3 = 1, \cdots, \mathscr{D}$$

for fixed values of the parameter ρ_2. Finally, for each pair (m_2, m_3), interpolate in ρ_1 to obtain all of the coefficients $a_{m_1 m_2 m_3 k}$. Altogether, $O\left[\mathscr{D}^3 \binom{1+\sum d_i}{n}^4\right]$ operations suffice.

4. The algorithm. In this section we present the algorithm which, given $F \in \mathscr{H}^{n+1}_{d_1, \cdots, d_n}$, determines if F has only finitely many solution lines and, if so, obtains ε-approximations to all of them.

The idea underlying the algorithm is rather simple, although the technicalities that must be dealt with are not. Here is the idea. Assume that F has only finitely many solution lines and, for simplicity, assume that each of these are of multiplicity 1. By Theorem 2.1,

$$R(u) = \prod_{l=1}^{\mathscr{D}} (\xi^{(l)} \cdot u), \tag{4.1}$$

where the $\xi^{(l)}$ are vectors on the solution lines. Let

$$H^{(l)} = \{x \in \mathbb{C}^{n+1};\ \xi^{(l)} \cdot x = 0\}. \tag{4.2}$$

Assume that $\alpha', \beta' \in \mathbb{C}^{n+1}$, $\alpha' \neq 0$, and assume that the complex line $\{\lambda\alpha' + \beta';\ \lambda \in \mathbb{C}\}$ intersects each of the hyperplanes $H^{(l)}$, but does not intersect $H^{(l)} \cap H^{(m)}$ if $l \neq m$. Compute the zeros λ' of the degree $\mathscr{D}$ single-variable polynomial $R(\lambda\alpha' + \beta')$—for the moment we assume that these can be calculated exactly. There is a one-to-one correspondence between the λ' and the l defined by the relation $\lambda'\alpha' + \beta' \in H^{(l)}$. For λ' corresponding to l, the vector

$$\left(\frac{\partial}{\partial u_1} R(u) \Big|_{u=\lambda'\alpha+\beta}, \cdots, \frac{\partial}{\partial u_{n+1}} R(u) \Big|_{u=\lambda'\alpha+\beta} \right)$$

is a nonzero scalar multiple of $\xi^{(l)}$ and hence is on the zero line $\{\lambda\xi^{(l)};\ \lambda \in \mathbb{C}\}$. This is the main idea behind the algorithm.

Of course the algorithm must be able to work without relying on the above simplifying assumptions.

We present the steps of the algorithm and state propositions regarding the steps simultaneously in hopes that this will better motivate what the steps are designed to accomplish. Proofs are relegated to § 5. Some of the propositions rely on $O(\ )$ notation for upper bounds and $\Omega(\ )$ notation for lower bounds. The constants are independent of $n, d_1, \cdots, d_n$ and hence independent of F. Specific constants can be obtained with more lengthy proofs.

For nonzero $X, Y \in \mathbb{C}^{n+1}$, define

$$\operatorname{dis}(X, Y) = \min \left\{ \left\| \frac{w_1 X}{\|w_1 X\|} - \frac{w_2 Y}{\|w_2 Y\|} \right\|;\ w_1, w_2 \in \mathbb{C} \setminus \{0\} \right\}.$$

Using the homogeneity of F, it is easily shown that $\mathbb{X}^{(i)}$, $i = 1, \cdots, \mathscr{D}$ are ε-approximations to the zero lines of F if and only if $\operatorname{dis}(\mathbb{X}^{(i)}, \xi^{(i)}) \leqq \varepsilon$ for all i, where the $\xi^{(i)}$ are as in (4.1).

For $x \in \mathbb{C}$, define

$$\mu(x) = (1, x, x^2, \cdots, x^n).$$

We use λ to denote a complex variable.

Step 1. Compute $R(\alpha)$ for all $\alpha \in \{\mu(j); j = 0, 1, \cdots, n\mathscr{D}\}$.

From the results in § 3, Step 1 can be accomplished with $O\left[n\mathscr{D}^2\binom{1+\sum d_i}{n}^4\right]$ operations.

PROPOSITION 4.1. *All numbers computed in Step* 1 *are zero if and only if* F *has infinitely many zero lines.*

Proof. Since for any distinct integers $j_1, \cdots, j_{n+1}$ the $(n+1) \times (n+1)$ matrix with ith row $\mu(j_i)$ is invertible, there are at most $n\mathscr{D}$ integer values j such that $\mu(j) \in \bigcup_l \{u; \zeta^{(l)} \cdot u = 0\}$. □

Hereafter, we assume that F has only finitely many zero lines.

The purpose of the next two steps is to determine a vector α' for which the "angle of incidence" of α' with any of the complex hyperplanes $H^{(l)}$ can be bounded away from zero. This property of α' will be relied on in the analysis in two ways. First, it will provide a bound on the absolute value of the zeros of any univariate polynomial $\lambda \mapsto R(\lambda\alpha' + \beta)$. We will need this bound when we call on the univariate algorithm of Renegar [10]. Second, the property of α' will guarantee that for any β, if dis $(\xi^{(l)}, \xi^{(m)})$ is small, then so is $|\lambda^{(l)} - \lambda^{(m)}|$ where $\lambda^{(l)}\alpha' + \beta \in H^{(l)}$, $\lambda^{(m)}\alpha' + \beta \in H^{(m)}$. This will be important for proving the correctness of the procedure for determining the number of lines in a "clustered" set of zero lines.

More specifically, for α' as determined by Steps 2 and 3, we have the following.

PROPOSITION 4.2. *For any* $\beta \in \mathbb{C}^{n+1}$, *all zeros* λ' *of* $R(\lambda\alpha' + \beta)$ *satisfy*

$$|\lambda'| = O(\|\beta\|[n\mathscr{D}]^{2n\mathscr{D}}).$$

Moreover, if $\lambda^{(l)}\alpha' + \beta \in H^{(l)}$, $\lambda^{(m)}\alpha' + \beta \in H^{(m)}$, *then*

$$|\lambda^{(l)} - \lambda^{(m)}| = O(\|\beta\|[n\mathscr{D}]^{5n\mathscr{D}} \text{ dis } (\xi^{(l)}, \xi^{(m)})).$$

Proof. Proposition 5.4. □

If we are only concerned with polynomial systems with rational coefficients, an analogue of Proposition 4.2 with bounds depending on the bit lengths of the coefficients is easily proven. Bounds, such as ours, which hold for all polynomial systems require more detailed arguments.

Step 2. For each $j = 0, 1, \cdots, n\mathscr{D}$ and each $i = 1, \cdots, n+1$, compute the coefficients $a_k(i, j)$ of $R(\lambda\alpha + e_i) = \sum_{k=1}^{\mathscr{D}} a_k(i, j)\lambda^k$, where $\alpha = \mu(j)$.

From the results in § 3, Step 2 can be accomplished with $O\left[n^2\mathscr{D}^2\binom{1+\sum d_i}{n}^4\right]$ operations.

Note that $a_{\mathscr{D}}(i, j) = R(\mu(j))$.

Step 3. Let $J \subseteq \{0, 1, \cdots, n\mathscr{D}\}$ denote the subset $J = \{j; R(\mu(j)) \neq 0\}$. By Proposition 4.1, $J \neq \varnothing$. Determine $j' \in J$ satisfying

$$\max_{\substack{i \\ k < \mathscr{D}}} \left| \frac{a_k(i, j')}{R(\mu(j'))} \right|^2 = \min_{j \in J} \max_{\substack{i \\ k < \mathscr{D}}} \left| \frac{a_k(i, j)}{R(\mu(j))} \right|^2.$$

Let $\alpha' = \mu(j)$.

The reduction to the univariate case occurs in the next step. However, rather than a reduction to a single univariate polynomial, we are forced to consider $n\mathscr{D}(\mathscr{D} - 1)/2 + 1$ univariate polynomials, approximating the zeros for each of these for the following reason. Recall the "idea" behind the algorithm as discussed at the beginning of this section. Assume β' is such that the complex line $\{\lambda\alpha' + \beta'; \lambda \in \mathbb{C}\}$ intersects $H^{(l)}$ and

$H^{(m)}$ at nearly the same point, yet dis $(\xi^{(l)}, \xi^{(m)})$ is large. Then even if $\gamma^{(l)}$ is a close approximation to the point $\lambda^{(l)}$, for which $\lambda^{(l)}\alpha'+\beta' \in H^{(l)}$, it is not likely that dis $(X^{(l)}, \xi^{(l)})$ is small where

$$X^{(l)} = \left(\frac{\partial}{\partial u_1} R(u) \Big|_{u=\gamma^{(l)}\alpha'+\beta'}, \cdots, \frac{\partial}{\partial u_{n+1}} R(u) \Big|_{u=\gamma^{(l)}\alpha'+\beta'} \right).$$

To guarantee good approximations, we need $\{\lambda\alpha'+\beta'; \lambda \in \mathscr{C}\}$ to intersect $H^{(l)}$ and $H^{(m)}$ at nearly the same point only if dis $(\xi^{(l)}, \xi^{(m)})$ is small. (Furthermore, in that case, we need to define $X^{(l)}$ by appropriate higher-order derivatives.) As will be proven, at least one of the β's considered in Step 4 has this property; the large amount of computation required in Steps 4, 5, and 6 is to determine which one. (What these steps are designed to accomplish can be achieved easily if we restrict ourselves to rational coefficients and are only concerned with bounding the number of required arithmetic operations in the algorithm by a polynomial in the bit length of the coefficients.)

Step 4 involves a new parameter, $\varepsilon' > 0$.

Step 4. For each $k = 0, 1, \cdots, n\mathscr{D}(\mathscr{D}-1)/2$ apply the algorithm in Renegar [10] (or any other algorithm with an $O(\log\log(R/\varepsilon))$ bound) to obtain ε'-approximations $\gamma_1(k), \cdots, \gamma_{\mathscr{D}}(k)$ for all of the zeros $\lambda_1(k), \cdots, \lambda_{\mathscr{D}}(k)$ (counting multiplicities) of $R(\lambda\alpha'+\beta)$, where $\beta = \mu(k)$.

We will later show that any value $\varepsilon' = O(\varepsilon^{n\mathscr{D}^4+1}/[n\mathscr{D}]^{10n^2\mathscr{D}^5+3n+3})$ suffices for our purposes.

The algorithm in Renegar [10] requires an a priori bound on the $|\lambda_i(k)|$. However, since $\|\mu(k)\| < [n\mathscr{D}]^{2n}$, such a bound can be obtained from Proposition 4.2. Using this and the $O(d^2(\log d)(\log\log(R/\varepsilon)) + d^3 \log d)$ bound for the algorithm in Renegar [10], we find that Step 4 can be accomplished with $O(n\mathscr{D}^4(\log \mathscr{D})(\log\log(1/\varepsilon')) + n\mathscr{D}^5 \log \mathscr{D})$ operations.

In the next step we partition the approximations $\gamma_1(k), \cdots, \gamma_{\mathscr{D}}(k)$ into clusters for each k. Roughly, a cluster of the approximations is a subset of the approximations that is contained in a disk and for which none of the other approximations is contained in a much larger concentric disk. The radius ε'' of the smaller disk and the magnitude δ of the quotient of the radius of the larger disk to that of the smaller disk will be crucial in our analysis.

In Step 6 we will single out a k for which Step 5 has produced the largest number of clusters. As we will prove, this k has the property that $\{\lambda\alpha'+\mu(k); \lambda \in \mathbb{C}\}$ intersects $H^{(l)}$ and $H^{(m)}$ at nearly the same point only if dis $(\xi^{(l)}, \xi^{(m)})$ is small.

Step 5 requires the cluster parameter δ mentioned above. As will be proven, all $\delta = \Omega([n\mathscr{D}]^{10n\mathscr{D}}/\varepsilon)$ will suffice for our purposes.

Step 5. Initially, let $\varepsilon'' = \varepsilon'$, where ε' is as in Step 4.

Step 5.1. Determine if there exists i, j, k such that

$$(\varepsilon'')^2 < |\gamma_i(k) - \gamma_j(k)|^2 \leqq (\delta\varepsilon'')^2.$$

If so, let $\delta\varepsilon'' \to \varepsilon''$ and repeat Step 5.1.

Step 5.2. For each k, partition the approximations into disjoint subsets $P^{[h]}(k)$, $h = 1, \cdots, h(k)$, where $\gamma_i(k)$ and $\gamma_j(k)$ are in the same subset if and only if $|\gamma_i(k) - \gamma_j(k)| \leqq \varepsilon''$. (To establish the existence of this partition, we need the property that $|\gamma_i(k) - \gamma_j(k)| \leqq \varepsilon''$ and $|\gamma_j(k) - \gamma_m(k)| \leqq \varepsilon''$ together imply $|\gamma_i(k) - \gamma_m(k)| \leqq \varepsilon''$.) But this is trivial, assuming $\delta \geqq 2$, since Step 5.1 has been passed through.

It is easily seen that Step 5.1 will be passed through after at most $O(n\mathscr{D}^4)$ iterations, and hence the final value of ε'' satisfies $\varepsilon'' \leqq \delta^{n\mathscr{D}^4}\varepsilon'$. Since each iteration of 5.1 involves $O(n\mathscr{D}^4)$ operations, as does Step 5.2, the operation count for Step 5 is $O(n^2\mathscr{D}^8)$.

Step 6. Determine k' satisfying $h(k') = \max_k h(k)$, i.e., a k with the largest number of clusters. Let $\beta' = \mu(k')$. For each h fix a $\gamma^{[h]} \in P^{[h]}(k')$. Let $x^{[h]} = \gamma^{[h]}\alpha' + \beta'$. (If $h(k') = 1$, we define $x[h] = 0$ to simplify the analysis later.)

To ease the exposition, we now alter our notation slightly. There is a one-to-one correspondence between the points $\lambda_i(k')$, k' as in Step 6, and the complex hyperplanes $H^{(l)}$, where $\lambda_i(k')$ corresponds to $H^{(l)}$ only if $\lambda_i(k')\alpha' + \beta' \in H^{(l)}$. Re-indexing the hyperplanes if necessary, we may write $\lambda^{(l)}$ for the $\lambda_i(k')$ corresponding to $H^{(l)}$, and $\gamma^{(l)}$ for the approximation $\gamma_i(k')$ of that $\lambda_i(k')$. Also, we replace $P^{[h]}(k')$ by $P^{[h]}$.

The goals of all preceding steps are summarized in the following proposition.

PROPOSITION 4.3. *For any* $\delta = \Omega([n\mathscr{D}]^{10n\mathscr{D}})$ *and for* ε'' *as given at the end of Step* 5.1,

$$\gamma^{(l)} \in P^{[h]} \Rightarrow |\xi^{(l)} \cdot x^{[h]}| = O(\varepsilon''[n\mathscr{D}]^{n+1}\|\xi^{(l)}\|). \tag{4.3}$$

$$\gamma^{(l)} \notin P^{[h]} \Rightarrow |\xi^{(l)} \cdot x^{[h]}| = \Omega(\delta\varepsilon''\|\xi^{(l)}\|/[n\mathscr{D}]^{2n\mathscr{D}}), \tag{4.4}$$

$$\gamma^{(l)}, \gamma^{(m)} \in P^{[h]} \Rightarrow \operatorname{dis}(\xi^{(l)}, \xi^{(m)}) = O(\varepsilon''[n\mathscr{D}]^{3n+2}). \tag{4.5}$$

Proof. Proposition 5.6. □

Combining Proposition 4.3 with the following two propositions will motivate the final step of the algorithm. The first of these two propositions covers a "trivial" case.

PROPOSITION 4.4. *Assume that for all* $l, m \in \{1, \cdots, \mathscr{D}\}$ *we have that* $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq \varepsilon'''$. *For all* $\varepsilon''' = O(1/\sqrt{n})$, *the following is then true. Let* i' *be an index satisfying*

$$\left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}}}\right|_{u=0}\Bigg| = \max_i \left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_i^{\mathscr{D}}}\right|_{u=0}\Bigg|$$

and let $\mathbb{X} \in \mathbb{C}^{n+1}$ *be the vector*

$$\mathbb{X}_i = \left.\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}-1}\partial u_i}\right|_{u=0} \qquad i = 1, \cdots, n+1.$$

Then $\operatorname{dis}(\mathbb{X}, \xi^{(l)}) = O(n\varepsilon''')$ *for all* l.

Proof. Proposition 5.8. □

PROPOSITION 4.5. *Let* $S \subset \{1, \cdots, \mathscr{D}\}$ *contain* N *elements, where* $0 < N < \mathscr{D}$. *Assume that for all* $l, m \in S$, *we have* $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq \varepsilon'''$. *Let* $x \in \mathbb{C}^{n+1}$, $x \neq 0$. *Assume that if* $l \notin S$, *then* $|\xi^{(l)} \cdot x| \leqq \rho_1\|\xi^{(l)}\|\,\|x\|$, *and assume that if* $l \notin S$, *then* $|\xi^{(l)} \cdot x| \geqq \rho_2\|\xi^{(l)}\|\,\|x\|$, *where* $\rho_2 > 0$. *Then for all* $\varepsilon''' = O(1/\sqrt{n})$ *and for all* $\rho_1/\rho_2 = O(\varepsilon'''/\mathscr{D}!n)$, *the following is true. Let* i' *be an index satisfying*

$$\left|\frac{\partial^N R(u)}{\partial u_{i'}^N}\right|_{u=x}\Bigg| = \max_i \left|\frac{\partial^N R(u)}{\partial u_i^N}\right|_{u=x}\Bigg|,$$

and let $\mathbb{X} \in \mathbb{C}^{n+1}$ *be the vector*

$$\mathbb{X}_i = \left.\frac{\partial^N R(u)}{\partial u_{i'}^{N-1}\partial u_i}\right|_{u=x} \qquad i = 1, \cdots, n+1.$$

Then $\operatorname{dis}(\mathbb{X}, \xi^{(l)}) = O(n\mathscr{D}\varepsilon''')$ *for all* $l \in S$.

Proof. Proposition 5.9. □

Now we combine the last three propositions to motivate and prove the correctness of the final step of the algorithm.

Assume that $0 < C \leqq 1$ is sufficiently small so that for all n, $\mathscr{D}$ and $0 < \varepsilon \leqq 1$,

$$\varepsilon''' \doteq \frac{C\varepsilon}{n\mathscr{D}} \tag{4.6}$$

satisfies the conditions required for Propositions 4.4 and 4.5.

For $h = 1, \cdots, h(k')$, let

$$S^{[h]} = \{l;\ \gamma^{(l)} \in P^{[h]}\},$$

i.e., the "indices" of the approximations in $P^{[h]}$. We now show that for all

$$\delta = \Omega\left(\frac{[n\mathscr{D}]^{10n\mathscr{D}}}{\varepsilon}\right), \tag{4.7}$$

and for all

$$\varepsilon' = O(\varepsilon/\delta^{n\mathscr{D}^4}[n\mathscr{D}]^{3n+3}) \tag{4.8}$$

the conditions required for Proposition 4.4 (if $h(k') = 1$) or the conditions required for Proposition 4.5 (if $h(k') > 1$) are satisfied by $S = S^{[h]}$, $x = x^{[h]}$ for ε''' as defined by (4.6).

First note that combining the bound $\varepsilon'' \leqq \delta^{n\mathscr{D}^4}\varepsilon'$ (as discussed after Step 5) with (4.5) gives

$$l, m \in S^{[h]} \Rightarrow \operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq O(\delta^{n\mathscr{D}^4}[n\mathscr{D}]^{3n+2}\varepsilon'). \tag{4.9}$$

Assume $h(k') = 1$. Then assuming δ is of the form (4.7) (to meet the requirement of Proposition 4.3) by choosing ε' of the form (4.8), we have from (4.9) that $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq \varepsilon'''$ for all l, m, where ε''' is as in (4.6). Hence the conditions required for Proposition 4.4 are then satisfied.

Now assume $h(k') > 1$ and fix $h \in \{1, \cdots, h(k')\}$. Define

$$\begin{aligned} \rho_1 &= \max\{|\xi^{(l)} \cdot x^{[h]}|/\|\xi^{(l)}\|\ \|x^{[h]}\|;\ l \in S^{[h]}\}, \\ \rho_2 &= \min\{|\xi^{(l)} \cdot x^{[h]}|/\|\xi^{(l)}\|\ \|x^{[h]}\|;\ l \notin S^{[h]}\}. \end{aligned} \tag{4.10}$$

By (4.3) and (4.4),

$$\frac{\rho_1}{\rho_2} = O([n\mathscr{D}]^{2n\mathscr{D}+n+1}/\delta),$$

and hence, assuming (4.7), we find that ρ_1/ρ_2 is sufficiently small as required by Proposition 4.5 for ε''' as in (4.6). Also, assuming δ fixed and of the form (4.7), by choosing ε' of the form (4.8) we have from (4.9) that

$$l, m \in S^{[h]} \Rightarrow \operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq \varepsilon''',$$

for ε''' as in (4.6). Finally, we note that $x^{[h]} \neq 0$ since otherwise we would contradict (4.4) for $l \notin S^{[h]}$. We have thus established the conditions required for Proposition 4.5 in the case that $h(k') > 1$ and for Proposition 4.4 in the case that $h(k') = 1$.

Step 7. If $h(k') = 1$, then determine the index i' satisfying

$$\left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}}}\bigg|_{u=0}\right|^2 = \max_i \left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_i^{\mathscr{D}}}\bigg|_{u=0}\right|^2,$$

and let $\mathbb{X}^{[1]} \in \mathbb{C}^{n+1}$ be the vector

$$\mathbb{X}_i^{[1]} = \frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}-1}\partial u_i}\bigg|_{u=0} \qquad i = 1, \cdots, n+1.$$

If $h(k') > 1$, then perform the following for each $h \in \{1, \cdots, h(k')\}$. Let $N = N_h$, where N_h is the number of indices in $S^{[h]}$. Determine the index i' satisfying

$$\left|\frac{\partial^N R(u)}{\partial u_{i'}^N}\bigg|_{u=x^{[h]}}\right|^2 = \max_i \left|\frac{\partial^N R(u)}{\partial u_i^N}\bigg|_{u=x^{[h]}}\right|^2,$$

and let $\mathbb{X}^{[h]}$ be the vector

$$\mathbb{X}_i^{[h]} = \frac{\partial^N R(u)}{\partial u_{i'}^{N-1} \partial u_i}\bigg|_{u=x^{[h]}} \qquad i = 1, \cdots, n+1.$$

Letting $\mathbb{X}^{(i)}$, $i = 1, \cdots, \mathscr{D}$ be the vectors $\mathbb{X}^{[h]}$, $h = 1, \cdots, h(k')$, where $\mathbb{X}^{[h]}$ occurs N_h times, we find from the conclusions of Proposition 4.4 and 4.5 that if C in (4.6) is sufficiently small, then $\mathbb{X}^{(i)}$, $i = 1, \cdots, \mathscr{D}$ give ε-approximations to the zero lines of F, accounting for their multiplicities.

We assume the computations in Step 7 are carried out as follows. (If $h(k') = 1$, redefine $\alpha' = \beta' = x^{[1]} = 0$.) First compute the coefficients for the three variable polynomials $R(\lambda\alpha' + \rho_1 e_i + \rho_2 e_j + \beta')$, $i, j = 1, \cdots, n+1$, using the method of § 3. This requires $O\left[n^2\mathscr{D}^3\binom{1+\sum d_i}{n}^4\right]$ operations. From these compute the required derivatives. In all, Step 7 requires $O\left[n^2\mathscr{D}^3\binom{1+\sum d_i}{n}^4\right]$ operations.

Relying on (4.7), (4.8) and the operation counts already given for each of the steps, the total operation count for the algorithm is

$$O\left[n\mathscr{D}^4(\log \mathscr{D})(\log\log(1/\varepsilon)) + n^2\mathscr{D}^8 + n^2\mathscr{D}^3\binom{1+\sum d_i}{n}^4\right].$$

5. Proofs. In this section we prove the propositions that we relied on in the previous section. In the course of doing this we will need to prove several lemmas.

In our calculations we sometimes implicitly use the assumption $\mathscr{D} \geqq 2$. For example, under this assumption we may write $n\mathscr{D} + n + 1 \leqq 2n\mathscr{D}$. The following lemma will also occasionally be used implicitly in the analysis.

We retain the notation $\xi^{(l)}$ and $H^{(l)}$ as in (4.1) and (4.2).

LEMMA 5.1. *Assume $R(u) \not\equiv 0$ (i.e., is not identically the zero polynomial). Let $\alpha \in \mathbb{C}^{n+1}$. Then $\alpha \in \bigcup_l H^{(l)}$ implies $R(\lambda\alpha + \beta)$ is of degree less than $\mathscr{D}$ for all $\beta \in \mathbb{C}^{n+1}$, and $\alpha \notin \bigcup_l H^{(l)}$ implies $R(\lambda\alpha + \beta)$ is of degree exactly $\mathscr{D}$ for all $\beta \in \mathbb{C}^{n+1}$.*

Proof. Follows immediately from the identity

$$R(\lambda\alpha + \beta) = \prod_l \xi^{(l)} \cdot (\lambda\alpha + \beta). \qquad \square$$

Recall that $\mu(x) = (1, x, x^2, \cdots, x^n)$.

LEMMA 5.2. *Let $\mathscr{M}$ be any set of complex hyperplanes M in $\mathbb{C}^{n+1}$. Let N denote the number of hyperplanes in $\mathscr{M}$. For at least one $j'' \in \{0, 1, \cdots, nN\}$, $\mu(j'')$ satisfies*

$$\|\mu(j'') - x\| \geqq 1/(1+nN)^{n+1} \quad \text{for all } x \in M \in \mathscr{M}.$$

Proof. For each $j \in \{0, 1, \cdots, nN\}$, let $v(j)$ be a vector of smallest length such that $\mu(j) + v(j)$ lies in a hyperplane in $\mathscr{M}$. Note that for some distinct $j_1, \cdots, j_{n+1}$, each of the $\mu(j_i) + v(j_i)$ must lie in the same hyperplane. Thus, letting A, respectively, B, be the matrix with ith row $\mu(j_i)$, respectively, $v(j_i)$, we have that $A + B$ is singular. Hence,

$$\min_{\|w\|=1} \|Aw\| \leqq \max_{\|w\|=1} \|Bw\| \leqq \sqrt{n+1} \max_{j_i} \|v(j_i)\|.$$

Letting $j'' \in \{j_1, \cdots, j_{n+1}\}$ denote an index satisfying $\|v(j'')\| = \max_{j_i} \|v(j_i)\|$, we thus have

$$\|\mu(j'') - x\| \geqq \frac{1}{\sqrt{n+1}} \min_{\|w\|=1} \|Aw\|, \quad \text{for all } x \in M \in \mathscr{M}. \tag{5.1}$$

Note that for any $w \neq 0$, Aw has coordinates equal to the values taken on by the nonzero polynomial of degree at most n, $z \mapsto \sum_{i=0}^{n} w_{i+1} z^i$, at the $n+1$ distinct integers j_i. Since this polynomial can have at most n zeros, $Aw \neq 0$ and hence A is invertible.

Finally we note that for each i, $A^{-1}e_i$ gives the coefficients of the degree-n polynomial p satisfying $p(j_i)=1$, $p(j_k)=0$ if $k\neq i$, that is, the coefficients of $p(z)=\prod_{k\neq i}(z-j_k)/\prod_{k\neq i}(j_i-j_k)$. Writing $p(z)=\sum a_i z^i$ and noting $|\prod_{k\neq i} j_i-j_k|\geqq 1$, we thus have

$$\|A^{-1}e_i\|\leqq\sum|a_i|\leqq\sum_{j=0}^{n}\binom{n}{j}(nN)^j=(1+nN)^n.$$

Hence

$$\min_{\|w\|=1}\|Aw\|=\frac{1}{\max_{\|w\|=1}\|A^{-1}w\|}\geqq\frac{1}{\sqrt{n+1}\max_i\|A^{-1}e_i\|}\geqq\frac{1}{\sqrt{n+1}(1+nN)^n}.$$

Together with (5.1) this gives the lemma. □

LEMMA 5.3. *Assume $R(u)\not\equiv 0$. For at least one $j''\in\{0,1,\cdots,n\mathscr{D}\}$, $\alpha''=\mu(j'')$ has the property that for all i, $R(\lambda\alpha''+e_1)$ is a degree-$\mathscr{D}$ polynomial with all zeros λ'' satisfying $|\lambda''|\leqq(1+n\mathscr{D})^{n+1}$.*

Proof. Let $\mathscr{M}=\{H^{(l)};\ l=1,\cdots,\mathscr{D}\}$ and let $j''\in\{0,1,\cdots,n\mathscr{D}\}$ denote an integer such that $\alpha''=\mu(j'')$ satisfies the conclusion of Lemma 5.2. Then $\alpha''\notin\bigcup_l H^{(l)}$ and hence, by Lemma 5.1, $R(\lambda\alpha''+e_i)$ is of degree $\mathscr{D}$ for all i. Moreover, for all i and l,

$$\frac{|\xi^{(l)}\cdot\alpha''|}{\|\xi^{(l)}\|}=\min\{\|x-\alpha''\|;\ x\in H^{(l)}\}\geqq 1/(1+n\mathscr{D})^{n+1}.$$

Hence, if $\lambda''\alpha''+e_i\in H^{(l)}$, so that $\xi^{(l)}\cdot(\lambda''\alpha''+e_i)=0$, then

$$|\lambda''|=\frac{|\xi^{(l)}\cdot e_i|}{|\xi^{(l)}\cdot\alpha''|}\leqq(1+n\mathscr{D})^{n+1}.$$ □

PROPOSITION 5.4. *Assume $R(u)\not\equiv 0$. Let α' be as chosen in Step 3 of the algorithm. Then for all l,*

$$|\xi^{(l)}\cdot\alpha'|=\Omega(\|\xi^{(l)}\|/[n\mathscr{D}]^{2n\mathscr{D}}). \tag{5.2}$$

Moreover, for any $\beta\in\mathbb{C}^{n+1}$, letting $\lambda^{(l)},\lambda^{(m)}\in\mathbb{C}$ satisfy $\lambda^{(l)}\alpha'+\beta\in H^{(l)}$, $\lambda^{(m)}\alpha'+\beta\in H^{(m)}$, we have

$$|\lambda^{(l)}|=O(\|\beta\|[n\mathscr{D}]^{2n\mathscr{D}}), \tag{5.3}$$

$$|\lambda^{(l)}-\lambda^{(m)}|=O(\|\beta\|[n\mathscr{D}]^{5n\mathscr{D}}\operatorname{dis}(\xi^{(l)},\xi^{(m)})). \tag{5.4}$$

Proof. For any polynomial $\sum_{k=0}^{d}a_k\lambda^k$, $a_d\neq 0$, with zeros $\lambda_1,\cdots,\lambda_d$ we of course have $|a_k/a_d|=|\sum\lambda_{i_1}\cdots\lambda_{i_k}|$, where the summation is over all tuples $i_1<i_2<\cdots<i_k$. Hence if $|\lambda_i|\leqq R$ for all i, then $|a_k/a_d|\leqq\binom{d}{k}R^k$. In particular, letting j'' be as in Lemma 5.3, then for j' as in Step 3 of the algorithm

$$\max_{i,k}\left|\frac{a_k(i,j')}{a_{\mathscr{D}}(i,j')}\right|\leqq\max_{i,k}\left|\frac{a_k(i,j'')}{a_{\mathscr{D}}(i,j'')}\right|\leqq(1+n\mathscr{D})^{(n+1)\mathscr{D}}. \tag{5.5}$$

For each zero λ' of a polynomial $\sum_{k=0}^{d}a_k\lambda^k$, $a_d\neq 0$, one has the property that $|\lambda'|\leqq 1+\max\{|a_k/a_d|;\ 0\leqq k<d\}$ (cf. Marden [7, Thm. 27.2]). In particular, using (5.5), we find that for each i, every zero λ' of $R(\lambda\alpha'+e_1)$ satisfies

$$|\lambda'|=O([n\mathscr{D}]^{(n+1)\mathscr{D}}). \tag{5.6}$$

Fix l and assume that $|\xi_i^{(l)}| \geq \|\xi^{(l)}\|/\sqrt{n+1}$ (this is certainly true for some i). Assume $\lambda^{(l)}\alpha' + e_i \in H^{(l)}$. Then $|\lambda^{(l)}||\xi^{(l)} \cdot \alpha'| = |\xi^l \cdot e_i| \geq \|\xi^{(l)}\|/\sqrt{n+1}$. Thus, using (5.6),

$$|\xi^{(l)} \cdot \alpha'| = \Omega(\|\xi^{(l)}\|\sqrt{n}[n\mathscr{D}]^{(n+1)\mathscr{D}}),$$

from which (5.2) is immediate.

Since $R(\lambda\alpha' + e_i)$ is of degree exactly $\mathscr{D}$ (for any i) by choice of α', Lemma 5.1 implies $R(\lambda\alpha' + \beta)$ is of degree exactly $\mathscr{D}$ for any $\beta \in \mathbb{C}^{n+1}$. Fix β and assume $\lambda^{(l)}\alpha' + \beta \in H^{(l)}$. Then

$$|\lambda^{(l)}|\,|\xi^{(l)} \cdot \alpha'| = |\xi^{(l)} \cdot \beta| \leq \|\xi^{(l)}\|\,\|\beta\|.$$

Substituting (5.2) into this inequality gives (5.3).

Finally, let $\mathscr{Z}^{(l)} = w_1\xi^{(l)}$, $\mathscr{Z}^{(m)} = w_2\xi^{(m)}$ ($w_1, w_2 \in \mathbb{C}$) be such that $|\mathscr{Z}^{(l)}| = |\mathscr{Z}^{(m)}| = 1$ and $\|\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)}\| = \operatorname{dis}(\xi^{(l)}, \xi^{(m)})$. Then

$$\begin{aligned} 0 &= \mathscr{Z}^{(l)} \cdot (\lambda^{(l)}\alpha' + \beta) \\ &= (\mathscr{Z}^{(m)} + [\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)}]) \cdot ([\lambda^{(m)}\alpha' + \beta] + [\lambda^{(l)} - \lambda^{(m)}]\alpha') \\ &= [(\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)})] \cdot [(\lambda^{(m)}\alpha' + \beta)] + [(\lambda^{(l)} - \lambda^{(m)})]\mathscr{Z}^{(l)} \cdot \alpha'. \end{aligned}$$

However, using (5.3) and $\|\alpha'\| < (n\mathscr{D})^{n+1}$, we have

$$|[(\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)})] \cdot [(\lambda^{(m)}\alpha' + \beta)]| = O([n\mathscr{D}]^{3n\mathscr{D}}\|\beta\| \cdot \operatorname{dis}(\xi^{(l)}, \xi^{(m)})),$$

and by (5.2),

$$|\lambda^{(l)} - \lambda^{(m)}|\,|\mathscr{Z}^{(l)} \cdot \alpha'| = \Omega(|\lambda^{(l)} - \lambda^{(m)}|/[n\mathscr{D}]^{2n\mathscr{D}}).$$

Now (5.4) follows. □

The next lemma will be used in proving Proposition 4.3.

LEMMA 5.5. *Assume $R(u) \not\equiv 0$. Let α' be as in Step 3 of the algorithm. For some $k'' \in \{0, 1, \cdots, n\mathscr{D}(\mathscr{D}-1)/2\}$, $\beta'' = \mu(k'')$ has the property that for all l and m, if $\lambda^{(l)}\alpha' + \beta'' \in H^{(l)}$, $\lambda^{(m)}\alpha' + \beta'' \in H^{(m)}$, then*

$$\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) = O([n\mathscr{D}]^{3n+2}|\lambda^{(l)} - \lambda^{(m)}|).$$

Proof. For all pairs $l < m$, $l, m \in \{1, \cdots, \mathscr{D}\}$ such that $H^{(l)} \neq H^{(m)}$, let

$$M^{(l,m)} = \{\lambda\alpha' + y; \qquad \lambda \in \mathbb{C}, y \in H^{(l)} \cap H^{(m)}\}.$$

Let $\mathscr{M} = \{M^{(l,m)}\}$. Let $k'' \in \{0, 1, \cdots, n\mathscr{D}(\mathscr{D}-1)/2\}$ denote an integer such that $\mu(k'')$ satisfies the conclusion of Lemma 5.2 with this choice of $\mathscr{M}$. Let $\beta'' = \mu(k'')$.

If $H^{(l)} = H^{(m)}$, then the bound on $|\lambda^{(l)} - \lambda^{(m)}|$ provided by the lemma is trivial. So assume $H^{(l)} \neq H^{(m)}$. By a change of coordinates $x \mapsto Qx$, where Q is a complex unitary matrix (to preserve distances), and by replacement of $\xi^{(l)}$ (respectively, $\xi^{(m)}$, α', β'') with $Q^{-1}\xi^{(l)}$ (respectively, $Q^{-1}\xi^{(m)}$, $Q\alpha'$, $Q\beta''$) we may assume without loss of generality that $\xi_3^{(l)} = 0, \cdots, \xi_{n+1}^{(l)} = 0$, $\xi_3^{(m)} = 0, \cdots, \xi_{n+1}^{(m)} = 0$ and

$$H^{(l)} \cap H^{(m)} = \{x \in \mathbb{C}^{n+1};\ x_1 = x_2 = 0\}.$$

Let $x = \lambda^{(l)}\alpha' + \beta'' \in H^{(l)}$. Then by the definition of $M^{(l,m)}$ and the choice of β'' (satisfying the conclusion of Lemma 5.2), we must have the distance from x to $H^{(l)} \cap H^{(m)}$ bounded below by $1/(1+n\mathscr{D}^2)^{n+1}$, that is,

$$(|x_1|^2 + |x_2|^2)^{1/2} \geq 1/(1+n\mathscr{D}^2)^{n+1}. \tag{5.7}$$

Let $\mathscr{Z}^{(l)} = w_1\xi^{(l)}$, $\mathscr{Z}^{(m)} = w_2\xi^{(m)}$ be such that $\|\mathscr{Z}^{(l)}\| = \|\mathscr{Z}^{(m)}\| = 1$ and dis $(\xi^{(l)}, \xi^{(m)}) = \|\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)}\|$. Since $\mathscr{Z}^{(l)} \cdot x = 0$, $\mathscr{Z}^{(m)} \cdot [x + (\lambda^{(m)} - \lambda^{(l)})\alpha'] = 0$ and the last $n-1$ coordinates of both $\mathscr{Z}^{(l)}$ and $\mathscr{Z}^{(m)}$ are zero, we have

$$\begin{bmatrix} \mathscr{Z}_1^{(l)} & \mathscr{Z}_2^{(l)} \\ \mathscr{Z}_1^{(m)} & \mathscr{Z}_2^{(m)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = (\lambda^{(l)} - \lambda^{(m)})(\mathscr{Z}^{(m)} \cdot \alpha') \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

solving for x via Cramer's rule and then using (5.7) gives

$$\frac{|\lambda^{(l)} - \lambda^{(m)}|\,|\mathscr{Z}^{(m)} \cdot \alpha'|}{|\mathscr{Z}_1^{(l)}\mathscr{Z}_2^{(m)} - \mathscr{Z}_2^{(l)}\mathscr{Z}_1^{(m)}|} \geqq \frac{1}{(1 + n\mathscr{D}^2)^{n+1}}. \tag{5.8}$$

Noting that $\{\omega(-\bar{\mathscr{Z}}_2^{(m)}, \bar{\mathscr{Z}}_1^{(m)});\ \omega \in \mathbb{C}\}$ is the orthogonal complement of $\{\omega(\mathscr{Z}_1^{(m)}, \mathscr{Z}_2^{(m)});\ \omega \in \mathbb{C}\}$ when considered as subspaces of $\mathbb{R}^4$, we have

$$(\mathscr{Z}_1^{(l)}, \mathscr{Z}_2^{(l)}) = v(\mathscr{Z}_1^{(m)}, \mathscr{Z}_2^{(m)}) + w(-\bar{\mathscr{Z}}_2^{(m)}, \bar{\mathscr{Z}}_1^{(m)}),$$

where $v \geqq 0$, $w \in \mathbb{C}$ and $v^2 + |w|^2 = 1$. (In stating $v \geqq 0$ we are using the fact that $\|\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)}\| = \text{dis}\,(\mathscr{Z}^{(l)}, \mathscr{Z}^{(m)})$.) Substituting for $\mathscr{Z}^{(l)}$ in the denominator of (5.8) gives

$$\frac{|\lambda^{(l)} - \lambda^{(m)}|\,|\mathscr{Z}^{(m)} \cdot \alpha'|}{|w|} \geqq \frac{1}{(1 + n\mathscr{D}^2)^{n+1}}. \tag{5.9}$$

Observe that

$$\text{dis}^2\,(\xi^{(l)}, \xi^{(m)}) = \|\mathscr{Z}^{(l)} - \mathscr{Z}^{(m)}\|^2 = (1 - v)^2 + |w|^2 \leqq 2|w|^2,$$

where the inequality follows from $v^2 + |w|^2 = 1$ and $v \geqq 0$. Substituting for $|w|$ in (5.9) and using $|\mathscr{Z}^{(m)} \cdot \alpha'| \leqq \|\alpha'\| \leqq \sqrt{n+1}(n\mathscr{D})^n$ gives the proposition. □

We can now give the proof of Proposition 4.3, which relies on the notation introduced just prior to that proposition. For the reader's convenience, we restate the proposition as

PROPOSITION 5.6. *For all* $\delta = \Omega([n\mathscr{D}]^{10n\mathscr{D}})$ *and for* ε'' *as given at the end of Step* 5,

$$\gamma^{(l)} \in P^{[h]} \Rightarrow |\xi^{(l)} \cdot x^{[h]}| = O(\varepsilon''[n\mathscr{D}]^{n+1}\|\xi^{(l)}\|), \tag{5.10}$$

$$\gamma^{(l)} \notin P^{[h]} \Rightarrow |\xi^{(l)} \cdot x^{[h]}| = \Omega(\delta\varepsilon''\|\xi^{(l)}\|/[n\mathscr{D}]^{2n\mathscr{D}}), \tag{5.11}$$

$$\gamma^{(l)}, \gamma^{(m)} \in P^{[h]} \Rightarrow \text{dis}\,(\xi^{(l)}, \xi^{(m)}) = O(\varepsilon''[n\mathscr{D}]^{3n+2}). \tag{5.12}$$

Proof. We begin by recalling that for each $k = 0, 1, \cdots, \mathscr{D}(\mathscr{D}-1)/2$, Step 5 partitions the approximations $\gamma_i(k)$ into "clusters" $P^{[h]}(k)$, $h = 1, \cdots, h(k)$, where

$$\gamma_i(k), \gamma_j(k) \in P^{[h]}(k) \Rightarrow |\gamma_i(k) - \gamma_j(k)| \leqq \varepsilon'', \tag{5.13}$$

$$\gamma_i(k) \in P^{[h]}(k),\ \gamma_j(k) \notin P^{[h]}(k) \Rightarrow |\gamma_i(k) - \gamma_j(k)| > \delta\varepsilon''. \tag{5.14}$$

To prove (5.10), note that since $x^{[h]} = \gamma^{(m)}\alpha' + \beta'$ for some $\gamma^{(m)} \in P^{[h]}$, where $\gamma^{(m)}$ approximates $\lambda^{(l)}$ within distance ε', we have

$$\begin{aligned} |\xi^{(l)} \cdot x^{[h]}| &\leqq |\xi^{(l)} \cdot (\gamma^{(m)} - \gamma^{(l)})\alpha'| + |\xi^{(l)} \cdot (\gamma^{(l)} - \lambda^{(l)})\alpha'| + |\xi^{(l)} \cdot (\lambda^{(l)}\alpha' + \beta')| \\ &\leqq (\varepsilon'' + \varepsilon')\|\xi^{(l)}\|\,\|\alpha'\| + 0 \quad \text{(using (5.13))}. \end{aligned}$$

Since $\varepsilon' \leqq \varepsilon''$ by the construction of Step 5 and $\|\alpha'\| < (n\mathscr{D})^{n+1}$, (5.10) follows.

Now we prove (5.11). Again assume $x^{[h]} = \gamma^{(m)}\alpha' + \beta'$, but now assume $\gamma^{(l)} \notin P^{[h]}$. Then $|\gamma^{(m)} - \gamma^{(l)}| > \delta\varepsilon''$ by (5.14). Using this and (5.2) along with $|\gamma^{(l)} - \lambda^{(l)}| \leqq \varepsilon'$ and

$\|\alpha'\| < (n\mathscr{D})^{n+1}$, we have

$$\begin{aligned}|\xi^{(l)} \cdot x^{[h]}| &= |\xi^{(l)} \cdot x^{[h]}| + |\xi^{(l)} \cdot (\lambda^{(l)}\alpha' + \beta')| \\ &= |\xi^{(l)} \cdot (\gamma^{(m)} - \lambda^{(l)})\alpha'| \\ &\geqq |\xi^{(l)} \cdot (\gamma^{(m)} - \gamma^{(l)})\alpha'| - |\xi^{(l)} \cdot (\gamma^{(l)} - \lambda^{(l)})\alpha'| \\ &= \Omega(\delta\varepsilon''\|\xi^{(l)}\|/[n\mathscr{D}]^{2n\mathscr{D}}) - O(\varepsilon'\|\xi^{(l)}\|(n\mathscr{D})^{n+1}).\end{aligned}$$

Assuming $\delta = \Omega([n\mathscr{D}]^{3n\mathscr{D}})$ and using $\varepsilon'' \geqq \varepsilon'$, (5.11) follows.

Now we turn to proving (5.12). Let β'' be as in Lemma 5.5. Let $P^{[j]}(k''), j = 1, \cdots, h(k'')$ be the sets determined for β'' at the end of Step 5. Assuming $\lambda^{(l)}(k'') \in \mathbb{C}$ satisfies $\lambda^{(l)}(k'')\alpha' + \beta'' \in H^{(l)}$, let $\gamma^{(l)}(k'')$ denote the ε'-approximation to $\lambda^{(l)}(k'')$ obtained in Step 4. We will show that if $\gamma^{(l)}(k''), \gamma^{(m)}(k'') \in P^{[j]}(k'')$ for some j, then $\gamma^{(l)}, \gamma^{(m)} \in P^{[h]}$ for some h. However, since the union $\cup_j P^{[j]}(k'')$ contains the same number of elements as the disjoint union $\cup_h P^{[h]}$, and since $h(k') \geqq h(k'')$ by choice of β'', it follows that the converse is also true: that is, if $\gamma^{(l)}, \gamma^{(m)} \in P^{[h]}$ for some h, then $\gamma^{(l)}(k''), \gamma^{(m)}(k'') \in P^{[j]}(k'')$ for some j. Then $|\gamma^{(l)}(k'') - \gamma^{(m)}(k'')| \leqq \varepsilon''$ by (5.13) and hence $|\lambda^{(l)}(k'') - \lambda^{(m)}(k'')| \leqq 2\varepsilon' + \varepsilon''$. Thus, since β'' satisfies the conclusion of Lemma 5.5,

$$\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) = O(\varepsilon''[n\mathscr{D}]^{3n+2}),$$

establishing (5.12).

Finally, we prove that if $\gamma^{(l)}(k''), \gamma^{(m)}(k'') \in P^{[j]}(k'')$ for some j, then $\gamma^{(l)}, \gamma^{(m)} \in P^{[h]}$ for some h. Assume otherwise. Then $|\gamma^{(l)} - \gamma^{(m)}| > \delta\varepsilon''$ by (5.14) and hence $|\lambda^{(l)} - \lambda^{(m)}| > \delta\varepsilon'' - 2\varepsilon'$. Assuming $\delta \geqq 3$ and thus $\delta\varepsilon'' - 2\varepsilon' \geqq \delta\varepsilon''/3$, and using $\|\beta''\| < (n\mathscr{D}^2)^n$, we conclude from (5.4) that

$$\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) = \Omega(\delta\varepsilon''/[n\mathscr{D}]^{7n\mathscr{D}}). \tag{5.15}$$

However, since $\gamma^{(l)}(k''), \gamma^{(m)}(k'') \in P^{[j]}(k'')$, we have that $|\lambda^{(l)}(k'') - \lambda^{(m)}(k'')| \leqq 2\varepsilon' + \varepsilon''$, and hence, since the conclusion of Lemma 5.5 holds for β'',

$$\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) = O(\varepsilon''[n\mathscr{D}]^{3n+2}). \tag{5.16}$$

But for $\delta = \Omega([n\mathscr{D}]^{10n\mathscr{D}})$, (5.15) and (5.16) contradict one another, concluding the proof of the proposition. □

Finally, we turn to proving Propositions 4.4 and 4.5. We begin with a lemma.

LEMMA 5.7. *Fix* $\xi^{(m)}$ *and assume* $\xi_i^{(m)} \neq 0$. *If* $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq |\xi_i^{(m)}|/2\|\xi^{(m)}\|$, *then*

$$\frac{|\xi_i^{(l)}|}{\|\xi^{(l)}\|} \geqq \frac{|\xi_i^{(m)}|}{2\|\xi^{(m)}\|}, \tag{5.17}$$

$$\left\| \frac{\xi^{(l)}}{\xi_i^{(l)}} - \frac{\xi^{(m)}}{\xi_i^{(m)}} \right\| \leqq \frac{4\|\xi^{(m)}\|^2 \cdot \operatorname{dis}(\xi^{(l)}, \xi^{(m)})}{|\xi_i^{(m)}|^2}. \tag{5.18}$$

Proof. Assume $\mathscr{X}^{(l)} = w_1\xi^{(l)}$, $\mathscr{X}^{(m)} = w_2\xi^{(m)}$ satisfy $\|\mathscr{X}^{(l)}\| = \|\mathscr{X}^{(m)}\| = 1$, and $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) = \|\mathscr{X}^{(l)} - \mathscr{X}^{(m)}\|$. Since $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq |\mathscr{X}_i^{(m)}|/2$, it is easily shown that $|\mathscr{X}_i^{(l)}| \geqq |\mathscr{X}_i^{(m)}|/2$, and hence (5.17).

Note that

$$\begin{aligned}\left\| \frac{\xi^{(l)}}{\xi_i^{(l)}} - \frac{\xi^{(m)}}{\xi_i^{(m)}} \right\| &= \left\| \frac{\mathscr{X}^{(l)}}{\mathscr{X}_i^{(l)}} - \frac{\mathscr{X}^{(m)}}{\mathscr{X}_i^{(m)}} \right\| \\ &= \left\| \frac{\mathscr{X}^{(l)}}{\mathscr{X}_i^{(l)}} - \frac{\mathscr{X}^{(m)}}{\mathscr{X}_i^{(l)}} + \left(\frac{\mathscr{X}_i^{(m)} - \mathscr{X}_i^{(l)}}{\mathscr{X}_i^{(l)}\mathscr{X}_i^{(m)}} \right) \mathscr{X}^{(m)} \right\| \\ &\leqq \frac{1}{|\mathscr{X}_i^{(l)}|} \left[\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) + \left(\frac{\operatorname{dis}(\xi^{(l)}, \xi^{(m)})}{|\mathscr{X}_i^{(m)}|} \right) \right].\end{aligned}$$

However, $2|\mathscr{X}_i^{(l)}| \geqq |\mathscr{X}_i^{(m)}| = |\xi_i^{(m)}|/\|\xi^{(m)}\|$. Now (5.18) follows. □

Here is Proposition 4.4 restated as Proposition 5.8.

PROPOSITION 5.8. *Assume that for all* $l, m \in \{1, \cdots, \mathscr{D}\}$ *we have that* $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq \varepsilon'''$. *For all* $\varepsilon''' = O(1/\sqrt{n})$, *the following is then true. Let* i' *be an index satisfying*

$$\left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}}}\bigg|_{u=0}\right| = \max_i \left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_i^{\mathscr{D}}}\bigg|_{u=0}\right|$$

and let $\mathbb{X} \in \mathbb{C}^{n+1}$ *be the vector*

$$\mathbb{X}_i = \frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}-1} \partial u_i}\bigg|_{u=0} \qquad i = 1, \cdots, n+1.$$

Then $\operatorname{dis}(\mathbb{X}, \xi^{(l)}) = O(n\varepsilon''')$ *for all* l.

Proof. Fix $m \in \{1, \cdots, \mathscr{D}\}$ and let i'' denote an index satisfying $|\xi_{i''}^{(m)}| \geqq \|\xi^{(m)}\|/\sqrt{n+1}$. Then, by (5.17), assuming $\varepsilon''' \leqq 1/2\sqrt{n+1}$, we have that $|\xi_{i''}^{(l)}| \geqq \|\xi^{(l)}\|/2\sqrt{n+1}$ for all $l \in \{1, \cdots, \mathscr{D}\}$.

Next, note that by definition of i' and using $R(u) = \prod_l (\xi^{(l)} \cdot u)$,

$$\mathscr{D}! \left|\prod_l \xi_{i'}^{(l)}\right| = \left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}}}\bigg|_{u=0}\right| \geqq \left|\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i''}^{\mathscr{D}}}\bigg|_{u=0}\right| = \mathscr{D}! \left|\prod_l \xi_{i''}^{(l)}\right|.$$

Hence, for at least one $k \in \{1, \cdots, \mathscr{D}\}$, we have that $|\xi_{i'}^{(k)}| \geqq |\xi_{i''}^{(k)}|$. Since $|\xi_{i''}^{(k)}| \geqq \|\xi^{(k)}\|/2\sqrt{n+1}$, we thus have $|\xi_{i'}^{(k)}| \geqq \|\xi^{(k)}\|/2\sqrt{n+1}$. It thus follows from (5.17) that if $\varepsilon''' \leqq 1/4\sqrt{n+1}$, then

$$|\xi_{i'}^{(l)}| \geqq \|\xi^{(l)}\|/4\sqrt{n+1} \quad \text{for all } l. \tag{5.19}$$

Consider the identity

$$\frac{\partial^{\mathscr{D}} R(u)}{\partial u_{i'}^{\mathscr{D}-1} \partial u_i}\bigg|_{u=0} = (\mathscr{D}-1)! \sum_l \left(\xi_i^{(l)} \prod_{m \neq l} \xi_{i'}^{(m)}\right).$$

Defining $\mathbb{X}$ as in the statement of the proposition, we thus have

$$\frac{\mathbb{X}}{\prod_l \xi_{i'}^{(l)}} = (\mathscr{D}-1)! \sum_l \frac{\xi^{(l)}}{\xi_{i'}^{(l)}}. \tag{5.20}$$

However, using (5.18) and (5.19), if $\varepsilon''' \leqq 1/8\sqrt{n+1}$ we have that for any m,

$$\left\| \mathscr{D} \frac{\xi^{(m)}}{\xi_{i'}^{(m)}} - \sum_l \frac{\xi^{(l)}}{\xi_{i'}^{(l)}} \right\| \leqq 64(\mathscr{D}-1)(n+1)\varepsilon'''.$$

Hence, for any m, (5.20) gives

$$\left\| \frac{\mathbb{X}}{\prod_l \xi_{i'}^{(l)}} - \mathscr{D}! \frac{\xi^{(m)}}{\xi_{i'}^{(m)}} \right\| \leqq 64(\mathscr{D}!)(n+1)\varepsilon'''.$$

Since $\|\xi^{(m)}\|/|\xi_{i'}^{(m)}| \geqq 1$, it follows that $\operatorname{dis}(\mathbb{X}, \xi^{(m)}) = O(n\varepsilon''')$. □

Finally, we prove Proposition 4.5 restated as Proposition 5.9.

PROPOSITION 5.9. *Let* $S \subset \{1, \cdots, \mathscr{D}\}$ *contain* N *elements, where* $0 < N < \mathscr{D}$. *Assume that for all* $l, m \in S$, *we have* $\operatorname{dis}(\xi^{(l)}, \xi^{(m)}) \leqq \varepsilon'''$. *Let* $x \in \mathbb{C}^{n+1}$, $x \neq 0$. *Assume*

that if $l \in S$, *then* $|\xi^{(l)} \cdot x| \leqq \rho_1 \|\xi^{(l)}\| \, \|x\|$, *and assume that if* $l \notin S$, *then* $|\xi^{(l)} \cdot x| \geqq \rho_2 \|\xi^{(l)}\| \, \|x\|$, *where* $\rho_2 > 0$. *Then for all* $\varepsilon''' = O(1/\sqrt{n})$ *and for all* $\rho_1/\rho_2 = O(\varepsilon'''/\mathscr{D}!n)$, *the following is true. Let* i' *be an index satisfying*

$$\left| \frac{\partial^N R(u)}{\partial u_{i'}^N}\bigg|_{u=x} \right| = \max_i \left| \frac{\partial^N R(u)}{\partial u_i^N}\bigg|_{u=x} \right|,$$

and let $\mathbb{X} \in \mathbb{C}^{n+1}$ *be the vector*

$$\mathbb{X}_i = \frac{\partial^N R(u)}{\partial u_{i'}^{N-1} \partial u_i}\bigg|_{u=x} \qquad i = 1, \cdots, n+1.$$

Then $\operatorname{dis}(\mathbb{X}, \xi^{(l)}) = O(n\mathscr{D}\varepsilon''')$ *for all* $l \in S$.

Proof. The proof is analogous to, but much more complicated than, the proof of Proposition 5.8.

We begin by showing that if ε'' and ρ_1/ρ_2 are as small as certain prescribed quantities, then $|\xi_{i'}^{(l)}| \geqq \|\xi^{(l)}\|/8\sqrt{n+1}$ for all $l \in S$.

Fix $m \in S$ and let i'' denote an index satisfying $|\xi_{i''}^{(m)}| \geqq \|\xi^{(m)}\|/\sqrt{n+1}$. Then, by (5.17), if $\varepsilon''' \leqq 1/2\sqrt{n+1}$ we have that

$$|\xi_{i''}^{(l)}| \geqq \|\xi^{(l)}\|/2\sqrt{n+1} \quad \text{for all } l \in S.$$

Consider the identity

$$\frac{\partial^N R}{\partial u_{i''}^N}\bigg|_{u=x} = \sum \left(\prod_{l \in L} \xi_{i''}^{(l)} \right) \left(\prod_{l \notin L} \xi^{(l)} \cdot x \right) \tag{5.21}$$

where the summation is over all ordered N-tuples $(l_1, \cdots, l_N)$ of distinct indices from $\{1, \cdots, \mathscr{D}\}$, and where $l \in L$ is used to denote $l \in \{l_1, \cdots, l_N\}$. Let $\sum^*$ denote summation over the $N!$ N-tuples with $l_1, \cdots, l_N \in S$, and let $\sum^{**}$ denote summation over the remaining tuples. Then from (5.21)

$$\frac{1}{(\prod_{l \in S} \xi_{i''}^{(l)})(\prod_{l \notin S} \xi^{(l)} \cdot x)} \cdot \frac{\partial^N R}{\partial u_{i''}^N}\bigg|_{u=x} = N! + \sum^{**} \left(\prod_{l \in S \setminus L} \frac{\xi^{(l)} \cdot x}{\xi_{i''}^{(l)}} \right) \left(\prod_{l \in L \setminus S} \frac{\xi_{i''}^{(l)}}{\xi^{(l)} \cdot x} \right). \tag{5.22}$$

Note that (i) $|\xi^{(l)} \cdot x / \xi_{i''}^{(l)}| \leqq 2\rho_1\sqrt{n+1}\|x\|$ for all $l \in S$ since $|\xi_{i''}^{(l)}| \geqq \|\xi^{(l)}\|/2\sqrt{n+1}$ for all $l \in S$; (ii) $|\xi_{i''}^{(l)}/\xi^{(l)} \cdot x| \leqq 1/\rho_2\|x\|$ for all $l \notin S$; (iii) $L \setminus S$ and $S \setminus L$ have the same number of elements; (iv) at least one $l \in S$ satisfies $l \notin L$ for each of the tuples defining $\sum^{**}$; (v) $\sum^{**}$ is a summand over fewer than $\mathscr{D}!$ N-tuples; and (vi)

$$\left| \frac{\partial^N R}{\partial u_{i'}^N}\bigg|_{u=x} \right| \geqq \left| \frac{\partial^N R}{\partial u_{i''}^N}\bigg|_{u=x} \right|$$

by choice of i'. Assuming $\rho_1 \leqq \rho_2/8\sqrt{n+1}\mathscr{D}!$, it follows from (5.22) that

$$\left| \frac{1}{(\prod_{l \in S} \xi_{i''}^{(l)})(\prod_{l \notin S} \xi^{(l)} \cdot x)} \cdot \frac{\partial^N R}{\partial u_{i'}^N}\bigg|_{u=x} \right| \geqq N! - \frac{1}{4}.$$

Relying on the identity (5.21) with i' replacing i'', this becomes

$$\left| \sum^* \left(\prod_{l \in S} \frac{\xi_{i'}^{(l)}}{\xi_{i''}^{(l)}} \right) + \sum^{**} \left(\prod_{l \in S \cap L} \frac{\xi_{i'}^{(l)}}{\xi_{i''}^{(l)}} \right) \left(\prod_{l \in S \setminus L} \frac{\xi^{(l)} \cdot x}{\xi_{i''}^{(l)}} \right) \left(\prod_{l \in L \setminus S} \frac{\xi_{i'}^{(l)}}{\xi^{(l)} \cdot x} \right) \right| \geqq N! - \frac{1}{4}. \tag{5.23}$$

Making use of the above observations (i)-(v) with i' replacing i'' in (ii), as well as $|\xi_{i'}^{(l)}/\xi_{i''}^{(l)}| \leqq 2\sqrt{n+1}$ for all $l \in S$ (since $|\xi_{i''}^{(l)}| \geqq \|\xi^{(l)}\|/2\sqrt{n+1}$ for all $l \in S$), and assuming that $\rho_1 \leqq \rho_2/4(2\sqrt{n+1})^N \mathscr{D}!$, it follows from (5.23) that

$$N!\left|\prod_{l\in S}\frac{\xi_{i'}^{(l)}}{\xi_{i''}^{(l)}}\right| \geqq N! - \frac{1}{2}.$$

Hence, for at least one $m \in S$ we must have $|\xi_{i'}^{(m)}| \geqq |\xi_{i''}^{(m)}|/2$. Since $|\xi_{i''}^{(m)}| \geqq \|\xi^{(m)}\|/2\sqrt{n+1}$, it now follows from (5.17) that if $\varepsilon''' < 1/8\sqrt{n+1}$, then

$$|\xi_{i'}^{(l)}| \geqq \|\xi^{(l)}\|/8\sqrt{n+1} \quad \text{for all } l \in S. \tag{5.24}$$

Consider the identity, for any i,

$$\begin{aligned}&\frac{1}{(\prod_{l\in S}\xi_{i'}^{(l)})(\prod_{l\notin S}\xi^{(l)}\cdot x)}\frac{\partial^N R}{\partial u_{i'}^{N-1}\,du_i}\bigg|_{u=x}\\ &\quad = (N-1)!\sum_{l\in S}\frac{\xi_i^{(l)}}{\xi_{i'}^{(l)}} + \sum{}^{**}\left(\prod_{l\in S\setminus L}\frac{\xi^{(l)}\cdot x}{\xi_{i'}^{(l)}}\right)\left(\prod_{l\in L^{\square}\setminus S}\frac{\xi_{i'}^{(l)}}{\xi^{(l)}\cdot x}\right)\cdot g(x, l_N),\end{aligned} \tag{5.25}$$

where $L^{\square} = \{l_1, \cdots, l_{N-1}\}$ and

$$g(x, l_N) = \begin{cases} \xi_i^{(l_N)}/\xi_{i'}^{(l_N)} & \text{if } l_N \in S \\ \xi_i^{(l_N)}/\xi^{(l_N)}\cdot x & \text{if } l_N \notin S. \end{cases}$$

Making use of the above observations (iii)-(v) and (5.23), we find from (5.25) that if $\rho_1/\rho_2 \leqq \varepsilon'''/8\sqrt{n+1}$, then for $\mathbb{X}$ as defined in the statement of the proposition, we have

$$\left\|\frac{\mathbb{X}}{(\prod_{l\in S}\xi_{i'}^{(l)})(\prod_{l\notin S}\xi^{(l)}\cdot x)} - (N-1)!\sum_{l\in S}\frac{\xi^{(l)}}{\xi_{i'}^{(l)}}\right\| \leqq \varepsilon'''. \tag{5.26}$$

However, using (5.18) and (5.24), if $\varepsilon''' \leqq 1/16\sqrt{n+1}$ we have that for any $m \in S$,

$$\left\|\frac{N}{\xi_{i'}^{(m)}}\xi^{(m)} - \sum_{l\in S}\frac{\xi^{(l)}}{\xi_{i'}^{(l)}}\right\| \leqq 256(N-1)(n+1)\varepsilon'''.$$

Hence, for any $m \in S$, (5.26) gives

$$\left\|\frac{\mathbb{X}}{(\prod_{l\in S}\xi_{i'}^{(l)})(\prod_{l\notin S}\xi^{(l)}\cdot x)} - \frac{N!\,\xi^{(m)}}{\xi_{i'}^{(m)}}\right\| \leqq \varepsilon''' + 16(N-1)(n+1)\varepsilon'''.$$

Since $\|\xi^{(m)}\|/|\xi_{i'}^{(m)}| \geqq 1$, the proposition follows. □

6. Appendix. Here we prove the claim (1.1). Let $\varepsilon' = \varepsilon/4(R+1)^2$, $0 < \varepsilon \leqq R$, and assume that $\mathbb{X}^{(i)}$, $i = 1, \cdots, \mathscr{D}$ are ε'-approximations to all of the zero lines of F. In the notation of § 1, $\|\mathbb{X}^{(i)}/\|\mathbb{X}^{(i)}\| - \xi^{(i)}/\|\xi^{(i)}\|\| \leqq \varepsilon'$ where the zero lines of F are precisely the lines $\{\lambda\xi^{(i)};\ \lambda \in \mathbb{C}\}$.

For each $\xi^{(i)}$ such that $\xi_{n+1}^{(i)} \neq 0$, let

$$\tilde{\xi}^{(i)} = \left(\frac{\xi_1^{(i)}}{\xi_{n+1}^{(i)}}, \cdots, \frac{\xi_n^{(i)}}{\xi_{n+1}^{(i)}}\right).$$

Then $\tilde{\xi}^{(i)}$ is the zero of f corresponding to the solution line $\{\lambda\xi^{(i)};\ \lambda \in \mathbb{C}\}$ of F. Define $\tilde{\mathbb{X}}^{(i)}$ analogously.

We show that

$$\|\tilde{\xi}^{(i)}\| \leqq R \Rightarrow \frac{|\mathbb{X}_{n+1}^{(i)}|}{\|\mathbb{X}^{(i)}\|} \geqq \frac{3}{4(R+1)}, \tag{6.1}$$

and

$$\frac{|\mathbb{X}_{n+1}^{(i)}|}{\|\mathbb{X}^{(i)}\|} \geqq \frac{3}{4(R+1)} \Rightarrow \|\tilde{\mathbb{X}}^{(i)} - \tilde{\xi}^{(i)}\| \leqq \varepsilon. \tag{6.2}$$

Together, (6.1) and (6.2) imply that the points $\{\tilde{\mathbb{X}}^{(i)}; |\mathbb{X}_{n+1}^{(i)}|/\|\mathbb{X}^{(i)}\| \geqq 3/4(R+1)\}$ are a solution for the (ε, R)-approximation problem for f.

To prove (6.1), first note that $\|\tilde{\xi}^{(i)}\| \leqq R$ if and only if $\|\xi^{(i)}\|^2 \leqq (R^2+1)|\xi_{n+1}^{(i)}|^2$. Thus, if $\|\tilde{\xi}^{(i)}\| \leqq R$, then

$$\frac{|\mathbb{X}_{n+1}^{(i)}|}{\|\mathbb{X}^{(i)}\|} \geqq \frac{|\xi_{n+1}^{(i)}|}{\|\xi^{(i)}\|} - \varepsilon' \geqq \left(1 - \frac{\varepsilon}{4(R+1)}\right)\frac{1}{R+1} \geqq \frac{3}{4(R+1)}.$$

In proving (6.2), let $\hat{\mathbb{X}}^{(i)} = \mathbb{X}^{(i)}/\|\mathbb{X}^{(i)}\|$ and $\hat{\xi}^{(i)} = \xi^{(i)}/\|\xi^{(i)}\|$. Then assuming that $\hat{\mathbb{X}}_{n+1}^{(i)} \geqq 3/4(R+1)$, we have

$$\begin{aligned}
\|\tilde{\mathbb{X}}^{(i)} - \tilde{\xi}^{(i)}\| &= \left\| \frac{\hat{\mathbb{X}}^{(i)}}{\hat{\mathbb{X}}_{n+1}^{(i)}} - \frac{\hat{\xi}^{(i)}}{\hat{\xi}_{n+1}^{(i)}} \right\| = \frac{1}{|\hat{\mathbb{X}}_{n+1}^{(i)}|} \left\| \hat{\mathbb{X}}^{(i)} - \frac{\hat{\mathbb{X}}_{n+1}^{(i)}}{\hat{\xi}_{n+1}^{(i)}} \hat{\xi}^{(i)} \right\| \\
&\leqq \frac{1}{|\hat{\mathbb{X}}_{n+1}^{(i)}|} \left[\|\hat{\mathbb{X}}^{(i)} - \hat{\xi}^{(i)}\| + \frac{|\hat{\mathbb{X}}_{n+1}^{(i)} - \hat{\xi}_{n+1}^{(i)}|}{|\hat{\xi}_{n+1}^{(i)}|} \right] \\
&\leqq \frac{4}{3}(R+1)\left[\varepsilon' + \frac{\varepsilon'}{\frac{3}{4(R+1)} - \varepsilon'} \right] \\
&\leqq \frac{4}{3}(R+1)[\varepsilon' + 2(R+1)\varepsilon'] \quad \text{(by substituting } \varepsilon' = \varepsilon/4(R+1)^2 < 1/4(R+1) \text{ in the denominator)} \\
&< 4(R+1)^2\varepsilon' = \varepsilon.
\end{aligned}$$

Hence, (6.2) is proven.

7. Acknowledgment. The author gratefully acknowledges the detailed comments provided by an anonymous referee.

REFERENCES

[1] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[2] J. CANNY, *A new algebraic method for robot motion planning and real geometry*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 39-48.

[3] A. L. CHISTOV AND D. Y. GRIGOR'EV (1983), *Subexponential time solving systems of algebraic equations, I and II*, LOMI preprints E-9-83, E-10-83, Leningrad.

[4] ———, *Complexity of quantifier elimination in the theory of algebraically closed fields*, Lecture Notes in Computer Science 176, Springer-Verlag, 1984.

[5] D. Y. GRIGOR'EV AND N. N. VOROBJOV, *Solving systems of polynomial inequalities in subexponential time*, J. Symbolic Comput., 5 (1988), pp. 37-64.

[6] D. LAZARD, *Résolution des systèmes d'équations algébriques*, Theoret. Comput. Sci., 15 (1981), pp. 77-110.

[7] M. MARDEN, *Geometry of Polynomials*, American Mathematical Society, Providence, RI, 1966.

[8] F. S. MACAULAY, *Some formulae in elimination*, Proc. London Math. Soc. 1, 35 (1902), pp. 3-27.

[9] J. RENEGAR, *On the efficiency of Newton's method in approximating all zeros of a system of complex polynomials*, Math. Oper. Res., 12 (1987), pp. 121-148.

[10] ———, *On the worst-case arithmetic complexity of approximating zeros of polynomials*, J. Complexity, 3 (1987), pp. 90-113.

[11] B. L. VAN DER WAERDEN, *Modern Algebra*, Vol. 2, (English Translation), Frederick Ungar Publishing Co., New York, 1950.

SIAM J. COMPUT
Vol. 18, No. 2, pp. 371-384, April 1989

013

PARTITIONING SPACE FOR RANGE QUERIES*

F. FRANCES YAO†, DAVID P. DOBKIN‡, HERBERT EDELSBRUNNER§, AND MICHAEL S. PATERSON¶

Abstract. It is shown that, given a set S of n points in R^3, one can always find three planes that form an *eight-partition* of S, that is, a partition where at most $n/8$ points of S lie in each of the eight open regions. This theorem is used to define a data structure, called an *octant tree*, for representing any point set in R^3. An octant tree for n points occupies $O(n)$ space and can be constructed in polynomial time. With this data structure and its refinements, efficient solutions to various range query problems in two and three dimensions can be obtained, including (1) half-space queries: find all points of S that lie to one side of any given plane; (2) polyhedron queries: find all points that lie inside (outside) any given polyhedron; and (3) circle queries in R^2: for a planar set S, find all points that lie inside (outside) any given circle. The retrieval time for all these queries is $T(n) = O(n^\alpha + m)$, where $\alpha = 0.8988$ (or 0.8471 in case (3)), and m is the size of the output. This performance is the best currently known for linear-space data structures that can be deterministically constructed in polynomial time.

Key words. range query, half-space query, partition, equipartition, Borsuk-Ulam

AMS(MOS) subject classifications. primary 68U05; secondary 55M20

1. Introduction.

Consider a database that contains a collection of records with multidimensional keys. Given a *range query*, which is specified by certain constraints on the value of the multidimensional key, the database is expected to return the set of all records (or some function of the set of all records) whose keys satisfy those constraints. Efficient solutions to range queries are important both in themselves and also as subroutines for solving other multidimensional search problems. In this paper we will consider solutions to range queries that use only linear space for data structure storage.

There is an extensive literature on efficient algorithms for handling *orthogonal queries*, that is, queries with constraints of the form $a_1 \leqq k_1 \leqq b_1, \cdots, a_d \leqq k_d \leqq b_d$, where the key is $(k_1, \cdots, k_d)$. Relatively little is known about solving queries of more general types, such as *half-space queries*, where the constraints are linear inequalities $a_1k_1 + \cdots + a_dk_d \leqq c$. Willard [W] was the first to consider half-space queries for $d = 2$, and gave a solution with linear space and sublinear query time $O(n^\alpha)$, where $\alpha \approx 0.774$. Edelsbrunner and Welzl [EW] improved α to $\log_2(\sqrt{5}+1)/2 \approx 0.695$. Both of these results are based on the fact that a set of n points in R^2 can be partitioned by two lines so that each open quadrant contains at most $n/4$ points.

For $d = 3$, the first nontrivial time bound was $O(n^\alpha)$ for $\alpha \approx 0.98$ by Yao [Y]. The data structure is based on a partition of any point set by three planes into eight regions with the property that no seven regions together contain more than 23/24 of the points. Such a partition was obtained by making use of the concept of a *centerpoint* of a set (see [YB]).

In this paper, we prove a stronger result on partitions in R^3 by using the *Borsuk-Ulam theorem* of topology. It is shown that, given a set S of n points in R^3, one can

* Received by the editors May 15, 1988; accepted for publication July 18, 1988.

† Xerox Palo Alto Research Center, Palo Alto, California 94304.

‡ Department of Computer Science, Princeton University, Princeton, New Jersey 08544. The work of this author was supported by National Science Foundation grant CCR87-00917.

§ Department of Computer Science, University of Illinois, Urbana, Illinois 61801. The work of this author was supported by National Science Foundation grant CCR-8714565.

¶ Department of Computer Science, University of Warwick, Coventry, United Kingdom. The work of this author was supported by Xerox during his visits to the Palo Alto Research Center and by a Senior Fellowship of the Science and Engineering Research Council of the United Kingdom.

always find three planes that form an *eight-partition* of S, that is, a partition where at most $n/8$ points of S lie in each of the eight open regions. This theorem is used to define a data structure, called an *octant tree*, for representing any point set in R^3. Efficient solutions to various range query problems in R^2 and R^3 can be obtained by using this data structure and its refinements. For example, one can solve in time $O(n^{0.8988}+m)$, where m is the size of the output,

1) half-space queries: find all points that lie to one side of a query plane;
2) polyhedron queries: find all points that lie inside (outside) a query polyhedron; and
3) circle queries in R^2: given a planar set, find all points that lie inside (outside) a query circle.

An octant tree for n points occupies $O(n)$ space and can be constructed in $O(n^6 \log n)$ preprocessing time.

The paper contains five sections. In § 2 we define the concepts necessary for discussing space partitions in both the continuous case and the discrete case. Section 3 contains the proof of the main theorem of the paper. In § 4 we present and analyze several data structures based on the main theorem for representing point sets. Finally we comment on open problems and related results in § 5.

2. Preliminaries. We use S^{d-1} to denote the unit sphere $\{(x_1, x_2, \cdots, x_d) | x_1^2+x_2^2+\cdots+x_d^2=1\}$ in R^d. An *oriented hyperplane* (or *hyperplane* for short) h in R^d with *normal vector* $\mathbf{v}=(v_1, v_2, \cdots, v_d) \in S^{d-1}$ is defined by an equation $\sum v_i x_i = t$. If v has length 1, then the real number t is the *distance* of h from the origin; it is the unique scalar for which the point $t \cdot \mathbf{v}$ lies on h. The hyperplane h separates R^d into a *positive half-space* h^+ defined by $\sum v_i x_i > t$ and a *negative half-space* h^- defined by $\sum v_i x_i < t$. When we consider continuous functions defined on the collection of hyperplanes in R^d, we assume that the latter is endowed with the topology of $S^{d-1} \times R$ through the representation of h by $(\mathbf{v}, t)$. Corresponding to $h=(\mathbf{v}, t)$ we let $-h$ denote the hyperplane $(-\mathbf{v}, -t)$; thus, $-h$ is a hyperplane defined by the same equation as h but with the opposite orientation.

We shall limit our discussions to R^d with $d \leqq 3$ in this paper. A hyperplane in R^3 will simply be called a *plane*. For a set S of n points in R^3, we are interested in finding three planes h_1, h_2, h_3 so that at most $n/8$ points lie in each of the eight open regions defined by the three planes. Such a triple (h_1, h_2, h_3) is termed an *eight-partition* of S. We shall prove the existence of an eight-partition for any finite point set by first transforming the problem to a continuous framework. Thus, let A be a positive density function defined on some bounded, connected region in R^3, and let an *eight-partition* of A be a triple of planes (h_1, h_2, h_3) that partitions A into eight parts of equal mass.

LEMMA 2.1. *If every positive density function over a bounded connected region in R^3 has an eight-partition, then every finite point set in R^3 has an eight-partition.*

Proof. We replace a set S of n points with a density function A by placing at each point $p \in S$ a small ball $b(p)$ of uniform mass $(1-\delta)/n$ with radius ε and center p. We choose ε to be small enough so that a set of balls can intersect a common plane only if their centers are coplanar. Let C be a large sphere of volume $\|C\|$ which contains all the balls, and place additional density $\delta/\|C\|$ uniformly inside C. Thus the total mass over C is 1, and the mass outside of the union of the balls is less than δ, which is chosen to be less than $1/2n$. Suppose we find an eight-partition for A with planes (h_1, h_2, h_3). Then we can find $(\bar{h}_1, \bar{h}_2, \bar{h}_3)$ with the property that (1) $p \in \bar{h}_\ell$ if $b(p)$ intersects h_ℓ, and (2) $p \in \bar{h}_\ell^+ \cup \bar{h}_\ell$ (or $\bar{h}_\ell^- \cup \bar{h}_\ell$) if $b(p)$ lies in h_ℓ^+ (or h_ℓ^-). This is possible by the choice of ε. The partition $(\bar{h}_1, \bar{h}_2, \bar{h}_3)$ has the property that there are at most $n/8$ points in each of the eight open regions of the partition. □

Because of Lemma 2.1, it suffices to prove the existence of eight-partitions in the continuous setting. Let A be a density function as described in the lemma. Any triple of planes (h_1, h_2, h_3) partitions A into the eight proportions denoted by $a_{xyz}(h_1, h_2, h_3)$ for $x, y, z \in \{0, 1\}$, where the ℓth subscript is 0 or 1 depending on whether the region lies in h_ℓ^+ or h_ℓ^-. We shall abbreviate $a_{xyz}(h_1, h_2, h_3)$ as a_{xyz} whenever possible. Thus, a_{010} denotes the mass contained in $h_1^+ \cap h_2^- \cap h_3^+$, and a_{111} denotes the mass contained in $h_1^- \cap h_2^- \cap h_3^-$. (See Fig. 1, where $\mathbf{v}_i$ is the normal vector to h_i.) The a_{xyz}'s are continuous functions of h_1, h_2, and h_3. We use $*$ as a subscript to indicate a summation of the a_{xyz}'s where that subscript can assume both 0 and 1. For example, we write a_{xy*} for $\sum_z a_{xyz} = a_{xy0} + a_{xy1}$, and a_{*y*} for $\sum_{x,z} a_{xyz} = a_{0y0} + a_{0y1} + a_{1y0} + a_{1y1}$, etc.

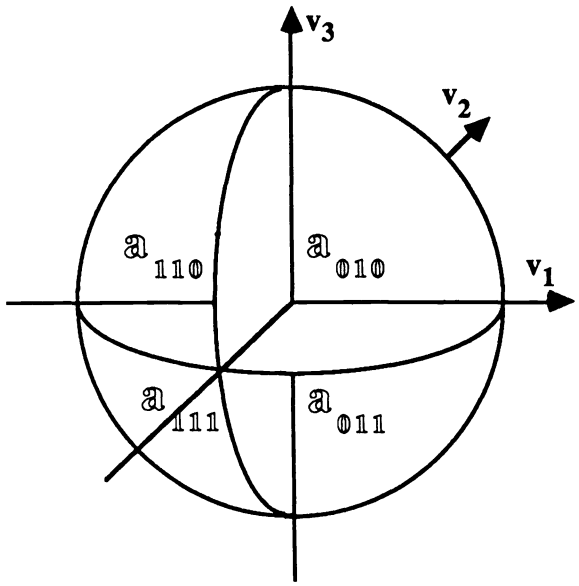


FIG. 1

DEFINITIONS. Let A be a positive density function on R^3, and consider the a_{xyz}'s defined by three planes (h_1, h_2, h_3). We say that

1) h_1 is a *bisector* of A if $a_{x**} = \frac{1}{2}$ for $x \in \{0, 1\}$;
2) the pair (h_1, h_2) forms a *four-partition* of A if $a_{xy*} = \frac{1}{4}$ for all $x, y \in \{0, 1\}$;
3) the triple (h_1, h_2, h_3) forms an *eight-partition* of A if $a_{xyz} = \frac{1}{8}$ for all $x, y, z \in \{0, 1\}$.

In order to achieve $a_{xyz} = \frac{1}{8}$ for all x, y, z, it is convenient to form eight linear combinations of the a_{xyz}'s as follows. Let $f_{ijk} = \sum_{xyz} \varepsilon_{ijk}^{xyz} a_{xyz}$, where $\varepsilon_{ijk}^{xyz} = (-1)^b$ with $b = (i, j, k) \cdot (x, y, z) = ix + jy + kz$. For example, $f_{000} = a_{***} \equiv 1$, $f_{100} = a_{0**} - a_{1**}$, and

$$f_{110} = a_{00*} - a_{01*} - a_{10*} + a_{11*}.$$

The f_{ijk}'s, like the a_{xyz}'s, are continuous functions of h_1, h_2, and h_3. Note that f_{110} is symmetric in the first and the second arguments: $f_{110}(h_1, h_2, h_3) = f_{110}(h_2, h_1, h_3)$. Also, when we flip the sign of an argument h_ℓ, the function f_{xyz} either changes sign or not depending on whether the corresponding subscript in f_{xyz} is 1 or 0. For example, for $\ell = 1$, $f_{110}(-h_1, h_2, h_3) = -f_{110}(h_1, h_2, h_3)$; while for $\ell = 3$, $f_{110}(h_1, h_2, -h_3) = f_{110}(h_1, h_2, h_3)$. We state these symmetry properties in the next lemma.

LEMMA 2.2.

1) $f_{ijk}(h_1, h_2, h_3) = f_{jik}(h_2, h_1, h_3)$.

2) $f_{ijk}(-h_1, h_2, h_3) = (-1)^i f_{ijk}(h_1, h_2, h_3)$.

Proof. Immediate from the definition of f_{ijk}. □

In seeking an eight-partition for A, we shall make use of the following characterization in terms of the f_{ijk}'s.

LEMMA 2.3.

1) *h_1 is a bisector of A if and only if $f_{100} = 0$.*

2) *the pair (h_1, h_2) forms a four-partition of A if and only if $f_{100} = f_{010} = f_{110} = 0$.*

3) *The triple (h_1, h_2, h_3) forms an eight-partition of A if and only if $f_{ijk} = 0$ for all $(i, j, k) \neq (0, 0, 0)$.*

Proof. 1) $a_{0**} = a_{1**} = \frac{1}{2}$ if and only if $a_{0**} - a_{1**} = f_{100} = 0$ since $a_{0**} + a_{1**} = f_{000} \equiv 1$.

2) Note that

$$\begin{pmatrix} f_{100} \\ f_{010} \\ f_{110} \\ f_{000} \end{pmatrix} = \begin{pmatrix} 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_{00*} \\ a_{01*} \\ a_{10*} \\ a_{11*} \end{pmatrix}$$

If $a_{00*} = a_{01*} = a_{10*} = a_{11*} = \frac{1}{4}$, then $f_{100} = f_{010} = f_{110} = 0$ and $f_{000} \equiv 1$. Since the 4×4 matrix is nonsingular, indeed orthogonal, the converse is also true.

3) We show that the matrix of the coefficients $\{\varepsilon_{ijk}^{xyz}\}$ is orthogonal, and so nonsingular. Since

$$\sum_{xyz} \varepsilon_{ijk}^{xyz} \varepsilon_{i'j'k'}^{xyz} = \sum_{xyz} (-1)^{(x,y,z)(i+i', j+j', k+k')}$$

$$= 0 \text{ unless } i = i',\ j = j',\ \text{and } k = k',$$

the inner product of any two distinct rows of the matrix is zero. □

3. Eight-partition in R^3. It is well known that a four-partition can always be found for a positive density function over a bounded connected region in R^2. (See, e.g., [Me].) We first show this as a lemma and then prove a slightly stronger version in R^3 for later use.

LEMMA 3.1 (Four-Partition). *Let A_0 and A_1 be two positive density functions on the plane whose domains are bounded, connected, and separable by a line L. There is a unique (unoriented) line L' that bisects A_0 and A_1 simultaneously.*

Proof. For any point p on L, let ℓ_p and r_p be the (unique) lines that go through p and bisect A_0 and A_1, respectively. We can assume that L is vertical and that A_1 is to the right of L. As p moves up L from bottom to top, the slope of r_p decreases continuously and monotonically from ∞ to $-\infty$, while that of ℓ_p increases continuously and monotonically from $-\infty$ to ∞. Hence there is a unique p for which ℓ_p and r_p coincide, giving the desired L'. □

We next consider four-partitions in R^3.

LEMMA 3.2. *Let A_0 and A_1 be two positive density functions in R^3 whose domains are bounded, connected, and separable by a plane h. Let $S_h \cong S^1$ denote the set of unit vectors in R^3 that are parallel to h. Then,*

1) *for any $u \in S_h$ there is a unique plane $p(u)$ parallel to u which bisects A_0 and A_1 simultaneously and has an orientation induced by u;*

2) *the mapping $f: S_h \mapsto S^2$ which maps $u \in S_h$ to the normal vector of $p(u)$ gives a continuous antipodal mapping of S_h into S^2, i.e., $f(-u) = -f(u)$ for all $u \in S_h$.*

Proof. For any $u \in S_h$, a plane $p(u)$ parallel to u bisects A_0 and A_1 simultaneously if and only if the projection of $p(u)$ along the direction u (onto a plane normal to u) gives a line that bisects the projections of A_0 and A_1 simultaneously. The intersection of h and $p(u)$ is a line parallel to u. The orientation of u gives it a natural orientation which can be used to define the left and right half-planes of h with respect to this line. The orientation of $p(u)$ is chosen so that these lie respectively in the positive and negative half-spaces determined by $p(u)$. By Lemma 3.1, $p(u)$ is unique.

It is easy to see that the function f defined in (2) is continuous and antipodal. □

We shall make use of the $d=2$ case of the following topological theorem in proving that every density function A in R^3 can be eight-partitioned.

THEOREM (Bursuk-Ulam). *Let $f: S^d \mapsto R^d$ be a continuous, antipodal map, i.e., $f(-p) = -f(p)$ for $p \in S^d$. Then there is a point $p \in S^d$ such that $f(p) = 0$.*

A proof of the Borsuk-Ulam theorem can be found in textbooks on algebraic topology such as Munkres [Mu]. The theorem does not extend in general to mappings defined on direct products of spheres. However, we can establish the following extension for mappings defined on the torus $S^1 \times S^1$ which satisfy certain additional symmetry properties. This lemma is sufficient, as we shall see, for establishing the existence of eight-partitions in R^3.

LEMMA 3.3. *Let $f: S^1 \times S^1 \mapsto R^2$ be a continuous map such that*

1) *f is symmetric,* i.e., $f(u, v) = f(v, u)$,

2) *f is antipodal in each argument,* i.e., $f(u, -v) = f(-u, v) = -f(u, v)$, *and*

3) *f is constant on the diagonal $\{(u, u) | u \in S^1\}$.*

Then there is a point $p \in S^1 \times S^1$ such that $f(p) = 0$.

Proof. We can represent $S^1 \times S^1$ by the square $\mathcal{Q} = [0, 2\pi] \times [0, 2\pi]$ with opposite sides identified. Consider the rectangle $\mathcal{P} = \{(u, v) | \pi \leqq u + v \leqq 3\pi,\ u \leqq v \leqq u + \pi\}$ contained in $\mathcal{Q}$ with vertices $A = (\pi/2, \pi/2)$, $B = (0, \pi)$, $C = (3\pi/2, 3\pi/2)$, and $D = (\pi, 2\pi)$ (Fig. 2). Note that f is constant on side AC by property (3), constant on side BD since $f(u, u+\pi) = -f(u, u)$ by (2), and defined identically on AB and CD since $f(u, v) = f(u+\pi, v+\pi)$. The involution $(u, v) \leftrightarrow (v, u+\pi)$ maps square $ABFE$ to $FECD$ and vice versa, while $f(v, u+\pi) = -f(v, u) = -f(u, v)$. Consider any map $\alpha: \mathcal{P} \mapsto S^2$ which identifies sides AB and CD, contracts BD and AC to points, and maps every pair of points of the form (u, v) and $(v, u+\pi)$ in $\mathcal{P}$ to a pair of antipodal

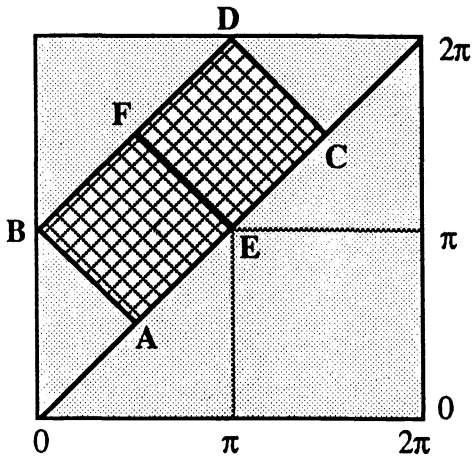


FIG. 2.

points $(p, -p)$ on S^2. Such an α yields an induced continuous function f': $S^2 \mapsto R^2$ where $f'(\alpha(p)) = f(p)$ for $p \in \mathcal{P}$. Since f' is an antipodal map, it follows from the Borsuk–Ulam theorem that f', and hence f, must map some point onto the origin of R^2. □

MAIN THEOREM. *Let A be a positive density function over a bounded connected region in R^3, and let $w_0 \in S^2$ be given. Then there exists a triple of planes (h_1, h_2, h_3) which forms an eight-partition for A, and where the normal vector of h_3 is w_0.*

Proof. By Lemma 2.3, this is equivalent to finding h_1, h_2, h_3 such that $f_{ijk}(h_1, h_2, h_3) = 0$ for all $(i, j, k) \neq (0, 0, 0)$. Since h_1, h_2, and h_3 must be bisectors for A, and for any $u \in S^2$ there is a unique bisector h_u for A with normal vector u, we can define functions g_{ijk}: $S^2 \times S^2 \times S^2 \mapsto R$ by

$$g_{ijk}(u, v, w) = f_{ijk}(h_u, h_v, h_w). \tag{1}$$

the g_{ijk}'s are obviously continuous. It suffices to find u_0 and v_0 such that $g_{110} = g_{101} = g_{011} = g_{111} = 0$ at (u_0, v_0, w_0). We can limit the choice of u_0 (or v_0) to the set $\{u\}$ (or $\{v\}$) for which (h_u, h_{w_0}) (respectively, (h_v, h_{w_0})) forms a four-partition, i.e.,

$$g_{101}(u, v, w_0) = 0 \quad \text{and} \quad g_{011}(u, v, w_0) = 0. \tag{2}$$

By Lemma 3.2, the set of (u, v) that satisfies (2) is homeomorphic to $S^1 \times S^1$. Our goal is thus to find a point $(u_0, v_0) \in S^1 \times S^1$ where both of the functions

$$G_0(u, v) \overset{\text{def}}{=} g_{110}(u, v, w_0) \text{ and } G_1(u, v) \overset{\text{def}}{=} g_{111}(u, v, w_0)$$

are zero. Let $G \overset{\text{def}}{=} (G_0, G_1)$: $S^1 \times S^1 \mapsto R^2$. Note that G is symmetric in its two arguments and antipodal on each S^1 by Lemma 2.2 and (1):

$$G(u, v) = G(v, u),$$

$$G(u, -v) = -G(u, v).$$

Furthermore it is easy to verify that on the diagonal we have

$$G(u, u) = (1, 0).$$

It follows from Lemma 3.3 that there exists $(u_0, v_0) \in S^1 \times S^1$ such that $G(u_0, v_0) = (0, 0)$. We conclude that $(h_{u_0}, h_{v_0}, h_{w_0})$ yields the desired eight-partition. □

LEMMA 3.4. *Consider a partition of R^3 into eight open regions by three mutually intersecting planes. Any plane h in R^3 can intersect at most seven of these eight regions.*

Proof. Define the origin O to be the point where the three planes meet, and axes X, Y, and Z to be the lines where pairs of planes intersect. Then O divides each axis into two half-axes $\{X^+, X^-\}$, $\{Y^+, Y^-\}$, and $\{Z^+, Z^-\}$. Without loss of generality, assume that h intersects the half-axes X^+, Y^+, and Z^+. Then h does not intersect the open region bounded by X^-, Y^-, and Z^-. □

The Borsuk–Ulam theorem also leads to the following well-known corollary (see, e.g., [E]), which we shall employ in defining a data structure in § 4.2. We state it in the discrete version for convenience.

THEOREM (Ham-Sandwich Cut). *Let $S_1, S_2, \cdots, S_d$ be d finite point sets in R^d. There exists a hyperplane which simultaneously bisects $S_1, S_2, \cdots, S_d$.*

4. Data structures and algorithms. Let S be a finite set of points in R^3. We will describe several tree structures for representing S, based on partitions of S by planes. These partitions are obtained by recursive applications of the theorems proved in the last section, and the resulting tree structures are suitable for the purpose of half-space retrieval and related searches on S.

We first present a recursive partition scheme based on the main theorem, giving a data structure termed an *octant tree.* Two variations of this basic scheme are derived by applying recursion in more intricate ways. The retrieval time is analyzed for each scheme, with the last variant achieving a retrieval time of $(n^{0.8988})$ for half-space query.

4.1. Octant tree. An octant tree is a recursively defined structure for storing a finite set S of points in R^3. If S is empty the octant tree is the special node NIL, otherwise the root of the octant tree contains a plane h, and its left, middle, and right children represent the subsets $S \cap h^-$, $S \cap h$, and $S \cap h^+$, respectively. More precisely, the middle child points to a two-dimensional data structure for $S \cap h$ (such as a polygon tree [W] or a conjugation tree [EW]), while the left and right children point to the root nodes of octant trees for $S \cap h^-$ and $S \cap h^+$, respectively. We define the *domain* of any node v, denoted by dom (v), to be the intersection of the "regions" on the path from the root to v. That is, the domain of the root is the whole space, and if v is a child of w, and h is the plane stored at w, then dom (v) is the intersection of dom (w) with h^-, h, or h^+, depending on whether v is the left, middle, or right child of w. The *set stored at* v, denoted by $S(v)$, is $S \cap$ dom (v).

The plane h at each node v of the octant tree is chosen so that both $|S(v) \cap h^-|$ and $|S(v) \cap h^+|$ are at most $|S(v)|/2$. There is no difficulty in finding planes with this property, but, without further conditions, such a data structure would have poor worst-case performance for half-space retrieval. (For example, k-d trees [B] require $O(n)$ time in the worst case.) In the schemes to be described, we achieve better performance by appropriately grouping the tree nodes so that all nodes within a group can share a common plane. For this purpose, we build octant trees recursively from small *primitive trees* and define the grouping among the nodes of a primitive tree. The data structures pointed to by middle children represent the corresponding two-dimensional sets efficiently for a retrieval time of $O(t^\delta)$ for a set of size t, where $\delta \approx 0.695$ [EW]. These substructures are ignored in the recursive structures described below and contribute just $O(n^\delta)$ terms to the recurrence relations given.

OCTANT TREE A. In this basic scheme, the primitive tree is of height 3 and all nodes on the same level share the same plane (see Fig. 3). The existence of such primitive trees is implied by the Main Theorem. The octant tree is obtained by applying recursion at the leaves of the primitive tree. In the figure solid nodes represent the roots of primitive trees.

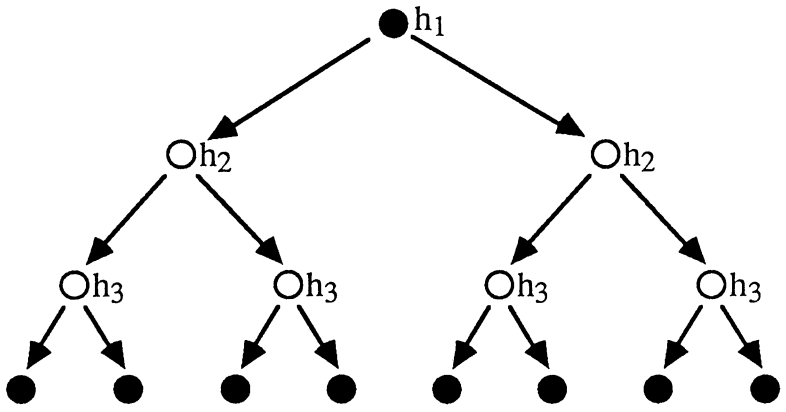


FIG. 3.

A search algorithm for the octant tree can be derived recursively from a search algorithm for the corresponding primitive tree. The general search strategy for primitive trees, with respect to a query plane q, is to identify those leaves in the primitive tree which need not be searched further. More precisely, a leaf v can be excluded from future search if dom (v) lies completely in a half-space of q, since the entire set $S(v)$ can be either reported or discarded. A leaf satisfying this condition is said to be *free* with respect to q.

By Lemma 3.4, the primitive tree has at least one free leaf with respect to any query q. Furthermore, the time required to identify the free leaves is bounded by a constant. Thus the search time for the derived octant tree is proportional to the total number of nodes visited. The recurrences below yield upper bounds for the search time. The reporting time, which is always linear in the size of the result, is not included here. Let $f(n)$ denote the maximum number of nodes visited in an octant tree for a set of n points. The constant δ arising from the two-dimensional subtrees has value at most 0.695.

LEMMA 4.1. *$f(n)$ satisfies the recurrence relation*

$$f(n) \leqq 7 + 7f(n/8) + O(n^{\delta}),$$

which gives a search time of $O(f(n)) = O(n^{\alpha})$ for $\alpha = \log_8 7 \approx 0.9358$.

Proof. In Fig. 3, the seven upper nodes of the primitive tree are visited, followed by visits to the seven substructures of size at most $n/8$ corresponding to the non-free leaves. The total number of nodes visited from middle children is at most $O(n^{\delta})$. The linear recurrence relation is solved by standard techniques (see, e.g., Knuth [K]). □

4.2. Refined octant trees.

OCTANT TREE B. In this variant of the basic scheme, we apply recursion to the four sets at level 2, while requiring that the same h_3 be used as the first plane for all four sets. The primitive tree is of depth 2 (see Fig. 4). Here we take advantage of the strength of the Main Theorem, which allows one of the three partitioning planes to be an arbitrary bisector.

To estimate the search time for scheme B, let $f(n)$ (and $g(n)$) denote the maximum number of nodes searched for any query, when the search starts at a root-level (and, respectively, second-level) node v of the primitive tree with $S(v) = n$.

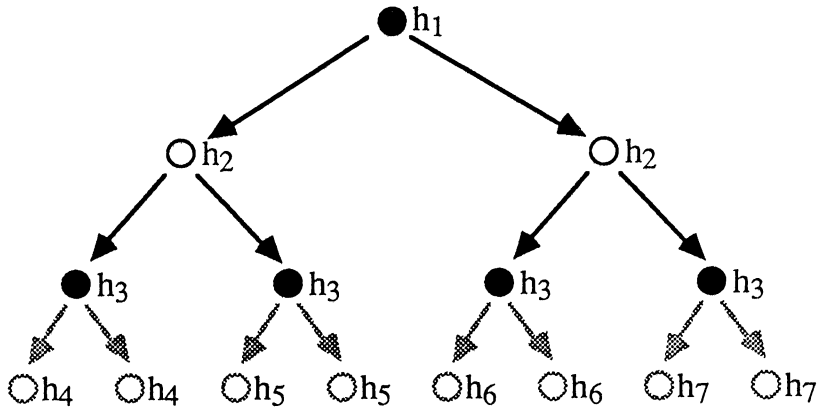


FIG. 4.

LEMMA 4.2. *$f(n)$ and $g(n)$ satisfy the recurrence relations*

$$f(n) \leqq 4+3f(n/4)+g(n/8)+O(n^{\delta}),$$

$$g(n) \leqq 1+2f(n/2)+O(n^{\delta}),$$

which give a search time under scheme B *of $O(f(n)) = O(n^{\beta})$, where $\beta \approx 0.9163$.*

Proof. Suppose without loss of generality that the rightmost of the eight lowest nodes, r, is free. The algorithm visits the three upper nodes and the parent of r, then recurses from the root level in the three left subtrees. In the fourth subtree, since r is free the search can begin at the second-level node which is r's sibling. This yields the first inequality. For the second, a search begun at second-level node searches that node and recurses on its children, which are root-level nodes. Substituting the second inequality into the first, we have

$$f(n) \leqq 5+3f(n/4)+2f(n/16)+O(n^{\delta}).$$

The recurrence yields $f(n) = O(n^{\beta})$, where $\beta \approx 0.9163$. □

OCTANT TREE C. This is a hybrid of schemes A and B with some further refinements. The primitive tree has six leaves on level 3 and one leaf on level 2. (See Fig. 5.) The six leaves on level 3 are divided into two triplets, where each triplet is to share a common first plane in the recursion. This is possible since, by the Ham-Sandwich theorem, any three point sets in R^3 can be bisected by a single plane. We choose each triplet to consist of three octants that do not share any common faces; indeed the six octants can be divided into two such triplets as shown in Fig. 6, where the octants are represented as the vertices of a cube.

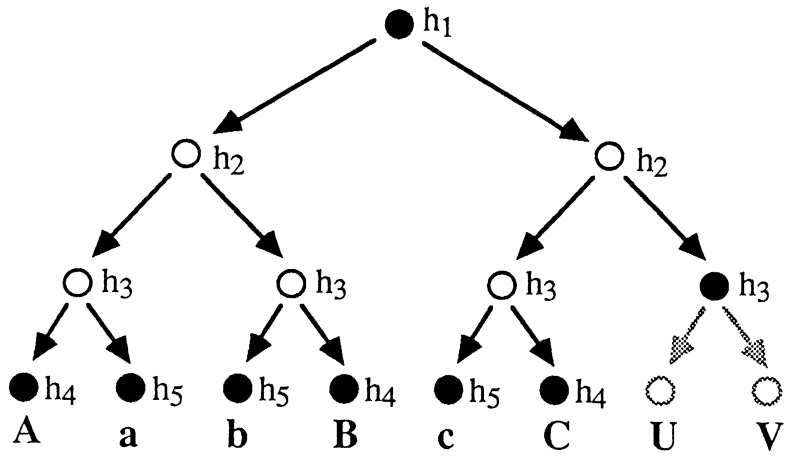


FIG. 5.

Again, define $f(n)$ and $g(n)$ as in the analysis of scheme B. We have the following bound for $f(n)$.

LEMMA 4.3. *Under scheme C, $f(n)$ satisfies the recurrence relation*

$$f(n) \leqq 10+f(n/4)+4f(n/8)+4f(n/64)+O(n^{\delta}),$$

which gives a search time of $O(f(n)) = O(n^{\gamma})$ for $\gamma \approx 0.8988$.

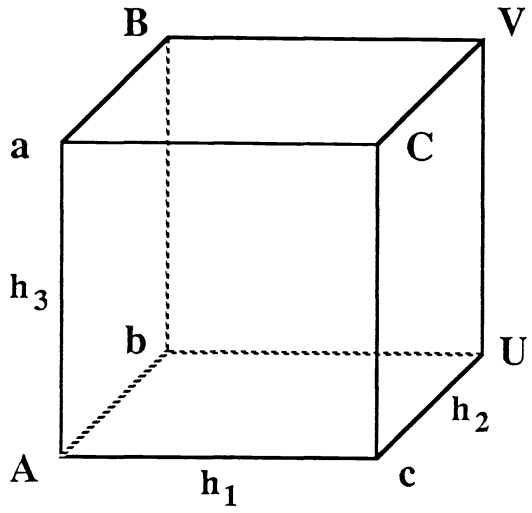


FIG. 6.

Proof. There are a number of cases to consider depending on which of the eight nodes is free.

If A is free then the intersection of the query plane q with the other domains will be similar to Fig. 7. The domains A, B, and C are bisected by h_4, while h_5 bisects a, b, and c. In the worst case, q will intersect both halves of B and C but, by our choice of the triplets, q cannot intersect both halves of a, b, and c. (In Fig. 7, the intersections of q with a, b, and c form three regions which cannot be simultaneously intersected by the line representing the intersection of q and h_5.) For the case detailed in Fig. 7, our algorithm is to search the six non-leaf nodes of the primitive tree; recursively

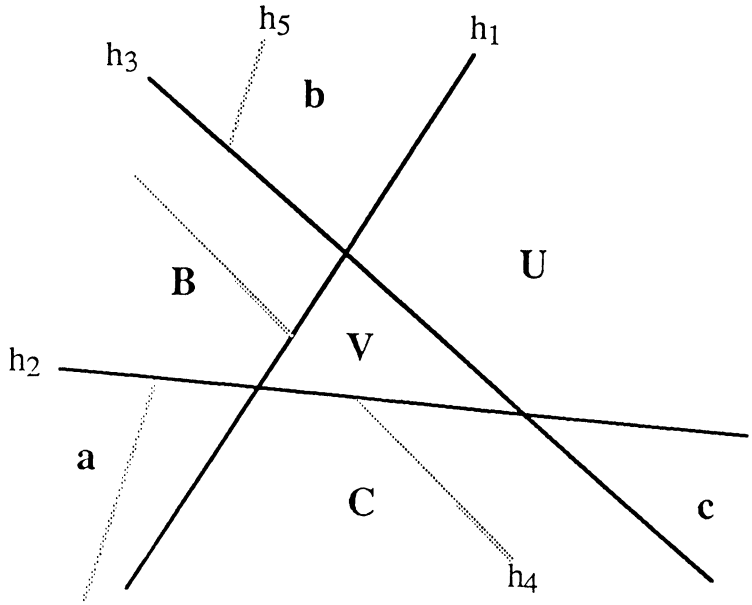


FIG. 7.

search from the parent of U and V; recursively search a, b, B, and C; search the node labelled c; and search from the second-level node corresponding to the child of c which is intersected by q. This yields the inequality

$$f(n) \leqq 7 + f(n/4) + 4f(n/8) + g(n/16) + O(n^{\delta}). \tag{3}$$

If U is free the intersection with q is similar to Fig. 8. Here at most two of A, B, and C and two of a, b, and c can have both of their halves (with regard to h_4 and h_5, respectively) intersected by q. For the case detailed in Fig. 8, our algorithm is to search the upper seven nodes; search the nodes labelled B and b; recursively search a, c, A, and C; recursively search the single children of B and b which are intersected by q; and recursively search V. The corresponding inequality is

$$f(n) \leqq 9 + 4f(n/8) + 2g(n/16) + g(n/8) + O(n^{\delta}). \tag{4}$$

We also have two inequalities for $g(n)$, depending on whether the second-level node where the search starts is the left child or the right child of a root node.

$$g(n) \leqq 3 + 4f(n/4) + O(n^{\delta}), \tag{5}$$

$$g(n) \leqq 2 + 2f(n/4) + f(n/2) + O(n^{\delta}). \tag{6}$$

The worst case is obtained by substituting (5) into (3), resulting in

$$f(n) \leqq 10 + f(n/4) + 4f(n/8) + 4f(n/64) + O(n^{\delta}).$$

This yields $f(n) = O(n^{\gamma})$, where $\gamma \approx 0.8988$. □

4.3. Preprocessing cost. We look at the time it takes to construct an octant tree for a set S of n points. First consider the computation of an eight-partition (h_1, h_2, h_3) for S. The first bisector h_1 can be found in $O(n)$ time by a median-finding algorithm.

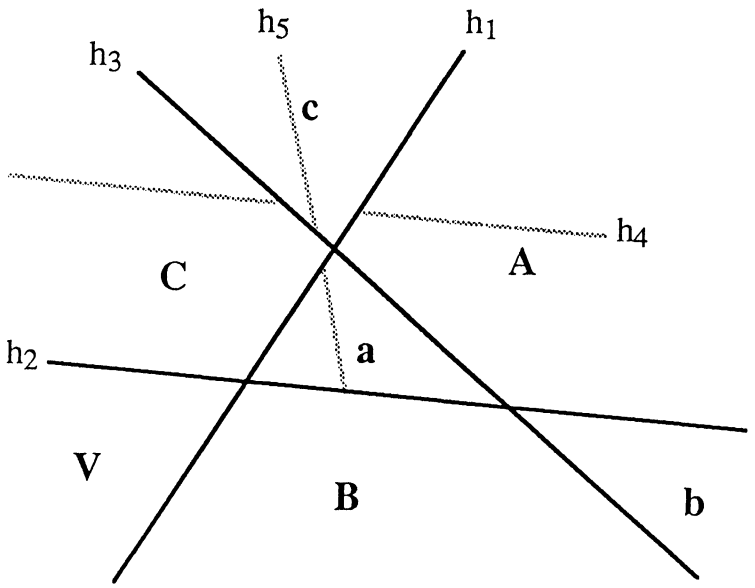


FIG. 8.

We may assume that h_2 and h_3 are each determined by three points of S. For each of the $O(n^6)$ possible choices of (h_2, h_3), we decide in $O(n)$ time whether it forms an eight-partition together with h_1 by explicitly counting the number of points in each octant. This amounts to a total time of $O(n^7)$ with linear space. The time can be reduced to $O(n^6)$ by lowering the cost due to counting as follows. For any two fixed points a, b, of S, order the remaining points as $c_1, c_2, \cdots$ by projecting the points of S onto a plane perpendicular to $\overline{ab}$ and sorting them radially. Coplanar sets of more than three points introduce some complication but no significant difficulty. With such orderings imposed on h_3 during the search, the task of counting associated with each pair (h_2, h_3) becomes that of doing simple updates, with constant cost per pair on average. The total cost for finding an eight-partition, with sorting included, is thus $O(n^6)$ time using $O(n^3)$ storage. The storage could be made $O(n)$ by repeating the sorting operations whenever needed, but at a cost of $O(n^6 \log n)$ time.

The octant trees of schemes A and B can be constructed by applying the above procedure recursively, in total time $O(n^6 \log^n)$ for n points with linear space. The same bounds hold for scheme C since a "ham-sandwich cut" can be computed in $O(n^3 \log n)$ time.

4.4. Circle queries. The problem of finding all points (x, y) in a planar set S which lie inside a query circle C with center (a, b) and radius r can be transformed to a three-dimensional half-space problem in the following way. Since

$$(x, y) \text{ lies inside } C \Leftrightarrow (x-a)^2+(y-b)^2 < r^2$$

$$\Leftrightarrow -2ax-2by+(x^2+y^2) < r^2-a^2-b^2,$$

if we represent each point $(x, y) \in S$ as a three-vector $\mathbf{v} = (x, y, x^2+y^2)$ then the query with respect to circle C can be expressed by the half-space query:

$$(-2a, -2b, 1) \cdot \mathbf{v} < (r^2-a^2-b^2).$$

A geometrical interpretation of this is that, when the xy-plane is projected upwards onto the paraboloid $z = x^2+y^2$, the image of any circle in the plane is the intersection of the paraboloid with a suitable plane. The same technique is applicable to other "algebraic" queries, but the dimension required is the number of degrees of freedom of the defining polynomial.

Rather than transform the circle query problem to three dimensions, we can also recast our three-dimensional results in two dimensions. An eight-partition of the n points on the paraboloid can be projected down to the xy-plane yielding a partition by three circles. Our main theorem implies the following.

COROLLARY 4.4. *For any finite point set and any bisecting circle in the plane, there are two circles such that each open region defined by the three circles contains at most one eighth of the set.*

Any query circle intersects at most six of the eight regions since it meets the three circles in at most six points. Using (the two-dimensional projection of) octant tree A we therefore get $O(f(n))$ query time, where

$$f(n) \leqq 7+6f(n/8)+O(\log n) = O(n^\alpha),$$

for $\alpha = \log_8 6 \approx 0.8617$. Here the $O(\log n)$ term takes care of the one-dimensional queries needed for points lying on the partitioning circles.

We get the same worst-case time using scheme C, but using octant tree B yields a slight improvement. The query time is $O(f(n))$, where

$$f(n) \leqq 7 + 2f(n/4) + 4f(n/16) + O(\log n),$$

which solves to $f(n) = O(n^{\beta})$, with $\beta = (\log_2 \sqrt{5} + 1)/2 \approx 0.8471$.

4.5. Polyhedron queries. Based on our half-space query schemes, we may derive a generalization to queries for convex polyhedra defined by the intersection of r hyperplanes. For fixed r, the query time will be of the same order as for half-space queries, but the constant increases with r. Since every (not necessarily convex) polyhedron can be decomposed into (possibly unbounded) tetrahedra (see, e.g., [Ch1]), it would suffice to consider at most tetrahedral queries, i.e., $r \leqq 4$.

Let $f_r(n)$ be the number of nodes of the data structure for n points which may be searched in a query with a polyhedron C which is the intersection of r half-spaces, where we already have by our scheme C above that $f_1(n) = O(n^{\gamma})$ for some $\gamma \leqq 0.8988$. We prove by induction on r that $f_r(n) = O(n^{\gamma})$ for all r.

Suppose $f_{r-1}(n) = O(n^{\gamma})$ and consider a query with respect to the intersection of r half-spaces. In the case that some level-three node is free with respect to all r planes, the recurrence relations considered above hold for f_r. In the alternative case, there are two level-three nodes, each free with respect to some plane or planes. Now the recurrence terms in f_r are diminished and so correspond to some exponent $\gamma' < \gamma$, while the remaining terms are of size $O(f_{r-1}(n)) = O(n^{\gamma})$. The result is that $f_r(n) = O(n^{\gamma})$ and the induction is complete.

Of course the same general argument is valid for any similar scheme in any finite dimension.

5. Conclusion and related results. We showed that an eight-partition with three planes exists for any finite point set in R^3. It was brought to the authors' attention that a continuous version of this theorem was proved earlier by Hadwiger with a more complicated argument [H]. As far as generalization to higher dimensions is concerned, Avis [A] showed that 2^d-partitions are not always possible in dimensions $d \geqq 5$. A different and simpler proof is as follows (stated here for $d = 5$ but adaptable to any larger d). Take thirteen small balls of equal mass and place them in general position so that no hyperplane in R^5 can intersect more than five of the balls. Now, in any 2^5-partition, each ball must be cut by at least two hyperplanes, otherwise some orthant will contain at least one half of a ball, with at least $1/26$ of the total mass, which is larger than the $1/32$ required. Therefore, for all thirteen balls, at least 26 instances of hyperplane-ball intersections are needed. Since the balls are in general position, five hyperplanes can provide at most 25 such instances and we have a contradiction.

The case of $d = 4$ still remains an intriguing open question, that is, given any finite point set in R^4, whether one can always partition it with four hyperplanes such that each orthant contains at most $1/16$ of the points. Our proof for $d = 3$ makes use of the Borsuk–Ulam theorem of algebraic topology; the case of $d = 4$ is likely to draw further upon classical mathematics for its resolution.

Generalizations of eight-partition to higher dimensions have also been studied along a different line, by relaxing the number of hyperplanes used in the partition (see Cole [Co] and Yao and Yao [YY]). Deterministic partition schemes in three dimensions that are different from those described in this paper can be found in [EH]. Their main construction is based on the existence of a six-partition for every planar point set, that

is, a partition by three concurrent lines so that every wedge contains at most one sixth of the points. The best such scheme achieves $O(n^{\alpha})$ query time, where $\alpha \approx 0.9089$.

Haussler and Welzl [HW] used random sampling to demonstrate the existence of partitions in R^d which afford the best query time currently known. In particular, for $d = 3$ their scheme gives query time $O(n^{\alpha})$ for $\alpha \approx 0.857$. As their algorithms are probabilistic, it is an interesting open question to find deterministic algorithms for constructing partitions to realize similar or even better query time in R^d.

Chazelle [Ch2] established a lower bound under a rather general model for range search in d dimensions. His bound in three dimensions assuming $O(n)$ space is $\Omega(n^{2/3} \log n)$ query time.

REFERENCES

[A] D. AVIS, *Non-partitionable point sets*, Inform. Process. Lett., 19 (1984), pp. 125-129.

[B] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching*, CACM, 18 (1975), pp. 509-516.

[Ch1] B. CHAZELLE, *Convex partitions of polyhedra: a lower bound and worst-case algorithm*, SIAM J. Comput., 13 (1984), pp. 488-507.

[Ch2] ———, *Polytope range searching and integral geometry*, Proc. 28th Annual IEEE Symposium on Foundations of Computer Science (1987), pp. 1-10.

[Co] R. COLE, *Partitioning point sets in arbitrary dimensions*, Theoret. Comput. Sci. 49 (1987), pp. 239-266.

[DE] D. P. DOBKIN AND H. EDELSBRUNNER, *Space searching for intersecting objects*, J. Algorithms, 8 (1987), pp. 348-361.

[E] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, Berlin, 1987.

[EH] H. EDELSBRUNNER AND F. HUBER, *Dissecting sets of points in two and three dimensions*, Report F 138, Inst. Informationsverarb., Technische Universität Graz, Austria, 1984.

[EW] H. EDELSBRUNNER AND E. WELZL, *Halfplanar range search in linear space and $O(n^{9.695})$ query time*, Inform. Process Lett., 23 (1986) pp. 289-293.

[H] H. HADWIGER, *Simultane Vierteilung zweier Körper*, Arch. Math. (Basel), 17 (1966), pp. 274-278.

[HW] D. HAUSSLER AND E. WELZL, *ε-nets and simplex range queries*, *Discrete Comput.* Geom., 2 (1987), pp. 127-151.

[K] D. E. KNUTH, *Fundamental Algorithms*: *The Art of Computer Programming* I, Addison-Wesley, Reading, MA, 1968.

[Me] N. MEGIDDO, *Partitioning with two lines in the plane*, J. Algorithms, 3 (1985), pp. 430-433.

[Mu] J. R. MUNKRES, *Elements of Algebraic Topology*, Addison-Wesley, Reading, MA, 1984.

[W] D. E. WILLARD, *Polygon retrieval*, SIAM J. Comput., 11 (1982), pp. 149-165.

[YB] I. M. YAGLOM AND V. G. BOLYANSKI, *Convex Figures* (English translation), Holt, Rinehart, and Winston, New York, 1961.

[Y] F. F. YAO, *A 3-space partition and its applications*, Proc. 15th Annual ACM Symposium on Theory of Computing, (1983), pp. 258-263.

[YY] A. C. YAO AND F. F. YAO, *A general approach to d-dimensional geometric queries*, Proc. 17th Annual ACM Symposium on Theory of Computing (1985), pp. 163-168.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 385-391, April 1989

014

CNF SATISFIABILITY TEST BY COUNTING AND POLYNOMIAL AVERAGE TIME*

KAZUO IWAMA†

Abstract. The average-case performance of an algorithm for CNF SAT, recently introduced by the author, is discussed. It is shown that the algorithm takes polynomial average time for a class of CNF equations satisfying the condition that for a constant c, $p^2 v \geqq \ln t - c$, where v is the number of variables, t is the number of clauses, and p is the probability that a given literal appears in a clause. It was known that backtracking plus the pure literal rule, a common way of solving CNF SAT, takes polynomial average time if $p \geqq \varepsilon$ (any small constant) or $p \leqq c\,(\ln v/v)^{3/2}$, but no algorithms were known to take polynomial average time (for all t) in the range $c\,(\ln v/v)^{3/2} < p < \varepsilon$. For reasonable t ($t \leqq v^{\alpha}$ for some constant $\alpha > 0$) the new algorithm runs in polynomial average time for $p > (\alpha \ln v/v)^{1/2}$, so the unfavorable region is reduced to $c(\ln v/v)^{3/2} < p < (\alpha \ln v/v)^{1/2}$.

Key words. CNF SAT, average time, independent sets, backtracking

AMS(MOS) subject classifications. 68Q20, 68Q25, 68R05

1. Introduction. A number of approaches to attack hard (typically NP-hard) combinatorial problems are known, most of which are claimed to be "realistic" in the sense that they work well in most average cases. However, if, as is widely believed, there are no good algorithms which work efficiently for *all* instances, those approaches should have some weak points. This seems to be true. For example, when trying to solve CNF Boolean equations (or their satisfiability, SAT) by backtracking [3], [4], [5], [9], [10], [11], [12], [13], it is known that the number of literals in each clause plays an important role. The more literals, the less efficiently backtracking works, which is clearly due to the fundamental structure of the approach. Probabilistic approaches to the Hamiltonian circuit problem [2], [7] also depend on the average degree of given graphs. (For example, [7] shows that there is a polynomial-time algorithm which almost surely finds a Hamilton circuit for the graphs whose average degree is greater than $\ln n$.) They seldom work for the graphs such that Hamiltonian circuits exist, but the average degree is small.

Knowing some specific weak point of one approach, it is quite natural to look for another approach which will compensate for that weak point. It is a little surprising, however, that there have been very few articles focusing on this complementary nature of algorithms for hard combinatorial problems. (Trivial ones like sequential searches in opposite directions usually do not help in the case of hard problems.) In solving CNF SAT, most efforts have been made to improve backtracking by means of heuristics or to analyze the efficiency of backtracking, which has not changed the approach's fundamental weak points.

Recently the author introduced a completely new approach, called IS, to solve CNF SAT [6]. In [6], it was shown that (i) IS is complementary to backtracking in that it becomes faster as the number of literals in each clause increases, and (ii) there is evidence that IS is actually faster than backtracking in a certain class of instances. In this paper, we look into the IS approach from a slightly different angle. It is shown that IS takes polynomial average time for CNF equations satisfying the condition that

* Received by the editors September 14, 1987; accepted for publication (in revised form) July 21, 1988. This work was supported by the Science Foundation Grant of the Ministry of Education, Science and Culture of Japan.

† Department of Computer Sciences, Kyoto Sangyo University, Kyoto 603, Japan.

for a constant c

$$p^2 v \geqq \ln t - c$$

where v is the number of variables, t is the number of clauses, and p is the probability that a given literal appears in a clause (the same for all literals).

Among others, Purdom and Brown have investigated extensively the performance of backtracking for CNF SAT [3], [5], [9], [10], [11], [12], [13], [14]. In [13], [14] it is shown that backtracking (with the pure literal rule) takes polynomial average time when any of the following conditions is satisfied: (1) $t \leqq c \ln v$, (2) $t \geqq \exp(\varepsilon v)$, (3) $p \geqq \varepsilon$, and (4) $p \leqq c(\ln v/v)^{3/2}$, where c is any large constant and ε any small constant (see [13], [14] for the t dependence of bound (4)). In discussing a practical class of instances, it is common to introduce the assumption $t = v^{\alpha}$ for some positive constant α [3], [12]. Then the above (1) and (2) do not apply because neither c nor ε can be constant. (3) and (4) leave the range

$$c(\ln v/v)^{3/2} < p < \varepsilon$$

where there are no known algorithms which take polynomial average time. Our present result makes this unfavorable gap smaller, i.e.,

$$c_1(\ln v/v)^{3/2} < p < c_2(\ln v/v)^{1/2}.$$

It should be noted that this gap is the worst case (when $t = (2 \ln 2)v/\ln v$). For example, when $t = v$ it becomes

$$c_1/v < p < c_2(\ln v/v)^{1/2}.$$

The algorithm, as well as some preliminaries, are described in the next section. In § 3 we analyze the performance of IS and prove our main result. In § 4 we present a preprocessor to reduce the number of clauses and discuss its advantages. Analysis of several heuristics other than this preprocessor (see [6]) will be given in future reports.

2. Algorithm IS. The following small example will clarify the basic idea of the algorithm. Let

$$f_0 = (x_1 + x_2)(x_3 + x_4)(\bar{x}_2 + \bar{x}_4)(\bar{x}_1 + \bar{x}_2)(\bar{x}_1 + x_3 + \bar{x}_4)(\bar{x}_1 + x_2 + \bar{x}_3)$$

and remember the Karnaugh map [8] used to minimize logic functions. Note that the first clause $(x_1 + x_2)$ of f_0 covers four *maxterms* (or *cells* of the map), $(x_1 + x_2 + x_3 + x_4)$, $(x_1 + x_2 + \bar{x}_3 + x_4)$, $(x_1 + x_2 + x_3 + \bar{x}_4)$ and $(x_1 + x_2 + \bar{x}_3 + \bar{x}_4)$. The second clause also covers four cells and so on. Note that cell $(x_1 + x_2 + x_3 + x_4)$ is covered by both the first and the second clauses. It is a fundamental fact that f_0 is satisfiable if and only if the number of cells covered by at least one clause is less than 2^4 (4 is the number of variables). In f_0, the first and second clauses overlap on one cell, as do the second and fourth clauses, the third and fifth clauses, the fourth and fifth clauses, and the third, fourth, and fifth clauses; the third and fourth clauses overlap on two cells. Therefore by the inclusion-exclusion principle we can calculate that

$$4+4+4+4+2+2-(1+1+2+1+1)+1 = 15$$

cells are covered, which means that f_0 is satisfiable since $15 < 2^4$. In fact, one can verify that no clauses cover cell $(x_1 + \bar{x}_2 + \bar{x}_3 + x_4)$. That means that $(x_1, x_2, x_3, x_4) = (0, 1, 1, 0)$ makes the whole f_0 logical 1 (true). The efficiency of this approach clearly depends on the number of overlaps, which can be exponentially large in general.

With the predicate f_0, we can associate the graph $G(f_0)$ of Fig. 1, where the nodes n_1 through n_6 correspond to the clauses s_1 through s_6 ($s_1 = (x_1 + x_2)$, $s_2 = (x_3 + x_4)$ and so on). Edges between two nodes show that the corresponding two clauses do not overlap. Hence two or more clauses that overlap are represented on the graph by two or more nodes that construct an independent set (IS stands for independent sets).

More formally, a (CNF) *predicate* is a product (logical and) of *clauses*, each of which is a sum (logical or) of *literals* (variables x or their negation $\bar{x}$). For the problem set, we assume, as in [5], [11], [13], a set of random predicates with parameters v (the number of variables), t (the number of clauses) and p (the probability that each literal appears in a clause). A random clause is formed by independently selecting each of the $2v$ literals with probability p. A random predicate is formed by independently selecting t random clauses. Thus the clause may consist of no literals (*false*) or may include both x and $\bar{x}$ for some variable x (*tautological*).

For a clause s and a set S of clauses, we define

$$\mathrm{LIT}(s) = \{z | s \text{ includes literal } z\},$$

$$\mathrm{LIT}(S) = \bigcup_{s \text{ in } S} \mathrm{LIT}(s).$$

A set $S(|S| \geqq 1)$ of clauses are said to be *independent* if there is no variable x such that both x and $\bar{x}$ are in $\mathrm{LIT}(S)$. Thus, if S is not independent then no two clauses in S overlap on the Karnaugh map and if S contains a tautological clause then it is not independent even if $|S| = 1$. For an independent set S of clauses of predicate f with v variables, we define

$$\mathrm{SIZE}_f(S) = 2^{v - |\mathrm{LIT}(S)|}.$$

For an integer i, $\mathrm{IND}_f(i)$ is defined as

$$\mathrm{IND}_f(i) = \{S | S \text{ is an independent set of } i \text{ clauses}\}.$$

SIZE_f and IND_f can be written as SIZE and IND, respectively, if predicate f is clear.

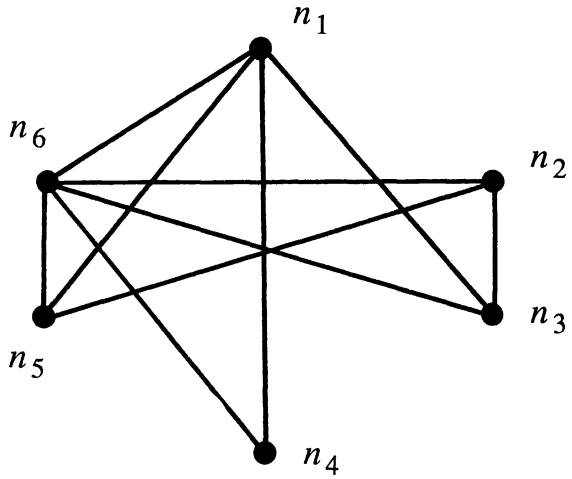


FIG. 1. *Graph* $G(f_0)$.

For the predicate f_0 above,

$$\text{IND}(2) = \{\{s_1, s_2\}, \{s_2, s_4\}, \{s_3, s_4\}, \{s_3, s_5\}, \{s_4, s_5\}\},$$
$$\text{LIT}(\{s_3, s_4\}) = \{\bar{x}_1, \bar{x}_2, \bar{x}_4\},$$
$$\text{SIZE}(\{s_3, s_4\}) = 2^{4-3} = 2.$$

Now here is the algorithm:

procedure IS
begin
  sum := 0; i := 1;
  repeat
    compute IND (i);
    sum := sum + $\sum_{S \text{ in IND}(i)} (-1)^{i-1}$ SIZE (S);
    $i := i+1$
  until IND (i) = ϕ;
  if sum = 2^v **then** answer "unsatisfiable" **else** answer "satisfiable"
end.

Note that SIZE (S) is the total number of the cells on the Karnaugh map that are covered by all the clauses in S. Then it is not hard to see that the algorithm is correct, namely, it counts the number of cells covered by at least one clause of the given predicate f in the repeat loop. There seem to be several methods to compute IND (i) efficiently. For example, the following way, although elementary, does not recompute the same element more than once. Suppose that f consists of t clauses $s_1, s_2, \cdots, s_t$ and that $S = \{s_{j_1}, s_{j_2}, \cdots, s_{j_{i-1}}\}, j_1 < j_2 < \cdots < j_{i-1}$, is in IND ($i-1$). Then check for each s_h, $h > j_{i-1}$, whether $S \cup \{s_h\}$ is independent. If so, we put $S \cup \{s_h\}$ into IND (i). Clearly IND (1) = $\{\{s\} | s$ is any clause but a tautological one$\}$.

Now let us take a look at the number of steps IS needs. For a single member S of IND ($i-1$), we have to check for $O(t)$ clauses s_h whether or not $S \cup \{s_h\}$ is independent. To check if $S \cup \{s_h\}$ is independent, $O(|s_h|)$ steps are enough by computing in advance LIT (S) for all S in IND ($i-1$) and LIT (s_h) for all clauses of f. Thus it takes $O(t \cdot |\text{IND}(i-1)|)$ steps to compute IND (i) if we consider $|s_h|$ a constant. (That is a usual and reasonable assumption since the real size parameter for given predicates is their length, but instead we now take the number of clauses as the size parameter.) To compute *sum*, $O(t \cdot |\text{IND}(i)|)$ steps are obviously enough. Therefore IS takes

$$O(t \cdot N_f)$$

steps where N_f is the number of all independent sets or

$$N_f = |\text{IND}(1)| + |\text{IND}(2)| + |\cdots.$$

The above method of computing IND (i) is simple and easy to implement but requires a lot of memory to hold the whole IND (i) and IND ($i-1$). Note that we can use a standard technique (recursive tree search) to save memory while increasing the number of steps only by a constant factor (see, e.g., pp. 64–69 of [1] for a general idea of this technique).

3. Polynomial average time. In this section, we obtain the expected number N_f of all independent sets of clauses for the predicate f such that the number of variables is v, the number of clauses is t, and the probability that a literal appears in a clause is p. Let $P(k)$ be the probability that k clauses are independent. One can see that the k clauses are independent if, for all variables x, x appears in none of the k clauses or

$\bar{x}$ appears in none of the k clauses. The probability that x appears in none of the k clauses is $(1-p)^k$ and the same for $\bar{x}$. The probability that neither x nor $\bar{x}$ appears is $[(1-p)(1-p)]^k$. Hence

$$
\begin{aligned}
P(k) &= \{(-p)^k + (1-p)^k - [(1-p)(1-p)]^k\}^v \\
&= \{(1-p)^k(2-(1-p)^k)\}^v \\
&= \{(1-p)^k(1+kp-\binom{k}{2}p^2+\cdots)\}^v \\
&\leqq \{(1-p)^k(1+kp)\}^v.
\end{aligned} \tag{1}
$$

Let $\bar{P}(k) = \{(1-p)^k(1+kp)\}^v$. (Note that $\bar{P}(k) < 1$ for $0 < p \leqq 1$ and $k \geqq 1$.)

The value N_f we want can be written as $N_f = \sum_k \binom{t}{k} P(k)$. Let

$$
\bar{N}_f = \sum_k \binom{t}{k} \bar{P}(k) \tag{2}
$$

and

$$
\bar{N}(k) = \binom{t}{k} \bar{P}(k).
$$

Then it is not hard to observe that, as k increases, $\bar{N}(k)$ first increases, becomes maximum at some point $k = k_0$ and then decreases. To obtain k_0 we compute

$$
\begin{aligned}
\frac{\bar{N}(k+1)}{\bar{N}(k)} &= \frac{\dfrac{t(t-1)\cdots(t-k+1)(t-k)}{(k+1)k\cdots 1}[(1-p)^{k+1}(1+(k+1)p)]^v}{\dfrac{t(t-1)\cdots(t-k+1)}{k(k-1)\cdots 1}[(1-p)^k(1+kp)]^v} \\
&= \frac{t-k}{k+1}(1-p)^v\left(1+\frac{p}{1+kp}\right)^v \qquad (3) \\
&\leqq \frac{t-k}{k+1}(1-p)^v(1+p)^v \\
&= \frac{t-k}{k+1}(1-p^2)^v.
\end{aligned}
$$

One can see that k_0 is the least integer that satisfies $\bar{N}(k_0+1)/\bar{N}(k_0) \leqq 1$. Hence $k_1 \geqq k_0$ if k_1 satisfies

$$
\frac{t-k_1}{k_1+1}(1-p^2)^v \leqq 1 \tag{4}
$$

or

$$
\frac{t(1-p^2)^v - 1}{1+(1-p^2)^v} \leqq k_1. \tag{5}
$$

The sum in (2) is less than t times the biggest term, so $\bar{N}_f \leqq t^c$ for some constant c if $k_1 = c-1$. The condition can be rewritten by (5) as

$$
\frac{t(1-p^2)^v - 1}{1+(1-p^2)^v} \leqq c-1. \tag{6}
$$

Clearly $1+(1-p^2)^v \geqq 1$ and therefore (6) holds if

$$t(1-p^2)^v \leqq c \tag{7}$$

or

$$p^2 v \geqq \ln t - \ln c, \tag{8}$$

since $\ln(1-p^2) \leqq -p^2$. Condition (8) guarantees that $N_f \leqq t^c$. We have already shown that it takes IS $O(t \cdot N_f)$ steps in Section 2; hence, the following theorem follows.

THEOREM 1. IS *takes* $O(t^{c+1})$ *steps on average for the class of predicates satisfying the condition that* $p^2 v \geqq \ln t - \ln c$ *for the constant* c.

4. Clause reduction. Consider the following predicate f_1

$$f_1 = (x_1 + \bar{x}_4)(x_1 + \bar{x}_2)(\bar{x}_1 + x_2 + x_3)(x_3 + \bar{x}_4)(x_1 + x_2 + \bar{x}_3)(x_2 + x_4).$$

Note that the first clause of f_1, $(x_1 + \bar{x}_4)$, can be removed without changing f_1's satisfiability since the four cells covered by $(x_1 + \bar{x}_4)$ are also covered by the conjunction of the second, fourth, and fifth clauses

$$(x_1 + \bar{x}_2)(x_3 + \bar{x}_4)(x_1 + x_2 + \bar{x}_3).$$

We can prove this through another satisfiability test, i.e., by showing that predicate

$$F(f_1, (x_1 + \bar{x}_4)) = \bar{x}_2 \cdot x_3 \cdot (x_2 + \bar{x}_3)$$

is not satisfiable. $F(f_1, (x_1 + \bar{x}_4))$ is obtained by (i) picking clauses s out of f_1 such that s and $(x_1 + \bar{x}_4)$ are independent and (ii) deleting the literals x_1 and $\bar{x}_4$ (of $(x_1 + \bar{x}_4)$) appearing in those clauses.

Since removing such clauses clearly makes the given predicate simpler, it is worth trying, if testing of $F(f, s)$'s satisfiability for each clause s of f is much easier than to test the whole f's satisfiability. (The test need not be done for all clauses but for some chosen by a certain heuristic.) Clause reduction may lead to a large saving, and the author believes that it should be investigated further.

5. Concluding remarks. Our main result clearly indicates the complementary nature between backtracking and IS: Backtracking works well for small p and IS for large p. However, there still remains the gap, $c_1(\ln v/v)^{3/2} < p < c_2(\ln v/v)^{1/2}$, where no polynomial-time algorithms are known. Just as the lower bound of this gap has long been raised by improvements of backtracking itself and its performance analysis, the upper bound also will possibly be lowered by future research on IS. Some candidates are (i) to include the close reduction (in § 4) in the algorithm systematically and (ii) to take into account in the analysis another heuristic [6] that makes it possible to stop the main loop before IND(i) becomes empty.

Acknowledgments. The author thanks Y. Okabe for his help in computing N_f in Section 3 and P. Purdom for pointing out an inappropriate approximation in an early version of this paper.

REFERENCES

[1] A. AHO, J. HOPCROFT, AND J. ULLMAN, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] D. ANGLUIN AND L. VALIANT, *Fast probabilistic algorithms for Hamiltonian circuits and matchings*, J. Comput. System Sci., 18 (1979), pp. 155-193.

[3] C. BROWN AND P. PURDOM, JR., *An average time analysis of backtracking*, SIAM J. Comput., 10 (1981), pp. 583-593.

[4] J. FILLMORE AND S. WILLIAMSON, *On backtracking: a combinatorial description of the algorithm*, SIAM J. Comput., 3 (1974), pp. 41-55.
[5] A. GOLDBERG, P. PURDOM, AND C. BROWN, *Average time analysis of simplified Davis-Putnam procedures*, Inform. Process. Lett., 15 (1982), pp. 72-75.
[6] K. IWAMA, *Complementary approaches to* CNF *Boolean equations*, in Discrete Algorithms and Complexity, pp. 223-236, Academic Press, Inc., New York, 1987.
[7] R. KARP, *The probabilistic analysis of some combinatorial search algorithms*, in Algorithms and Complexity, pp. 1-19, Academic Press, Inc., New York, 1976.
[8] Z. KOHAVI, *Switching and Finite Automata Theory*, McGraw-Hill, Inc., New York, 1970.
[9] P. PURDOM, *Tree size by partial backtracking*, SIAM J. Comput., 7 (1978), pp. 481-491.
[10] P. PURDOM, C. BROWN, AND E. ROBERTSON, *Backtracking with multi-level dynamic search rearrangement*, Acta Inform., 15 (1981), pp. 99-113.
[11] P. PURDOM, *Search rearrangement backtracking and polynomial average time*, Artificial Intelligence, 21 (1983), pp. 117-133.
[12] P. PURDOM AND C. BROWN, *An analysis of backtracking with search rearrangement*, SIAM J. Comput., 12 (1983), pp. 717-733.
[13] ———, *The pure literal rule and polynomial average time*, SIAM J. Comput., 14 (1985), pp. 943-953.
[14] ———, *Polynomial-average-time satisfiability problems*, Inform. Sci., 41 (1987), pp. 23-42.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 392-408, April 1989

015

RELATIVIZED POLYNOMIAL TIME HIERARCHIES HAVING EXACTLY K LEVELS*

KER-I KO†

Abstract. It is proved that for every integer $k \geq 0$, there is an oracle A_k relative to which the polynomial time hierarchy collapses so that it has exactly k levels. Furthermore, sets B_k and C_k may be constructed so that, relative to B_k, the polynomial time hierarchy has exactly k levels and the class PSPACE coincides with the polynomial time hierarchy, and, relative to C_k, the polynomial time hierarchy has exactly k levels and the class PSPACE is different from the polynomial time hierarchy.

Key words. polynomial time hierarchy, relativization, oracle

AMS(MOS) subject classification. 68C25

1. Introduction. One of the main goals in complexity theory is to develop proof techniques to separate complexity classes. While it is well recognized that most separation results are beyond today's proof techniques, interesting progress has been made recently on separation results for relativized complexity classes. Baker, Gill, and Solovay [2] showed that the relativized P = ?NP question may be answered in both ways depending on the oracles; i.e., there exist sets X and Y such that $\mathrm{P}(X) = \mathrm{NP}(X)$ and $\mathrm{P}(Y) \neq \mathrm{NP}(Y)$. Baker and Selman [3] extended it to the second level of the polynomial time hierarchy showing that there exists a set Z such that $\Sigma_2^P(Z) \neq \Sigma_3^P(Z)$. The proof technique of Baker and Selman's result is a complicated counting argument which, however, does not seem powerful enough to be applicable to separating the third level of the relativized polynomial time hierarchy.

More recently, Furst, Saxe, and Sipser [4] and Sipser [9] proposed the idea of applying probabilistic arguments to this problem. They reduced the problem of separating the relativized polynomial time hierarchy to the problem of proving lower bounds on the size of constant depth circuits. The major breakthrough in this direction is due to Yao [12] who, based on Furst, Saxe, and Sipser's idea, showed an exponential lower bound on the size of constant depth parity circuits and hence exhibited an oracle A which separates the class PSPACE (A) from PH (A). Hastad [5], [6] simplified Yao's proof and gave a proof for the claim made in [12] that there exists an oracle B such that for all $k > 0$, $\mathrm{PH}(B) \neq \Sigma_k^P(B)$. We summarize the known results about the relativized polynomial time hierarchies as follows:

(1) $\forall A \forall k > 0 [\Sigma_k^P(A) = \Pi_k^P(A) \Rightarrow \mathrm{PH}(A) = \Sigma_k^P(A)]$ (Stockmeyer [10]).

(2) $\exists B\ \mathrm{PSPACE}(B) = \mathrm{P}(B)$ (Baker, Gill, and Solovay [2]).

(3) $\exists C \forall k > 0\ \mathrm{PSPACE}(C) \neq \mathrm{PH}(C) \neq \Sigma_k^P(C)$ (Yao [12] and Hastad [5], [6]).

(4) $\forall k = 1, 2, \exists D_k\ \mathrm{PH}(D_k) = \Sigma_k^P(D_k) \neq \Sigma_{k-1}^P(D_k)$ (Baker, Gill, and Solovay [2], Heller [7]).

From the above results, the relativized polynomial time hierarchies may have quite different structures depending on the oracles. However, these results have not exhausted all possible structures of the relativized polynomial time hierarchies. For example, the following question remains open: does there exist a set D_k for each $k \geq 3$ such that

* Received by editors June 30, 1987; accepted for publication July 21, 1988. This research was supported in part by National Science Foundation grant CCR-8696135; a preliminary version of this paper has appeared in the 20th ACM Symposium on Theory of Computing (1988), pp. 245-253.

† Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794.

(4) above holds? Furthermore, if such sets D_k exist, can we construct them to also separate PSPACE (D_k) from PH (D_k)? In this paper, we show that the probabilistic arguments developed by Yao [12] and Hastad [5], [6] are powerful enough to construct oracles D_k with the above required properties. More precisely, we prove the following results.

(5) $\forall k \geqq 1 \exists E_k$ PSPACE $(E_k) = \Sigma_k^P(E_k) \neq \Sigma_{k-1}^P(E_k)$.

(6) $\exists F_0$ PSPACE $(F_0) \neq$ NP $(F_0) = P(F_0)$.

(7) $\forall k \geqq 1 \exists F_k$ PSPACE $(F_k) \neq$ PH $(F_k) = \Sigma_k^P(F_k) \neq \Sigma_{k-1}^P(F_k)$.

The proof techniques for these results are the combination of the encoding scheme of Baker, Gill, and Solovay [2] and the probabilistic arguments of Yao [12] and Hastad [5]. The main complication comes from the possible interference between the two constructions, which can be handled by using slightly different formulations of Yao and Hastad's basic lemmas.

2. Preliminaries.

2.1. Basic notation. In this paper, all sets A are sets of strings over the alphabet $\Sigma = \{0, 1\}$. For each string x, let $|x|$ denote its *length*. Let Σ^n be the set of all strings of length n. We assume that there is a one-to-one *pairing function* $\langle , \cdots , \rangle$ that encodes an arbitrary number of strings $x_1, \cdots, x_n$ into a single string $\langle x_1, \cdots, x_n \rangle$. We assume that $|\langle x_1, \cdots, x_n \rangle| \geqq \Sigma_{i=1}^n |x_i|$. For each set A, let χ_A be its characteristic function; i.e., for each x, $\chi_A(x) = 1$ if $x \in A$, and $\chi_A(x) = 0$ if $x \notin A$.

2.2. Complexity classes. We assume that the reader is familiar with Turing machines (TMs), nondeterministic TMs, oracle TMs, nondeterministic oracle TMs, and their time and space complexity (see, for example, [8]). Let $\mathscr{C}$ be a class of sets. We let P $(\mathscr{C})$ (NP $(\mathscr{C})$) denote the class of sets that are computable in polynomial time by a deterministic (nondeterministic, respectively) oracle TM using some set $A \in \mathscr{C}$ as an oracle. If $\mathscr{C} = \{A\}$ then we write P (A), NP (A) for P $(\{A\})$, NP $(\{A\})$, respectively. Also, if $A = \varnothing$ then we write P, NP for P $(\varnothing)$, NP $(\varnothing)$, respectively. Let PSPACE $(\mathscr{C})$ denote the class of sets that are computable in polynomial space by a deterministic oracle TM using some set $A \in \mathscr{C}$ as an oracle. The *relativized polynomial time hierarchy* is defined as follows. For any class $\mathscr{C}$, let *co-*$\mathscr{C}$ be the class of sets whose complements are in $\mathscr{C}$.

$$\Sigma_0^P(\mathscr{C}) = \Pi_0^P(\mathscr{C}) = \Delta_0^P(\mathscr{C}) = \mathrm{P}(\mathscr{C});$$

$$\Sigma_{k+1}^P(\mathscr{C}) = \mathrm{NP}(\Sigma_k^P(\mathscr{C}));$$

$$\Pi_{k+1}^P(\mathscr{C}) = co\text{-}\Sigma_{k+1}^P(\mathscr{C});$$

$$\Delta_{k+1}^P(\mathscr{C}) = \mathrm{P}(\Sigma_k^P(\mathscr{C})).$$

Let PH $(\mathscr{C})$ be the union of all of the above classes. When $\mathscr{C} = \{A\}$ or $\mathscr{C} = \{\varnothing\}$, we also use simpler notation $\Sigma_k^P(A)$ or Σ_k^P, respectively, for $\Sigma_k^P(\mathscr{C})$, etc.

The relativized polynomial time hierarchy PH $(\mathscr{C})$ can be characterized by alternating quantifiers. Let $R(A; x)$ be a predicate over a set variable A and a string variable x. We say that $R(A; x)$ is a P^1-*predicate* if R is computable in polynomial time by a deterministic oracle machine that uses set A as the oracle and takes string x as the input. (The superscript 1 indicates that the predicate is on a type-1 object.) Let $k \geqq 1$. We say $\sigma(A; x)$ is a $\Sigma_k^{P,1}$-*predicate* if there exist a P^1-predicate $R(A; x, y_1, \cdots, y_k)$ over a set variable and $k+1$ string variables and a polynomial q, such that for all sets A and all x with $|x| = n$, $\sigma(A; x)$ is true if and only if

$$(\exists y_1, |y_1| \leqq q(n))(\forall y_2, |y_2| \leqq q(n)) \cdots (Q_k y_k, |y_k| \leqq q(n)) R(A; x, y_1, \cdots, y_k),$$

where $Q_k = \exists$ if k is odd, and $Q_k = \forall$ if k is even. It is well known that a set B is in $\Sigma_k^P(\mathscr{C})$ if and only if there exist a $\Sigma_k^{P,1}$-predicate σ and a set $A \in \mathscr{C}$ such that for all x, $[x \in B \Leftrightarrow \sigma(A; x)]$ [10], [11].

2.3. Enumerations of machines. We will assume a fixed enumeration $\{p_i\}$ of polynomials, a fixed enumeration $\{M_i\}$ for all polynomial time oracle TMs, a fixed enumeration $\{N_i\}$ of all polynomial time nondeterministic oracle TMs, and, for each $k \geqq 1$, a fixed enumeration $\{\sigma_i^k\}$ of all $\Sigma_k^{P,1}$-predicates. We assume that the ith machine M_i or N_i has its runtime bounded by the ith polynomial p_i. Without loss of generality, we may assume that $p_i(n) \leqq n^i$ for sufficiently large n. Let $k > 1$. We assume that the ith $\Sigma_k^{P,1}$-predicate $\sigma_i^k(A; x) \equiv (\exists y_1, |y_1| \leqq q(n))(\forall y_2, |y_2| \leqq q(n)) \cdots (Q_k y_k, |y_k| \leqq q(n))$ $R(A; x, y_1, \cdots, y_k)$ has the property that both the length-bonding polynomial q and the runtime of the deterministic oracle TM that computes the predicate R are bounded by the ith polynomial p_i. We say that the computation of σ_i^k on input x queries about the string z if the computation of the corresponding P^1-predicate $R(A; x, y_1, \cdots, y_k)$ queries about the string z for some $y_1, \cdots, y_k$ of length $\leqq p_i(n)$ and for some oracle A.

2.4. Complete sets. For any of the above defined class $\mathscr{D}$, a set $A \in \mathscr{D}$ is $(\leqq_m^P)$-*complete* for $\mathscr{D}$ if for every set $B \in \mathscr{D}$ there exists a polynomial time computable function f such that for all x, $x \in B$ if and only if $f(x) \in A$. We will use some specific complete sets for these classes. Define, for each set A, the set $K(A)$ to be $\{\langle i, z, 1^j\rangle |$ the nondeterministic oracle TM N_i accepts z in j moves when A is used as the oracle$\}$. Then, it is obvious that $K(A)$ is complete for NP(A). Furthermore, for any string x, the question of whether $x \in K(A)$ depends only on the set $\{y \in A \mid |y| < |x|\}$, because $x = \langle i, z, 1^j\rangle$ implies $j < |x|$. (In other words, if B agrees with A on strings of length $< |x|$, then $x \in K(A)$ if and only if $x \in K(B)$.) We can extend this to $\Sigma_k^P(A)$-complete sets for $k > 1$. Let $K^1(A) = K(A)$ and $K^k(A) = K(K^{k-1}(A))$ for $k > 1$. Then, for each $k \geqq 1$ and each set A, $K^k(A)$ is complete for $\Sigma_k^{P,A}$, and the question of whether $x \in K^k(A)$ depends only on the set $\{y \in A \mid |y| < |x|\}$ (this can be proved by induction).

Define, for each set A, the set $Q(A)$ to be $\{\langle i, x, 1^j\rangle |$ the ith oracle machine M_i accepts x using at most j cells$\}$. Then $Q(A)$ is complete for the class PSPACE(A) and the question of whether $x \in Q(A)$ depends only on the set $\{y \in A \mid |y| < |x|\}$.

2.5. Circuits. We will deal with *circuits* of unbounded fanin which have only AND and OR gates and which have variables or its negations as inputs. We formally define a circuit as a tree. Each interior node of the tree is attached with an AND gate or an OR gate, and has an unlimited number of child nodes. Unless otherwise specified, it is assumed that the gates alternate so that all children of an OR (or AND) gate are AND (or OR, respectively) gates. Each leaf is attached with a constant 0, a constant 1, a variable v, or a negated variable $\bar{v}$. In this paper, we will relate circuits to oracle TMs. So, each variable in C is represented by a string $z \in \Sigma^*$. We write v_z to denote the variable, which will eventually be given the value $\chi_A(z)$ for some set A, and write $\overline{v_z}$ to denote its negation, which will eventually be given the value $1 - \chi_A(z)$. Each circuit computes a boolean function on its variables. The *depth* of a circuit is the length of the longest path in the tree. The *size* of a circuit is the number of gates (or, the number of interior nodes) in the tree. The *fanin* of a gate is the number of children of the node. The *bottom fanin* of a circuit is the maximum fanin of a gate of the lowest level in the tree.

Each circuit C has a dual circuit $\tilde{C}$ which has the same tree structure as C but computes the negation of the function computed by C. Formally, the dual circuit of a single variable v is its negation $\bar{v}$. The dual circuit $\tilde{C}$ of a circuit C of depth $\geqq 1$ and

with a top OR (AND) gate is the circuit with a top AND (OR, respectively) gate and its children being the dual circuits of the children of the top gate of C.

Let V be the set of variables which occur in a circuit C. Then a *restriction* ρ of C is a mapping from V to $\{0, 1, *\}$. For each restriction ρ of C, $C\lceil_\rho$ denotes the circuit C' obtained from C by replacing each variable v_x with $\rho(v_x)=0$ by 0 and each v_y with $\rho(v_y)=1$ by 1 (and each v_z with $\rho(v_z)=*$ remaning a variable). Assume that ρ' is a restriction of $C\lceil_\rho$. We write $C\lceil_{\rho\rho'}$ to denote $(C\lceil_\rho)\lceil_{\rho'}$. We also write $\rho\rho'$ to denote the combined restriction on C with values $\rho\rho'(v_x)=\rho(v_x)$ if $\rho(v_x)\neq *$ and with values $\rho\rho'(v_x)=\rho'(v_x)$ if $\rho(v_x)=*$. If a restriction ρ of C maps no variable to $*$, then we say ρ is an *assignment* of C. Let ρ be a restriction of C, we say that ρ *completely determines* C if $C\lceil_\rho$ computes a constant function 0 or 1. An assignment ρ of C always completely determines the circuit C.

There are some specific circuits that are useful in our proofs. A particularly interesting class of circuits has been used by Sipser [9], and later adopted by Hastad [5], to define the functions f_k^m. Our definition of function f_k^m is a little different from those defined in [5] and [9]. Let C_k^m be a depth-k circuit having the following properties:

(a) the top gate of C_k^m is an OR gate with fanin $\sqrt{m}$,

(b) the fanin of all bottom gates of C_k^m is $\sqrt{m}$,

(c) the fanin of all other gates is m, and

(d) there are m^{k-1} variables each of which occurs exactly once in a leaf in the positive form.

Let the function computed by C_k^m be f_k^m.

2.6. Circuits and relativized complexity classes. The following relation between the relativized polynomial time hierarchy and constant depth circuits is due to Furst, Saxe, and Sipser [4].

LEMMA 2.1 [4]. *Let $k\geqq 1$ and $q(n)$ and $r(n)$ be two polynomial functions. Let $\sigma(A; x)=(\exists y_1, |y_1|\leqq q(n))(\forall y_2, |y_2|\leqq q(n))\cdots(Q_k y_k, |y_k|\leqq q(n))R(A; x, y_1, \cdots, y_k)$ be a $\Sigma_k^{P,1}$-predicate, where $n=|x|$ and $R(A; x, y_1, \cdots, y_k)$ is computable in time $r(n)$ by a deterministic oracle* TM *M using oracle A. Then, for each string x, there exists a circuit C having the following properties*:

(a) *the depth of C is $k+1$,*

(b) *the fanin of each gate in C is $\leqq 2^{q(n)+r(n)}$,*

(c) *the bottom fanin of C is $\leqq r(n)$,*

(d) *the variables of C are represented by strings queried by M on input $(x, y_1, \cdots, y_k)$ for some $y_1, \cdots, y_k$ of length $\leqq q(n)$ and some oracle A, and*

(e) *for each set A, if we use $\chi_A(z)$ as the input value for each variable v_z in C then C outputs 1 if and only if $\sigma(A; x)$ is true.*

Sketch of Proof. The proof is done by induction on k. For the case $k=1$, let $\sigma(A; x)=(\exists y, |y|\leqq q(n))R(A; x, y)$, where R is computable by a deterministic oracle TM M in time $r(n)$. Then, for each y of length $\leqq q(n)$, consider the computation tree T of $M(x, y)$. Each path of T corresponds to a sequence of answers to queries made by M. For each accepting path in T, let U_1 be the set of strings answered positively by the oracle and U_0 the set of strings answered negatively by the oracle. Define an AND gate with $|U_0\cup U_1|$ many children, each attached with a variable v_z, with $z\in U_1$, or the negation $\overline{v_z}$ of a variable v_z, with $z\in U_0$. Then, the OR of all these AND gates is a circuit G_y (depending only on x and y) which computes $R(A; x, y)$ when each variable v_z in G_y is given the value $\chi_A(z)$. Note that each AND gate of the circuit G_y has $\leqq r(n)$ children. Now consider the circuit C which is the OR of all G_y's for y of length $\leqq q(n)$. We know that C computes $\sigma(A; x)$ when each variable v_z in C is given

value $\chi_A(z)$. Furthermore, combining all OR gates of G_y's into one, C can be written as a depth-2 circuit with top fanin $\leqq 2^{q(n)+r(n)}$ and bottom fanin $\leqq r(n)$.

For the inductive step, assume that

$$\sigma(A; x) = (\exists y_1, |y_1| \leqq q(n))(\forall y_2, |y_2| \leqq q(n)) \cdots (Q_k y_k, |y_k| \leqq q(n)) R(A; x, y_1, \cdots, y_k).$$

Then, for each y_1 of length $\leqq q(n)$ there is a depth-k circuit C_{y_1} satisfying the conditions (a)-(e) with respect to predicate

$$\tau(A; x, y_1) = (\exists y_2, |y_2| \leqq q(n)) \cdots (Q_{k-1} y_k, |y_k| \leqq q(n))[\text{not } R(A; x, y_1, \cdots, y_k)].$$

Take the dual circuits $\tilde{C}_{y_1}$ of C_{y_1} and let C be the OR of all these circuits. Then, C is the circuit we need. □

Another interesting relation between circuits and sets in relativized polynomial time hierarchies is about complete sets $K^k(A)$. The following lemma will not be used in § 3. The reader may wish to skip it until § 4.

LEMMA 2.2. *Let $k \geqq 1$. For every x of the form $\langle i, y, 1^j \rangle$, there is a circuit C such that*

(a) *the depth of C is $\leqq 2k$,*

(b) *the fanin of each gate is C is $\leqq 2^{|x|}$,*

(c) *the bottom fanin of C is $\leqq |x|$,*

(d) *the variables of C are v_z's over strings z of length $\leqq |x|$, and*

(e) *for each set A, if we use $\chi_A(z)$ as the input value for each variable v_z in C then C outputs 1 if and only if $x \in K^k(A)$.*

Proof. We prove the lemma by induction on k. First, let $k = 1$. Then, $x = \langle i, y, 1^j \rangle \in K(A)$ if and only if the machine N_i accepts y in $\leqq j$ moves using A as an oracle. That is, $x \in K(A)$ if and only if $(\exists w, |w| \leqq j) R(A; y, w)$ for some P^1-predicate R which is computable in time $\leqq j - |w|$ (without loss of generality, we may assume that N_i is a specific machine which first uses $|w|$ moves to generate the witness string w and then uses $\leqq j - |w|$ moves to compute R). By Lemma 2.1, there is a depth-2 circuit C such that its top gate has fanin $\leqq 2^j \leqq 2^{|x|}$, its bottom fanin is $\leqq j \leqq |x|$, the variables in C are represented by strings of length $\leqq j \leqq |x|$, and for each set A, if we use $\chi_A(z)$ as the input value for each variable v_z in C then C outputs 1 if and only if $x \in K(A)$.

Assume that $k > 1$. Let $x = \langle i, y, 1^j \rangle$ be given. Then, $x \in K^k(A)$ if and only if the machine N_i accepts y in $\leqq j$ moves using $K^{k-1}(A)$ as an oracle. As in the case $k = 1$, there is a depth-2 circuit C_1 such that its top gate has fanin $\leqq 2^j \leqq 2^{|x|}$, its bottom fanin is $\leqq j \leqq |x|$, its variables are represented by strings of length $\leqq |x|$, and for each set A, if we use $\chi_{K^{k-1}(A)}(z)$ as the input value for each variable v_z in C_1 then C_1 outputs 1 if and only if $x \in K^k(A)$.

Now, by the inductive hypothesis, for each string z of the form $\langle i_1, u, 1^{j_1} \rangle$ such that variable v_z occurs in C_1, there is a circuit C_z of depth $2(k-1)$ such that

(a) the fanin of each gate of C_z is $\leqq 2^{|z|}$,

(b) the bottom fanin of C_z is $\leqq |z|$,

(c) the variables in C_z are represented by strings of length $\leqq |z|$, and

(d) for any set A, if we use $\chi_A(w)$ as the input value for each variable v_w in C_z, then C_z outputs 1 if and only if $z \in K^{k-1}(A)$.

For each variable v_z in C_1 such that z is not of such a form, let C_z be the constant 0 (because $z \notin K^{k-1}(A)$). Replace each variable v_z in C_1 by the circuit C_z (i.e., each leaf with the variable v_z is replaced by the tree C_z, and each leaf with the negated variable $\overline{v_z}$ is replaced by the dual circuit $\tilde{C}_z$ of C_z). Since strings z corresponding to variables v_z in C_1 have length $|z| \leqq |x|$, we obtain a circuit C of depth $2k$ such that the fanin of each gate of C is $\leqq 2^{|x|}$, the bottom fanin of C is $\leqq |x|$, and the variables in C are

represented by strings of length $\leqq|x|$. (It should be pointed out that in circuit C, a child of an AND gate is not necessarily an OR gate. If we combine all adjacent AND gates and OR gates, then the resulting circuit may have fanin $\leqq|x|\cdot 2^{|x|}$.)

Finally, for any set A, if we use $\chi_A(w)$ as the input value for each variable v_w in C, then for each variable v_z in C_1, the circuit C_z outputs 1 if and only if $z\in K^{k-1}(A)$; therefore, the circuit C outputs 1 if and only if $x\in K^k(A)$. This completes the proof. □

3. Relativized hierarchy having exactly k levels. In this section we prove that for each $k\geqq 1$, there exists an oracle A such that $\Sigma_k^P(A)=\Pi_k^P(A)\neq\Sigma_{k-1}^P(A)$. We need a lower bound result on constant depth circuits. Hastad [5] has proved that there exists a function f_k^m computable by a polynomial-size depth-k circuit but not by any depth-k circuit with small bottom fanin. The following lemma is a stronger form of this result. It states that no depth-k circuit with small bottom fanin can compute any of an exponential number of f_k^m functions. The main idea of the proof is the same as that of Hastad's proof. We give the formal proof in the Appendix.

Recall that C_k^m is a circuit defining the function f_k^m. Let $\mathrm{CIR}(k, t)$ be the class of depth-k circuits which have size $\leqq 2^t$ and bottom fanin $\leqq t$.

LEMMA 3.1. *For every $k\geqq 2$ there exists a constant n_k such that the following holds for all $n>n_k$. Let $t=n^{\log n}$, $m<2^t$, and $C_0, C_1, \cdots, C_m$ be $m+1$ circuits each defining a $f_k^{2^n}$ function, with their variables pairwisely disjoint. Let C be a circuit in $\mathrm{CIR}(k, t)$. Then, there exists a restriction ρ on C such that ρ completely determines C but it does not completely determine any C_i, $0\leqq i\leqq m$.*

THEOREM 3.2. *For each $k\geqq 1$, there exists a set A such that $\Sigma_k^P(A)=\Pi_k^P(A)\neq\Sigma_{k-1}^P(A)$.*

Proof. The case $k=1$ has been proven by Baker, Gill, and Solovay [2]. We assume that $k\geqq 2$. (Actually, the case $k=2$ has been proven by Baker and Selman [3] and Heller [7].)

Recall that $\{p_i\}$ is an enumeration of polynomial functions, $\{M_i\}$ is an enumeration of all polynomial time deterministic oracle TMs, and $\{\sigma_i^{k-1}\}$ is an enumeration of all $\Sigma_{k-1}^{P,1}$-predicates which are of the form

$$\sigma_i^{k-1}(A; x)=(\exists y_1, |y_1|\leqq q(n))(\forall y_2, |y_2|\leqq q(n))\cdots (Q_{k-1}y_{k-1}, |y_{k-1}|\leqq q(n))R(A; x, y_1, \cdots, y_{k-1}),$$

where $n=|x|$, $q(n)\leqq p_i(n)$, and $R(A; x, y_1, \cdots, y_{k-1})$ is computable in time $p_i(n)$ by some deterministic oracle machine M. Also recall that $K^k(A)$ is a complete set for $\Sigma_k^P(A)$ which has the property that the question of whether $x\in K^k(A)$ depends only on the set $\{y\in A\,|\,|y|<|x|\}$. Let

$$L_k(A)=\{1^n\,|\,(\exists z_1, |z_1|=n)(\forall z_2, |z_2|=n)\cdots(Q_kz_k, |z_k|=n)1^nz_1z_2\cdots z_k\in A\}.$$

Note that $L_k(A)$ is in $\Sigma_k^P(A)$.

The construction of the oracle A will be done by stages. At each stage $\alpha=(k+1)n+1$, we will satisfy the requirement

$R_{0,n}$: for all strings u of length n, $u\notin K^k(A)$
$\Leftrightarrow(\exists z_1, |z_1|=n)(\forall z_2, |z_2|=n)\cdots(Q_kz_k, |z_k|=n)0uz_1z_2\cdots z_k\in A.$

At stage $\alpha=(k+1)n$, we will try to satisfy the following requirement $R_{1,i}$ with the least integer i for which $R_{1,i}$ is not yet satisfied:

$R_{1,i}$: there exists an n_i such that $1^{n_i}\in L_k(A)$ if and only if $\sigma_i^{k-1}(A; 1^{n_i})$ is false.

The requirement $R_0=\Lambda_{n=1}^{\infty}R_{0,n}$ states that $K^k(A)$ is in $\Pi_k^P(A)$, and hence $\Sigma_k^P(A)=$

$\Pi_k^P(A)$. The requirement $R_1 = \Lambda_{i=1}^{\infty} R_{1,i}$ states that $L_k(A)$ is not in $\Sigma_{k-1}^P(A)$. Therefore a set A satisfying all requirements has the property $\Sigma_k^P(A) = \Pi_k^P(A) \neq \Sigma_{k-1}^P(A)$.

The main difficulty of the construction is that when we try to satisfy requirement $R_{1,i}$ in stage $\alpha = (k+1)n$, we may have to simulate some oracle machine M which may query about strings of length longer than α. We cannot arbitrarily assign answers to the queries made by M because such an assignment may conflict with requirement $R_{0,m}$ for some $m > n$. What we need is an assignment of answers to queries which does not conflict with future constructions at stage $\alpha' > \alpha$. The existence of such an assignment will be proved by using Lemma 3.1.

In each stage α, we will determine an initial segment of set A by putting some strings into set A and some into $\bar{A}$. We let $A(\alpha)$ denote the set of strings reserved for A by stage α and $A'(\alpha)$ denote the set of strings reserved for $\bar{A}$ by stage α. Sets $A(\alpha)$ and $A'(\alpha)$ are defined so that $A(\alpha)$ is always an extension of $A(\alpha-1)$, $A'(\alpha)$ is always an extension of $A'(\alpha-1)$, and $A(\alpha) \cap A'(\alpha) = \emptyset$. Also, in stage α we do not add any string of length $< \alpha$ to $A(\alpha)$ or $A'(\alpha)$. Eventually, we will define set A to be the union of all $A(\alpha)$. That is, a string x is in $\bar{A}$ either if it is put in $A'(\alpha)$ at some stage α or if it has never been touched in the construction.

In the construction, we keep track of all integers i for which the corresponding requirement $R_{1,i}$ has been satisfied. We let all integers i for which $R_{1,i}$ is not yet satisfied be *uncancelled.* If the requirement $R_{1,i}$ is satisfied in stage α, then we *cancel* this integer i at this stage. To avoid the potential interference between requirements $R_{1,i}$ and $R_{1,j}$ for $i \neq j$, we set a pointer β_α in each stage α. The integer β_α is defined to be an upper bound of the maximum length of strings added to $A(m)$ or $A'(m)$ in stages $m \leqq \alpha$. When $\alpha > \beta_{\alpha-1}$, the construction in stage α can be done without interfering with the constructions made in earlier stages.

Prior to stage 1, assume that $A(0) = A'(0) = \emptyset$, and let $\beta_0 = 1$. Let all integers i be uncancelled.

Stage α, where α does not have the form $\alpha = (k+1)n+1$ or $\alpha = (k+1)n$. Do nothing. Let $A(\alpha) := A(\alpha-1)$, $A'(\alpha) := A'(\alpha-1)$ and $\beta_\alpha := \beta_{\alpha-1}$.

Stage $\alpha = (k+1)n$. Let i be the least integer that is not yet cancelled. If $\alpha \leqq \beta_{\alpha-1}$ or $n \leqq n_k$ (n_k is the constant defined in Lemma 3.1) or $2kp_i(n) \geqq t = n^{\log n}$, then do nothing. Let $A(\alpha) := A(\alpha-1)$, $A'(\alpha) := A'(\alpha-1)$, and $\beta_\alpha := \beta_{\alpha-1}$.

If $\alpha > \beta_{\alpha-1}$ and $n > n_k$ and $2kp_i(n) < t = n^{\log n}$, then consider the following circuits:

(1) For each u of length $n \leqq |u| \leqq p_i(n)$, the circuit C_u of depth k is defined as follows:

(a) the top gate of C_u is an OR gate,

(b) the fanin of each gate of C_u is $2^{|u|}$,

(c) the variables of C_u are exactly those in $\{v_y \mid y \in 0u\Sigma^{k|u|}\}$; each occurred positively in exactly one leaf of C_u in the increasing order (under the lexicographic order on y).

There are totally $\leqq 2^{p_i(n)} < 2^t$ many such circuits C_u. Note that each circuit C_u has the property that for all sets A, if we use $\chi_A(y)$ as the input value for each variable v_y then C_u outputs 1 if and only if

$$(\exists z_1, |z_1| = |u|)(\forall z_2, |z_2| = |u|) \cdots (Q_k z_k, |z_k| = |u|) 0u z_1 z_2 \cdots z_k \in A.$$

Also note that each circuit C_u contains a subcircuit computing a function $f_k^{2^n}$.

(2) The circuit C_0 has the same tree structure as the circuit C_u defined above, with $|u| = n$, except that the variables of C_0 are those in $\{v_y \mid y \in 1^n\Sigma^{kn}\}$. Note that if we use $\chi_A(y)$ as the input value for each variable v_y then C_0 outputs 1 if and only if $1^n \in L_k(A)$.

(3) The circuit C is the circuit associated with the $\Sigma_{k-1}^{P,1}$-predicate $\sigma_i^{k-1}(A; 1^n)$ as defined in Lemma 2.1, with the restriction that each variable v_y with $|y| < \alpha = (k+1)n$ is replaced by the constant value $\chi_{A(\alpha-1)}(y)$. In particular, C has depth k, has size $\leqq 2^{2kp_i(n)}$, and has the bottom fanin $\leqq p_i(n)$. The variables in C are represented by strings of length $\leqq p_i(n)$. For each set A which agrees with $A(\alpha-1)$ on strings of length $<\alpha$, if we use $\chi_A(y)$ as the input value for each variable v_y then C outputs 1 if and only if $\sigma_i^{k-1}(A; 1^n)$ is true.

It is easy to see that each circuit C_u or C_0 contains a subcircuit C'_u or C'_0, respectively, which defines a $f_k^{2^n}$ function. Choose a restriction ρ such that for each u, $C'_u = C_u\lceil_\rho$ computes exactly a $f_k^{2^n}$ function and $C'_0 = C_0\lceil_\rho$ computes exactly a $f_k^{2^n}$ function. We observe that by the choice of n such that $2kp_i(n) < t = n^{\log n}$, $C \in \mathrm{CIR}(k, t)$. So, $C' = C\lceil_\rho$ is also in $\mathrm{CIR}(k, t)$. Furthermore, the number of circuits C'_u is $<2^t$. Thus, we can apply Lemma 3.1 to the circuits C'_0 and C'_u, $n \leqq |u| \leqq p_i(n)$, and the circuit C' to obtain a restriction ρ' of C' such that ρ' completely determines the circuit C' but not circuit C'_0 nor any circuit C'_u, $n \leqq |u| \leqq p_i(n)$. Finally, we find an assignment ρ'' of variables of $C'_0\lceil_{\rho'}$ such that $C'_0\lceil_{\rho'\rho''}$ computes a constant function 1 if and only if $C'\lceil_{\rho'}$ computes a constant function 0. Note that ρ'' only assigns values to variables corresponding to strings in $1^n\Sigma^{kn}$, and so none of $C'_u\lceil_{\rho'}$ is completely determined by ρ''.

Define $A(\alpha) := A(\alpha-1) \cup \{y \mid \rho\rho'\rho''(v_y) = 1\}$ and $A'(\alpha) := A'(\alpha-1) \cup \{y \mid \rho\rho'\rho''(v_y) = 0\}$. Let $\beta_\alpha = \max\{\alpha, p_i(n)+1\}$ and cancel i. This completes stage $\alpha = (k+1)n$.

Stage $\alpha = (k+1)n+1$. For each u of length n, we determine whether $u \in K^k(A(\alpha-1))$. Then, we find a subset $B \subseteq \{y \in 0u\Sigma^{kn} \mid y \notin A(\alpha-1) \cup A'(\alpha-1)\}$ such that $u \notin K^k(A(\alpha-1)) \Leftrightarrow (\exists z_1, |z_1| = n)(\forall z_2, |z_2| = n) \cdots (Q_k z_k, |z_k| = n) 0uz_1z_2 \cdots z_k \in A(\alpha-1) \cup B$. (We will show that such a set B always exists.) Let $A(\alpha) = A(\alpha-1) \cup B$ and $A'(\alpha) = A'(\alpha-1)$. Let $\beta_\alpha = \max\{\alpha, \beta_{\alpha-1}\}$. Stage $\alpha = (k+1)n+1$ is complete when we finish the above construction for each u of length n.

Let $A = \bigcup_{\alpha=1}^{\infty} A(\alpha)$. First, we claim that in each stage $\alpha = (k+1)n+1$; for each u of length n, the set B can be found.

Proof of claim. If none of strings in $0u\Sigma^{kn}$ has been assigned to $A(\alpha-1)$ or $A'(\alpha-1)$, then certainly such a set B exists. Assume that some strings in $0u\Sigma^{kn}$ have been assigned to $A(\alpha-1)$ or $A'(\alpha-1)$. Then, by the choice of β_α, there is at most one stage $\alpha' = (k+1)m < \alpha$ in which these assignments are made.

In that stage, the assignments are made such that the corresponding circuit C_u, after applying the restriction $\rho\rho'\rho''$, is not completely determined. Note that the circuit C_u and the predicate

$$\tau(A; u) = (\exists z_1, |z_1| = n)(\forall z_2, |z_2| = n) \cdots (Q_k z_k, |z_k| = n) 0uz_1z_2 \cdots z_k \in A$$

has the relation that when assigning value $\chi_A(y)$ to each variable v_y, circuit C_u outputs 1 if and only if the predicate $\tau(A; u)$ is true. The fact that the restriction $\rho\rho'\rho''$ does not completely determine C_u implies that there exist assignments ρ_0 and ρ_1 such that $C_u\lceil_{\rho\rho'\rho''\rho_0}$ outputs 0 and $C_u\lceil_{\rho\rho'\rho''\rho_1}$ outputs 1. Let $B_0 = \{y \in 0u\Sigma^{kn} \mid \rho\rho'\rho''(v_y) = *,\ \rho_0(v_y) = 1\}$ and $B_1 = \{y \in 0u\Sigma^{kn} \mid \rho\rho'\rho''(v_y) = *,\ \rho_1(v_y) = 1\}$. Then, B_0 and B_1 are disjoint from $A(\alpha-1)$ and $A'(\alpha-1)$ and $\tau(A(\alpha-1) \cup B_0; 1^n) = 0$ and $\tau(A(\alpha-1) \cup B_1; 1^n) = 1$. This proves the claim. □

Next we observe that after stage α, we never add any string of length $\leqq \alpha$ to $A(\alpha)$ or $A'(\alpha)$. From this observation and the fact that the question of $x \in K^k(A)$ does not depend on the strings of length $\geqq |x|$, we see that each stage $\alpha = (k+1)n+1$ satisfies requirement $R_{0,n}$.

Finally, for each i, we observe that eventually we will cancel it in some stage $\alpha=(k+1)n$, since the inequality $2kp_i(n)<t=n^{\log n}$ holds for almost all n. In that stage, we add strings to $A(\alpha)$ or $A'(\alpha)$ (by $\rho\rho'\rho''$) so that when we use $\chi_A(y)$ as the input value for each variable v_y, the circuits C and C_0 are completely determined and circuit C outputs 1 if and only if circuit C_0 outputs 0. By the relation between circuit C and predicate $\sigma_i^{k-1}(A;1^n)$ and the relation between circuit C_0 and the predicate $1^n\in L_k(A)$, we know that $\sigma_i^{k-1}(A;1^n)$ is true if and only if $1^n\notin L_k(A)$. This shows that the requirement $R_{1,i}$ is satisfied by A and $n_i=n$. This completes the proof of Theorem 3.2. □

The above proof can easily be modified to construct an oracle A such that $\mathrm{PSPACE}(A)=\Sigma_k^P(A)\neq\Sigma_{k-1}^P(A)$.

COROLLARY 3.3. *For each $k\geqq 1$, there exists a set A such that* $\mathrm{PSPACE}(A)=\Sigma_k^P(A)\neq\Sigma_{k-1}^P(A)$.

Proof. First consider the cases when $k>1$. The proof is similar to that of Theorem 3.2. All we need to do is to replace the set $K^k(A)$ by the set $Q(A)$, which is $\leqq_m^P$-complete for $\mathrm{PSPACE}(A)$. Furthermore, note that $Q(A)$ has the same property as $K^k(A)$: for any x, the question of whether $x\in Q(A)$ depends only on the set $\{y\in A\,||y|<|x|\}$.

The case $k=1$ can be similarly proved. Baker, Gill, and Solovay [2] have shown that there exists a set A such that $\mathrm{NP}(A)=co\text{-}\mathrm{NP}(A)\neq\mathrm{P}(A)$. To extend it to $\mathrm{PSPACE}(A)=\mathrm{NP}(A)\neq\mathrm{P}(A)$, all we need to do is to replace the set $K(A)$ used in that proof by $Q(A)$. We omit the details. □

4. Relativized hierarchies and PSPACE. In this section, we show that for each $k\geqq 1$, there exists an oracle A such that $\Sigma_k^P(A)=\Pi_k^P(A)\neq\Sigma_{k-1}^P(A)$ and also $\mathrm{PSPACE}(A)\neq\mathrm{PH}(A)$. The proof also uses the lower-bound results on constant depth circuits developed by Yao [12] and Hastad [5]. The following lemma is from [5]. We say a circuit C *computes the parity* of n inputs if C has n variables and for all inputs to those varibables, C outputs 1 if and only if the number of 1's in the input is odd.

LEMMA 4.1 [5]. *There exist an integer n_0' and a real number $\varepsilon>0$ such that for any $k>0$ and any $n>(n_0')^k$, no depth-k circuit C of $\leqq 2^{\varepsilon n^{1/(k-1)}}$ gates can compute the parity of n inputs.*

COROLLARY 4.2. *For any constant c, there is an n_c' such that for all $n>n_c'$, no depth-k, $k=c\log\log n$, circuit C of $\leqq 2^{\varepsilon n^{1/(k-1)}}$ gates can compute the parity of n inputs.*

Proof. Let n_c' be the smallest integer m such that $m>(n_0')^{c\log\log m}$, where n_0' is the absolute constant of Lemma 4.1. □

We first consider the simplest case that the relativized polynomial time hierarchy collapses to the class $\mathrm{P}(A)$.

THEOREM 4.3. *There exists a set A such that* $\mathrm{PSPACE}(A)\neq\mathrm{NP}(A)=\mathrm{P}(A)$.

Proof. Recall that $\{N_i\}$ is an enumeration of all polynomial time nondeterministic oracle TMs, and the machine N_i has its runtime bounded by polynomial p_i. Without loss of generality, we assume that $p_i(n)\leqq n^i$. Recall that $K(A)$ is a set $\leqq_m^P$-complete for $\mathrm{NP}(A)$ such that the question of $x\in K(A)$ depends only on the set $\{y\in A\,||y|\leqq|x|\}$. Let $L_{\mathrm{odd}}(A)=\{1^n\,|$ the number of strings of length n which are in A is odd$\}$. Note that $L_{\mathrm{odd}}(A)$ is in $\mathrm{PSPACE}(A)$.

The construction of A is done by stages. At stage $n=2t$, we want to satisfy the requirement

$R_{0,t}$: for each u of length t, $u\in K(A)$ if and only if $0'u\in A$.

At stage $n=2t+1$, we will try to satisfy the following requirement $R_{1,i}$ with the least integer i for which $R_{1,i}$ is not yet satisfied:

$R_{1,i}$: there exists an m_i such that $1^{m_i}\in L_{\mathrm{odd}}(A)$ if and only if N_i rejects 1^{m_i}.

The requirement $R_0 = \bigwedge_{t=1}^{\infty} R_{0,t}$ states that $K(A)$ is in $\mathrm{P}(A)$, and hence $\mathrm{NP}(A) = \mathrm{P}(A)$. The requirement $R_1 = \bigwedge_{i=1}^{\infty} R_{1,i}$ states that $L_{\mathrm{odd}}(A)$ is not in $\mathrm{NP}(A)$. Therefore a set A satisfying all requirements has the property $\mathrm{PSPACE}(A) \neq \mathrm{NP}(A) = \mathrm{P}(A)$.

In each stage n, we will define, in terms similar to the ones used in the proof of Theorem 3.2, two sets $A(n)$ and $A'(n)$ and an integer β_n. Sets $A(n)$ and $A'(n)$ contain strings reserved by stage n for sets A and $\bar{A}$, respectively. The integer β_n is defined to be an upper bound of the maximum length of the strings added to $A(m)$ or $A'(m)$ in stages $m \leqq n$.

Prior to stage 1, assume that $A(0) = A'(0) = \emptyset$, and let $\beta_0 = 2$. Let all integers i be uncancelled.

Stage $n = 2t$, $t > 0$. We will satisfy requirement $R_{0,t}$. For each string u of length t, we determine whether $u \in K(A(n-1))$. Then, we let $A(n) = A(n-1) \cup \{0^t u\}$ and $A'(n) = A'(n-1)$ if $u \in K(A(n-1))$; and $A(n) = A(n-1)$ and $A'(n) = A'(n-1) \cup \{0^t u\}$ if $u \notin K(A(n-1))$. After this is done for all u of length t, let $\beta_n = \max\{n, \beta_{n-1}\}$.

Stage $n = 2t+1$. Let i be the least integer which has not been cancelled. Let $m = 2i \log n$. If $n \leqq \beta_{n-1}$ or $2^n \leqq n'_{2i}$ or $\varepsilon 2^{n/(m-1)} \leqq 2(m+1)p_i(n)$ (where n'_{2i} is the constant defined in Corollary 4.2), then do nothing. Let $A(n) = A(n-1)$, $A'(n) = A'(n-1)$, and $\beta_n = \beta_{n-1}$.

If $n > \beta_{n-1}$ and $2^n > n'_{2i}$ and $\varepsilon 2^{n/(m-1)} > 2(m+1)p_i(n)$, then consider the machine N_i on input 1^n. From Lemma 2.1, we know that for machine N_i and input 1^n, there is a circuit C of depth 2 such that

(a) the top gate of C is an OR gate with fanin $\leqq 2^{2p_i(n)}$,

(b) the bottom fanin of C is $\leqq p_i(n)$,

(c) the variables in C are those corresponding to strings queried by N_i on input x under some oracle A, and

(d) for each set A, if we use $\chi_A(y)$ as the input value for each variable v_y then C outputs 1 if and only if N_i^A accepts 1^n.

We are going to modify this circuit to a new circuit C' having the following properties. During the modification of C to C', we also define a set B.

(a′) C' has depth $\leqq m = 2i \log n$.

(b′) The number of gates in C' is $\leqq 2^{2(m+1)p_i(n)}$.

(c′) The variables of C' are represented by strings of length n.

(Circuit C' and set B also have nice properties related to the computation of $N_i(1^n)$ and set $L_{\mathrm{odd}}(A)$. We will prove them later.)

Recall that $m = 2i \log n$. The modification of the circuit C is done in $m/2$ steps. Let $C_1 = C$. In each step $j \leqq m/2 - 1$, assume that a circuit C_j is given. For each variable v_y in C_j such that $|y| > n$ and $y = 0^{|u|}u$ for some u, we do the following.

By Lemma 2.2, there is a depth-2 circuit C'_y such that

(a″) the top fanin of C'_y is $\leqq 2^{|u|}$,

(b″) the bottom fanin of C'_y is $\leqq |u|$,

(c″) the variables in C'_y are represented by strings of length $\leqq |u|$, and

(d″) for any set A, if we use $\chi_A(z)$ as the input value for each variable v_z in C'_y then C'_y outputs 1 if and only if $u \in K(A)$.

Replace v_y by the circuit C'_y. (That is, the leaf node of C_j with variable v_y is replaced by the tree C'_y, and the leaf node with the negation $\overline{v_y}$ of variable v_y is replaced by the tree of the dual circuit $\tilde{C}'_y$ of C'_y.) Let C_{j+1} be the new circuit with all such variables v_y in C_j replaced by circuit C'_y.

Assume that a variable v_y is C_j with $y = 0^{|u|}u$ is replaced by C'_y. Then, by Lemma 2.2, each variable v_w in circuit C'_y corresponds to a string w of length $|w| \leqq |u| = |y|/2$.

Note that all strings y such that $v_y \in C_1 = C$ have length $\leqq p_i(n) \leqq n^i$. So, after $\log(p_i(n)) - 1 \leqq i \log n - 1 = m/2 - 1$ steps, none of the variables v_y in $C_{m/2}$ corresponds to a string y of length $|y| > n$ and of the form $0^{|u|}u$ for some u.

In step $m/2$, we replace each variable v_y in $C_{m/2}$ such that $|y| < n$ by the constant $\chi_{A(n-1)}(y)$, and replace each variable v_y in $C_{m/2}$ such that $|y| > n$ by a constant 0 (note that none of these y such that $|y| > n$ is of the form $0^{|u|}u$). Also, each $\overline{v_y}$ is replaced by the opposite value. Let C' be the resulting circuit, and let $B = \{y \,|\, |y| > n, v_y \text{ occurs in } C_{m/2}\}$.

We verify that the final circuit C' satisfies the properties (a')-(c') listed above. First, in each step $j \leqq m/2 - 1$, we replaced some variables in C_j by depth-2 circuits. So, C_{j+1} has depth 2 plus the depth of C_j. Thus, $C_{m/2}$ has depth at most m. Since we only replaced variables in $C_{m/2}$ by constants, the final circuit C' also has depth at most m.

Next, to check (b') we note that every gate in C has fanin $\leqq 2^{2p_i(n)}$. Furthermore, by Lemma 2.2, every gate in circuit C'_y has the same bound for its fanin. Therefore, without combining adjacent gates of the same type, each gate of C' has fanin $\leqq 2^{2p_i(n)}$. That is, the total number of gates in C' is at most $(2^{2p_i(n)})^{m+1} = 2^{2(m+1)p_i(n)}$.

For condition (c'), we note that all variables v_y such that $|y| \neq n$ are replaced by constants or other circuits. So, the only variables left in C' are those v_y with $|y| = n$.

Now from the inequality $\varepsilon 2^{n/(m-1)} > 2(m+1)p_i(n)$, we can apply Corollary 4.2 to circuit C' and conclude that C' does not compute the parity of the 2^n variables v_x, with $x \in \Sigma^n$. So, we can find a set $D \subseteq \Sigma^n$ such that $1^n \in L_{\text{odd}}(D)$ if and only if C' outputs 0 when variables v_z are given values $\chi_D(z)$.

Define $A(n) = A(n-1) \cup D$, $A'(n) = A'(n-1) \cup B$, $\beta_n = \max\{n, p_i(n)+1\}$, and cancel i. Stage $n = 2t+1$ is complete.

Let $A = \bigcup_{n=1}^{\infty} A(n)$. We need to verify that A satisfies every requirement $R_{0,t}$, $t > 0$, and $R_{1,i}$, $i > 0$. For requirement $R_{0,t}$, $t > 0$, we note that in Stage $n = 2t$, we have assigned all strings $0^t u$, $|u| = t$, to $A(n)$ or $A'(n)$ such that $u \in K(A(n-1)) \Leftrightarrow 0^t u \in A(n)$. Since A agrees with $A(n-1)$ on strings of length $< n$, we have $u \in K(A(n-1)) \Leftrightarrow u \in K(A)$. Furthermore, once a string $0^t u$ is added to $A(n)$ or $A'(n)$, its membership in A is never changed in later stages. So, $0^t u \in A(n) \Leftrightarrow 0^t u \in A$. The only thing left to check is that in earlier stages $n' < n$, no string of the form $0^t u$, with $|u| = t$, has been put in $A(n')$ or $A'(n')$. This is true because in an even stage $n' = 2t'$ we never add any string of length longer than n' to $A(n')$ or $A'(n')$, and in an odd stage $n' = 2t'+1$ we only add strings of length n', or strings which are not of the form $0^{|u|}u$ to $A(n')$ or $A'(n')$ (note that none of the strings in B is of the form $0^{|u|}u$).

To verify requirement $R_{1,i}$, we note that the inequality $\varepsilon 2^{n/(m-1)} > 2(m+1)p_i(n)$ is satisfied by almost all integers n. Therefore, each integer i will eventually be cancelled. Assume that i is cancelled in stage $n = 2t+1$. We want to show that $1^n \in L_{\text{odd}}(A)$ if and only if N_i^A rejects 1^n. To show this, we claim that circuit C' constructed in stage n satisfies the following property.

(d') When we use $\chi_A(w)$ as the input value for each variable v_w in C', C' outputs 1 if and only if N_i^A accepts 1^n. (Note that the set A here is the fixed set defined by $A = \bigcup_{n=0}^{\infty} A(n)$ which satisfies requirements $R_{0,t}$ for all $t > 0$.)

In fact, property (d') is satisfied by all circuits C_j, $1 \leqq j \leqq m/2$, constructed in Stage n. We prove it by induction. First, for $j = 1$, we observe that this claim is exactly the property (d) of circuit C.

Now, assume that property (d') is satisfied by C_j, for some $j \leqq m/2 - 1$. To obtain C_{j+1}, assume that some circuit C'_y has replaced a variable v_y in C_j with $y = 0^{|u|}u$. Then, by Lemma 2.2, if we use $\chi_A(z)$ as input value for each variable v_z in C'_y, then C'_y

outputs 1 if and only if $u \in K(A)$. Since A satisfies the property $R_{0,|u|}$ that $u \in K(A)$ if and only if $0^{|u|}u \in A$, C'_y outputs exactly the value $\chi_A(y)$. This means that C_{j+1} outputs the same value as C_j. So, the induction proof is complete.

Finally, in step $m/2$, we replaced all variables v_y by 1 if $y \in A(n-1)$, all variables v_y by 0 if $[|y| \leqq n-1$ and $y \notin A(n-1)]$ or $y \in B$. Since A agrees with these assignments, circuit C' outputs the same value as $C_{m/2}$. This completes the proof of property (d').

Now, observe that, in stage n, the set D is chosen such that $1^n \in L_{\text{odd}}(D)$ if and only if C' outputs 0 when each variable v_z is given the input value $\chi_D(z)$. Since we have added set D to A, the following holds when each variable v_z in C' is given the input value $\chi_A(z)$:

$$1^n \in L_{\text{odd}}(A) \Leftrightarrow 1^n \in L_{\text{odd}}(D) \Leftrightarrow C' \text{ outputs } 0 \Leftrightarrow N_i^A \text{ rejects } 1^n.$$

Thus requirement $R_{1,i}$ is satisfied when i is cancelled. This completes the proof of Theorem 4.3. □

Now we extend Theorem 4.3 to more general cases.

THEOREM 4.4. *For each $k > 0$, there exists a set A such that* PSPACE $(A) \neq \Sigma_k^P(A) = \Pi_k^P(A) \neq \Sigma_{k-1}^P(A)$.

Proof. First, assume that $k > 1$. The proof is a combination of the constructions in the proofs of Theorems 3.2 and 4.3. We only give an outline of the proof. We will construct a set A to satisfy three sets of requirements:

$R_{0,n}$: for all strings u of length n, $u \notin K^k(A) \Leftrightarrow (\exists z_1, |z_1| = n)$

$$(\forall z_2, |z_2| = n) \cdots (Q_k z_k, |z_k| = n) 0 u z_1 z_2 \cdots z_k \in A.$$

$R_{1,i}$: there exists an n_i such that $1^{n_i} \in L_k(A)$ if and only if the ith $\Sigma_{k-1}^{P,1}$-predicate $\sigma_i^{k-1}(A; 1^{n_i})$ is false.

$R_{2,j}$: there exists an m_j such that $1^{m_j} \in L_{\text{odd}}(A)$ if and only if the jth $\Sigma_k^{P,1}$-predicate $\sigma_j^k(A; 1^{m_j})$ is false.

As we argued before, the requirement $R_0 = \bigwedge_{n=1}^{\infty} R_{0,n}$ implies $\Sigma_k^P(A) = \Pi_k^P(A)$ and the requirement $R_1 = \bigwedge_{i=1}^{\infty} R_{1,i}$ implies that $\Sigma_k^P(A) \neq \Sigma_{k-1}^P(A)$. Furthermore, the requirement $R_2 = \bigwedge_{j=1}^{\infty} R_{2,j}$ implies that $L_{\text{odd}}(A) \notin \Sigma_k^P(A)$ and hence PSPACE $(A) \neq \Sigma_k^P(A)$.

In each stage α, we will define, in terms similar to the ones used in the proof of Theorem 3.2, two sets $A(\alpha)$ and $A'(\alpha)$ and an integer β_α. Sets $A(\alpha)$ and $A'(\alpha)$ contain strings reserved by stage α for sets A and $\bar{A}$, respectively. The integer β_α is defined to be an upper bound of the maximum length of the strings added to $A(m)$ or $A'(m)$ in stages $m \leqq \alpha$.

In our construction, we will consider three types of stages. At stage $\alpha = (k+1)n$, we try to satisfy requirement $R_{1,i}$ for the integer i such that $2i$ is the least uncancelled integer. (If the least uncancelled integer is an odd integer, then do nothing.) Assume that $2i$ is the least uncancelled integer and α is sufficiently large (so that $\alpha > \beta_{\alpha-1}$, $n > n_k$, and $2kp_i(n) < n^{\log n}$). Then, we satisfy $R_{1,i}$, with $n_i = n$, by doing almost the same thing as in stage α of the construction in the proof of Theorem 3.2; the only difference is that at the end of the stage we turn on a flag: $F =$ **true**, and cancel the integer $2i$ (instead of i).

At stage $\alpha = (k+1)n+2$, we try to satisfy requirement $R_{2,j}$ for the integer j such that $2j+1$ is the least uncancelled integer. The action in this stage is similar to the construction in an odd stage of the proof of Theorem 4.3.

More precisely, when α is sufficiently large (so that $\alpha > \beta_{\alpha-1}$, $2^\alpha > n'_{2kj}$ and $\varepsilon 2^{\alpha/(m-1)} > 2(m+1)p_j(\alpha)$, where $m = 2k \log p_j(\alpha)$), we consider the jth $\Sigma_k^{P,1}$-predicate $\sigma_j^k(A; 1^\alpha)$. By Lemma 2.1, there is a circuit C associated with $\sigma_j^k(A; 1^\alpha)$ having the following properties:

(a) the depth of C is $\leqq k+1$,

(b) each gate of C has fanin $\leqq 2^{2p_j(\alpha)}$,

(c) the bottom fanin of C is $\leqq p_j(\alpha)$,

(d) the variables in C are represented by strings of length $\leqq p_j(\alpha)$, and

(e) for every set A, if every variable v_y in C is given the value $\chi_A(y)$, then C outputs 1 if and only if $\sigma_j^k(A; 1^\alpha)$ is true.

Using a method similar to the construction in Theorem 4.3, we modify circuit C into C' and define set B to satisfy the following properties:

(a') the depth of C' is $\leqq m = 2k \log p_j(\alpha)$,

(b') the number of gates in C' is $\leqq 2^{2(m+1)p_j(\alpha)}$, and

(c') all variables in C' are represented by strings of length α.

The modification of C into C' is similar to the modification in an odd stage of the construction in Theorem 4.3. The main idea is to replace each variable v_y having $|y| > \alpha$ and $y = 0uz$, for some u and z with $|z| = k|u|$, by the dual circuit $\tilde{C}'_y$ of the circuit C'_y, where C'_y is a depth-$2k$ circuit satisfying conditions of Lemma 2.2 (with respect to the string u). (The reason for using the dual circuit $\tilde{C}'_y$ of C'_y instead of C'_y itself is that we want to get the relation $u \notin K^k(A) \Leftrightarrow y \in A$.) Note that for each y of the form $0uz$, with $|z| = k|u|$, the variables in circuit C'_y correspond to strings of length $\leqq |u| \leqq |y|/(k+1) \leqq |y|/2$. Thus, as argued in the proof of Theorem 4.3, the circuit C will be expanded into a new circuit with depth $\leqq 2k \log p_j(\alpha)$ such that no variable v_y in it is represented by a string y of length $> \alpha$ and of the form $y = 0uz$, $|v| = k|u|$. By replacing all other variables v_w such that $|w| > \alpha$ by constant 0 (and form the set B') and all other variables v_w such that $|w| < \alpha$ by value $\chi_{A(\alpha-1)}(z)$, we obtain a new circuit C' satisfying the above conditions (a')-(c'). Choose a set $D \subseteq \Sigma^\alpha$ such that $1^n \in L_{\text{odd}}(D)$ if and only if C' outputs 0 when each variable v_z is given value $\chi_D(z)$. Let $A(\alpha) = A(\alpha-1) \cup D$, $A'(\alpha) = A'(\alpha-1) \cup B'$, and $\beta_\alpha = \max\{\alpha, p_j(\alpha)+1\}$. Finally, we cancel integer $2j+1$ and turn off the flag: $F =$ **false**.

We can prove, as we did in the proof of Theorem 4.3, that circuit C' satisfies the following property:

(d') Assume that A is an extension of $A(\alpha)$ and $A \cap A'(\alpha) = \varnothing$. Also assume that for all u of length $n < |u| \leqq p_j(\alpha)/(k+1)$, $(\forall z, |z| = k|u|)[u \notin K^k(A) \Leftrightarrow 0uz \in A]$. Then, if we give $\chi_A(w)$ as input value for each variable v_w in C', then C' outputs 1 if and only if $\sigma_j^k(A; 1^\alpha)$ is true.

At stage $\alpha = (k+1)n+1$, we satisfy requirement $R_{0,n}$. For each u of length n, we determine whether $u \in K^k(A(\alpha-1))$, and try to find a set $B \subseteq 0u\Sigma^{kn}$ such that

$$(*) \quad u \notin K^k(A(\alpha-1)) \Leftrightarrow (\exists z_1, |z_1| = n) \cdots (Q_k z_k, |z_k| = n) 0uz_1 \cdots z_k \in A(\alpha-1) \cup B.$$

This set B will be determined as follows: if the flag F is true, then search for a set B satisfying both $(*)$ and $B \cap (A(\alpha-1) \cup A'(\alpha-1)) = \varnothing$; if the flag is false, then let $B = \varnothing$ when $u \in K^k(A(\alpha-1))$ and $B = 0u\Sigma^{kn}$ when $u \notin K^k(A(\alpha-1))$. Finally, let $A(\alpha) = A(\alpha-1) \cup B$ and $A'(\alpha) = A'(\alpha-1)$.

Note that the flag F is turned *on* whenever an even integer $2i$ is cancelled. When $2i$ is cancelled in stage α', some strings of the form $0uz$, $|u| = n$ and $|z| = kn$, may have been added to set $A(\alpha')$ or $A'(\alpha')$. However, later in stage $\alpha = (k+1)n+1$, before the flag F is turned off, a set B satisfying both $(*)$ and $B \cap (A(\alpha-1) \cup A'(\alpha-1)) = \varnothing$ can always be found, as proved by Lemma 3.1. (Note that by setting $\beta_{\alpha'}$ to be an upper bound of the maximum length of strings added to $A(\alpha')$ or $A'(\alpha')$, we know that the flag F will not be turned off until in some stage $\alpha > \beta_{\alpha'}$.)

The flag F will be turned *off* when we cancel an odd integer $2j+1$. Suppose we cancel $2j+1$ at stage α'', then we must have $\alpha'' > \beta_{\alpha''-1}$, and hence no string of length

$\geqq \alpha''$ is in set $A(\alpha''-1)$ or in set $A'(\alpha''-1)$. Thus, in a later stage $\alpha=(k+1)n+1$, before the flag is turned on, we have $0u\Sigma^{kn} \cap (A(\alpha-1) \cup A'(\alpha-1))=\varnothing$ for all u of length n. So, in stage α, the choice of the set B can be made free from interference of earlier stages.

This completes the construction of set A. Note that by setting β_α be an upper bound of the maximum length of the strings added to $A(m)$ or $A'(m)$ for all $m \leqq \alpha$, we prevent the possible interference between stages $(k+1)n$ and stages $(k+1)n+2$. By the discussions in the construction, set A satisfies requirements $R_{0,n}$ for all n and $R_{1,i}$ for all i. The only thing left to check for requirement $R_{2,j}$ is that after we cancel $2j+1$ in stage $\alpha=(k+1)n+2$, we construct A in later stages such that, for all u of length $n<|u| \leqq p_j(\alpha)/(k+1)$, $(\forall z, |z|=k|u|)[u \notin K^k(A) \Leftrightarrow 0uz \in A]$. This is stronger than the requirement $R_{0,|u|}$. However, we note that by the definition of β_α the flag F is off at stage $\alpha'=(k+1)|u|+1$. Therefore, in stage α', we satisfy the requirement $R_{0,|u|}$ by letting $0u\Sigma^{k|u|} \subseteq A$ if $u \notin K^k(A)$ and $0u\Sigma^{k|u|} \cap A=\varnothing$ if $u \in K^k(A)$. This shows that the assumption, and hence the conclusion, of property (d′) of stage α is satisfied by A. As a consequence, requirement $R_{2,j}$ is satisfied when $2j+1$ is cancelled. This completes the proof of the cases $k>1$.

The proof of the case $k=1$ is almost identical to the general case $k>1$, except that the requirement $R_{1,i}$ is actually easier to satisfy. We may simply use Baker, Gill, and Solovay's [2] original proof for the result $(\exists A)[\mathrm{NP}(A)=co\text{-}\mathrm{NP}(A) \neq \mathrm{P}(A)]$. We omit the details. □

5. Open questions. In the last two sections, we have constructed oracles which collapse the polynomial time hierarchy to exactly the kth level. Furthermore, relative to different oracles, the class PSPACE may either collapse to the kth level of the polynomial time hierarchy or may be different from the polynomial time hierarchy. Several questions about the relativized polynomial time hierarchy, however, remain open. First, note that the set $L_{\mathrm{odd}}(A)$ is actually in the class $D\#\mathrm{P}(A)$ (see, for definition, Angluin [1]). Thus, our results together with Yao's [12] result actually showed that relative to some oracles, the class $D\#\mathrm{P}$ may be separated from the polynomial time hierarchy while the hierarchy may have either finite or infinite levels. An interesting question here is to find an oracle to separate the class PSPACE from $D\#\mathrm{P}$.

Heller [7] has constructed oracles X and Y such that $\Sigma_2^P(X)=\Pi_2^P(X) \neq \Delta_2^P(X)$ and $\Sigma_2^P(Y)=\Delta_2^P(Y) \neq \Sigma_1^P(Y)$. It would be interesting to see whether these results could be extended to the kth level of the polynomial time hierarchy.

Appendix. In this appendix, we give a proof for Lemma 3.1. The proof will be done by induction on k. The induction proof is easier on the following stronger form of Lemma 3.1.

LEMMA 3.1 (stronger form). *For every $k \geqq 2$ there exists a constant n_k such that the following holds for all $n>n_k$. Let $t=n^{\log n}$, $m<2^t$, and $C_0, C_1, \cdots, C_m$ be $m+1$ circuits each defining a $f_k^{2^n}$ function, with their variables pairwisely disjoint. Let C be a depth-k circuit such that the bottom fanin is $\leqq t$ and the number of gates in C of distance at least 2 from the leaves is $\leqq 2^t$. Then, there exists a restriction ρ on C such that ρ completely determines C but it does not completely determine any C_i, $0 \leqq i \leqq m$.*

First we need some definitions from Hastad [5]. Let V be a set of variables, and $\mathcal{B}=\{B_j\}_{j=1}^r$ a partition of V. Let q be a real number, $0<q<1$. Define $R_{q,\mathcal{B}}^+$ to be the probability space of restrictions which take values as follows. To define a random restriction ρ in $R_{q,\mathcal{B}}^+$, first, for each B_j, $1 \leqq j \leqq r$, let $s_j=*$ with probability q and $s_j=0$ with probability $1-q$; and then, independently, for each variable $x \in B_j$, let $\rho(x)=s_j$ with probability q and $\rho(x)=1$ with probability $1-q$. Similarly, a $R_{q,\mathcal{B}}^-$ probability

space of restrictions is defined by interchanging the roles played by 0 and 1. Furthermore, define for each $\rho \in R^+_{q,\mathcal{B}}$ a restriction $g(\rho)$: for all B_j with $s_j = *$, let V_j be the set of all variables in B_j which are given value $*$ by ρ; $g(\rho)$ selects one variable y in V_j and gives value $*$ to y and value 1 to all others in V_j. For $\rho \in R^-_{q,\mathcal{B}}$, $g(\rho)$ is similarly defined by interchanging the roles of 0 and 1.

Now, for given circuits C_i, $0 \leqq i \leqq m$, let $\mathcal{B} = \{B_j\}$ be the partition of all variables that occurred in C_i, $0 \leqq i \leqq m$, such that each B_j is the set of all variables leading to a bottom gate in some C_i, $0 \leqq i \leqq m$ (when k is even, the bottom gates of C_i's are AND gates). Also let $t = n^{\log n}$ and $q = 1/(24t)$. We will prove two lemmas which state that with a high probability, a random restriction ρ from $R^+_{q,\mathcal{B}}$ (or, from $R^-_{q,\mathcal{B}}$) has the properties that $C\lceil_{\rho g(\rho)}$ is equivalent to a depth-$(k-1)$ circuit with bottom fanin $\leqq t$ and that each $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, contains a subcircuit computing a $f^{2^n}_{k-1}$ function. Thus an induction can be used.

In the following, the first lemma is exactly Lemma 6.3 of Hastad [5], and the second lemma is a stronger form of Lemma 6.8 of [5]. The main difference is that we use a smaller bound t for the bottom fanin, and hence a bigger probability q of assigning $*$ to variables, so that the probability of a circuit $C_i\lceil_{\rho g(\rho)}$ having a subcircuit computing a $f^{2^n}_{k-1}$ function is bigger.

LEMMA A.1. *Let s, t be integers and q a real number, $0 < q < 1$. Let G be an* AND *of* ORs *with bottom fanin $\leqq t$, and $\mathcal{B} = \{B_j\}$ be a partition of variables in G. Then, for a random restriction ρ from $R^+_{q,\mathcal{B}}$, the probability that $G\lceil_{\rho g(\rho)}$ is not equivalent to a circuit of* OR *of* ANDs *with bottom fanin $\leqq s$ is bounded by α^s, where $\alpha < 6qt$.*

Lemma A.1 also holds with $R^+_{q,\mathcal{B}}$ replaced by $R^-_{q,\mathcal{B}}$, or with G being an OR of ANDs to be converted to a circuit of an AND of ORs.

Proof. See Hastad [5]. □

LEMMA A.2. *For each $k > 2$, there exists an integer n_k such that the following holds for all $n > n_k$. If k is even and ρ is a random restriction from $R^+_{q,\mathcal{B}}$, then the probability that every circuit $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, contains a subcircuit computing an $f^{2^n}_{k-1}$ function is $\geqq \frac{2}{3}$. If k is odd, then the same probability holds for a random restriction ρ from $R^-_{q,\mathcal{B}}$.*

Proof. Assume k is even and ρ is a random restriction from $R^+_{q,\mathcal{B}}$. Note that the subcircuits of the two lower levels of a $C^{2^n}_k$ circuit are ORs of ANDs with bottom fanin exactly $2^{n/2}$. There are $2^{(k-5/2)n}$ such depth-2 subcircuits in $C^{2^n}_k$.

(i) We first show that with a high probability all bottom AND gates $H_j\lceil_{\rho g(\rho)}$ (corresponding to block B_j) of all circuits $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, take the value s_j (the value assigned to block B_j by ρ). Note that each $H_j\lceil_{\rho g(\rho)}$ has $2^{n/2}$ inputs. Also note that $H_j\lceil_{\rho g(\rho)}$ has value $\neq s_j$ if and only if all inputs to $H_j\lceil_{\rho g(\rho)}$ are 1. Therefore, we have

$$\begin{aligned}\Pr[H_j\lceil_{\rho g(\rho)} \text{ has value} \neq s_j] &= \Pr[\text{all inputs to } H_j\lceil_{\rho g(\rho)} \text{ are } 1] \\ &= (1-q)^{2^{n/2}} = \left(\left(1 - \frac{1}{24n^{\log n}}\right)^{24n^{\log n}}\right)^{2^{n/2}/(24n^{\log n})} \\ &< e^{-2[(n/2-5(\log n)^2]} < 2^{-2^{n/4}},\end{aligned}$$

if $n > 20(\log n)^2$. In the above, the first inequality comes from the observation that $(1-1/n)^n < e^{-1}$ for all n.

Note that there are exactly $m+1$ many circuits C_i, and each has exactly $2^{(k-3/2)n}$ many bottom AND gates. So the probability that all bottom AND gates $H_j\lceil_{\rho g(\rho)}$ of all circuits $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, take the value s_j is

$$\geqq 1 - 2^{-2^{n/4}} \cdot 2^{(k-3/2)n} \cdot (m+1) > 1 - 2^{-2^{n/8}} > \tfrac{5}{6},$$

if $2^{n/8} > n^{\log n} + kn$ and $n \geqq 16$.

(ii) Next we show that with a high probability all OR gates at level 2 of circuits $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, have at least $2^{n/2}$ many child nodes $H_j\lceil_{\rho g(\rho)}$ of AND gates having values $s_j = *$. Let G be an OR gate at level 2 of some C_i, $0 \leqq i \leqq m$. Note that $G\lceil_{\rho g(\rho)}$ has 2^n many children $H_j\lceil_{\rho g(\rho)}$, each having probability q of having the value $s_j = *$. Let p_l be the probability that $G\lceil_{\rho g(\rho)}$ has exactly l many AND gates $H_j\lceil_{\rho g(\rho)}$ having values $s_j = *$. Then,

$$p_l = \binom{2^n}{l} q^l (1-q)^{2^n - l}.$$

Note that when $l \leqq 2^{n/2+1}$, we have $p_l \leqq \frac{1}{2}$ and

$$\frac{p_l}{p_{l-1}} = \frac{2^n - l + 1}{l} \cdot \frac{q}{1-q} \geqq (2^{n/2-1} - 1)q \geqq 2,$$

if $2^{n/4} > 10n^{\log n}$. So, we have

$$\sum_{l=0}^{2^{n/2}} p_l \leqq p_{2^{n/2}} \cdot \sum_{l=0}^{2^{n/2}} 2^{-l} \leqq 2 \cdot p_{2^{n/2}} \leqq 2 \cdot 2^{-2^{n/2}} \cdot p_{2^{n/2+1}} \leqq 2^{-2^{n/2}}.$$

Thus the probability that all OR gates at level 2 of circuits $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, have at least $2^{n/2}$ many child nodes H_j of AND gates having value $s_j = *$ is

$$\geqq 1 - 2^{-2^{n/2}} \cdot 2^{(k-5/2)n} \cdot (m+1) > 1 - 2^{-2^{n/4}} > \frac{5}{6},$$

if $2^{n/4} > n^{\log n} + kn$ and $n \geqq 8$.

Combining results (i) and (ii) and letting n_k be the smallest integer which satisfies all the inequalities of the "if" clauses above, we have proved Lemma A.2 for even k. The case for odd k is symmetric. □

Now we are ready to prove the stronger form of Lemma 3.1. Note that the integer n_k chosen in Lemma A.2 is so large that $2^{n/4} > n^{\log n} + kn$. First let $k = 2$. Assume that C is an OR of ANDs which does not compute a constant function 0. Then we define a restriction ρ that maps all variables which are children of the first AND gate of C to 1, and maps all variables whose negations are the children of the first AND gate of C to 0, and all other variables to $*$. This makes C computing a constant function 1. However, this restriction ρ only assigns $\leqq t = n^{\log n}$ many variables to 0 or 1. Since each circuit C_i, $0 \leqq i \leqq m$, has both the bottom fanin and the top fanin $2^{n/2} > t$, it has the following properties:

(a) For each AND gate of C_i, there is at least one variable assigned $*$ by ρ.

(b) There is at least one AND gate of C_i having all variables assigned $*$ by ρ.

From these two properties, we know that $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, does not compute a constant function.

The argument for a depth-2 circuit C which is an AND of ORs is similar.

For the inductive step, let $k > 2$. Recall that $q = 1/(24t)$. Assume, without loss of generality, that the bottom gates of C are OR gates. Then, by Lemma A.1, for any random restriction ρ from $R^+_{q,\mathcal{B}}$ or from $R^-_{q,\mathcal{B}}$, the probability that any single subcircuit G of the two lower levels of $C\lceil_{\rho g(\rho)}$ is not equivalent to an OR of ANDs with bottom fanin $\leqq t$ is at most α^t. Since there are at most 2^t such subcircuits, the probability that at least one such subcircuit is not equivalent to an OR of ANDs with bottom fanin $\leqq t$ is at most $(2\alpha)^t < (12qt)^t = 2^{-t}$. Thus, with probability at least $\frac{2}{3}$, the circuit $C\lceil_{\rho g(\rho)}$ can be written as a depth-$(k-1)$ circuit with the bottom fanin $\leqq t$. Furthermore, the number of gates in $C\lceil_{\rho g(\rho)}$ of distance at least 2 from the leaves is exactly the number of gates in C of distance at least 3 from the leaves and is $\leqq 2^t$.

Assume that k is even, then we choose a restriction ρ from $R^{+}_{q,\mathcal{B}}$ such that (a) $C\lceil_{\rho g(\rho)}$ is equivalent to a depth-$(k-1)$ circuit with the bottom fanin $\leqq t$ and with $\leqq 2^{t}$ gates of distance $\geqq 2$ from leaves, and (b) every $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$, has a subcircuit computing an $f^{2^n}_{k-1}$ function. (By Lemma A.2 and the above argument, there are at least $\frac{1}{3}$ of restrictions from $R^{+}_{q,\mathcal{B}}$ having these properties.) Similarly, if k is odd, we can choose a restriction ρ from $R^{-}_{q,\mathcal{B}}$ which has these properties. By the inductive hypothesis, we can find a restriction ρ' such that ρ' completely determines the circuit $C\lceil_{\rho g(\rho)}$ but none of $C_i\lceil_{\rho g(\rho)}$, $0 \leqq i \leqq m$. The combined restriction $\rho g(\rho)\rho'$ satisfies our requirement, and Lemma 3.1 is proven. □

REFERENCES

[1] D. ANGLUIN, *On counting problems and the polynomial-time hierarchy*, Theoret. Comput. Sci., 12 (1980), pp. 161-173.

[2] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the* P = ?NP *question*, SIAM J. Comput., 4 (1975), pp. 431-442.

[3] T. BAKER AND A. SELMAN, *A second step toward the polynomial hierarchy*, Theoret. Comput. Sci., 8 (1979), pp. 177-187.

[4] M. FURST, J. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13-27.

[5] J. T. HASTAD, *Computational limitations for small-depth circuits*, Ph.D. thesis, Massachusetts Institute of Technology, MIT Press, Cambridge, 1987.

[6] ———, *Almost optimal lower bounds for small depth circuits*, Proc. 18th ACM Symposium on Theory of Computing, (1986), pp. 6-20.

[7] H. HELLER, *Relativized polynomial hierarchies extending two levels*, Math. Systems Theory, 17 (1984), pp. 71-84.

[8] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[9] M. SIPSER, *Borel sets and circuit complexity*, Proc. 15th ACM Symposium on Theory of Computing (1983), pp. 61-69.

[10] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.

[11] C. WRATHALL, *Complete sets and the polynomial hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23-33.

[12] A. YAO, *Separating the polynomial-time hierarchy by oracles*, Proc. 26th IEEE Symp. on Foundations of Computer Science (1985), 1-10.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 409–418, April 1989

016

EXPONENTIAL AVERAGE TIME FOR THE PURE LITERAL RULE*

KHALED M. BUGRARA†, YOUFANG PAN‡, AND PAUL WALTON PURDOM, JR. §

Abstract. This paper gives exponential lower bounds for the average time of an algorithm based on a simplified version of the pure literal rule from the Davis–Putnam procedure. It is shown that this algorithm requires an average time that is exponential in v, the number of variables, when p, the probability that a literal is in a clause, is proportional to $1/v$ and t, the number of clauses in the predicate, is at least a linear function of v. The time is greater than any polynomial in v over a somewhat larger range of parameters. For the two cases $t = \Theta(\ln v)$ and $p = O(\ln v/v^{3/2})$, the results of this lower-bound analysis are the same (to within a constant factor) as the upper-bound results in [SIAM J. Comput., 14 (1985), pp. 943–953]. The results of this and other papers show that the fastest analyzed algorithm for random satisfiability problems depends on the parameters of the distribution. Backtracking, the pure literal rule, and the new algorithm of Iwama [Report KSU/ICS 88-01, Institute of Computer Science, Kayoto Sangyo University, Kyoto, Japan, 1988] each have a large region of parameter setting where it is exponentially faster than the other two.

Key words. average time, backtracking, combinatorial search, Davis–Putnam, pure literal rule, NP-complete, satisfiability, searching

AMS(MOS) subject classifications. 68P10, 68Q20, 68Q25, 68T15

1. Introduction.

Many of the ideas that are used in practice for solving satisfiability problems are related to the Davis–Putnam procedure: the problem instance is solved directly if it has no clauses (in which case the problem is satisfiable) or if it has a clause of length zero (in which case it is unsatisfiable). Otherwise, a variable is selected and two subproblems are generated: one with the variable set to *true* and one with it set to *false*. The original problem has no solutions if and only if both subproblems have no solutions. The subproblems are simplified by removing all clauses with true literals (because they are satisfied with the present partial assignment of values to variables) and by removing all false literals from clauses (because a false literal cannot contribute to making the clause true). The subproblems are solved by recursive use of the algorithm.

The Davis–Putnam procedure uses four additional techniques that often speed up the algorithm: 1) the unit clause rule, 2) the pure literal rule, 3) subsumption, and 4) a dynamic order of assigning values to variables. The unit clause rule says that a variable that appears in a clause with only one literal must be set so as to make that literal true. The pure literal rule says that a variable that appears only in positive literals or only in negative literals needs to be set only in the way that makes its literal true. The pure literal rule does not find all the solutions to a problem, but it does find a solution if one exists. Subsumption removes each clause whose literals are a superset of the literals of some other clause. The search order is dynamic because whenever possible the Davis–Putnam procedure sets a variable that generates only one subinstance (a variable associated with a unit clause or a pure literal).

For several models of random satisfiability (SAT) problems, it is well known that the Davis–Putnam procedure can solve some random sets of SAT problems in polyno-

* Received by the editors November 25, 1987; accepted for publication July 21, 1988.
† College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, Massachusetts 02115.
‡ Computer Science Department, Indiana University, Lindley Hall 315C, Bloomington, Indiana 47405–4101.
§ Computer Science Department, Indiana University, Lindley Hall 101, Bloomington, Indiana 47405–4101.

mial average time. The first result was that of Goldberg [4]. He showed that a variant of the Davis–Putnam procedure that uses only the pure literal rule and splitting requires polynomial average time for one probability distribution. His *Pure Literal Rule Algorithm* uses a fixed order for assigning values to variables. The main feature of the algorithm is that only one subproblem is generated when the splitting variable is a pure literal.

Goldberg's analysis was later extended in [5]. Additional analysis of the Pure Literal Rule Algorithm was done in [9], where it was shown that the algorithm could solve a much larger class of problems in polynomial average time. This behavior is a direct result of two factors: the ability of the pure literal rule to recognize variables that are relevant to the solution, and the large (exponential) number of solutions associated with these problem sets. The previous upper-bound studies show that the pure literal rule plus splitting is more efficient than backtracking [8], [11] for some random sets of problems. (The version of backtracking that was analyzed was required to find all solutions.) The lower-bound results in [2] showed that there are some random sets of problems where backtracking is more efficient. The lower-bound results of this paper show that there is a large region in parameter space where the pure literal rule is known to be slower than backtracking, and they show that there is a large region where neither the pure literal rule nor backtracking runs in polynomial average time.

So far it has been too difficult to analyze the complete Davis–Putnam procedure. Instead, investigators have simplified the algorithm in various ways that reduce its performance. When the simplified algorithm is fast, the original one is also fast. When the simplified algorithm is slow, the complete one may or may not be slow. One might suspect, however, that the Davis–Putnam procedure is slow in that region where neither backtracking nor the pure literal rule is fast.

Figure 1 is a diagram of the parameter space (p, the probability that a literal is in a clause, and t, the number of clauses, as functions of v, the number of variables) that shows where both backtracking and the Pure Literal Rule Algorithms take large amounts of time to solve random conjunctive normal form SAT problems. The only points that are significant are those where lines intersect. The axes are labeled with various functions of v. The ϵ on the diagram is a positive constant that can be as small as one wishes. (In the text, the results that use ϵ are expressed with the more formal o and ω notations.) Constants have been suppressed in these functions (except for $\ln 2/v$ on the p axis, and v on the t axis). For example, the points marked with $p = (\ln v)^2/(\epsilon v^{3/2})$ and $t = v/\ln v$ have a much larger constant associated with the formula for t than has the point marked with $p = (1/\epsilon)[(\ln v)/v]^{3/2}$. Problems in the lower right region (small p and large t) can be solved in polynomial average time by backtracking [11]. For these problems the average number of solutions per problem is exponentially small. Problems in the lower left region (small p and small t), in the left region (small t), and in the upper region (large p) can be solved in polynomial average time by the Pure Literal Rule Algorithm [9]. The analyses for the superpolynomial region (the region where the average running time grows faster than any polynomial in v) and the exponential region are from this paper. The left boundary of the exponential region ($t = \epsilon v$) is definite; problem sets to the left of it require superpolynomial, but nonexponential average time. The upper boundary is not definite. Problem sets below the boundary require exponential time, while those above the boundary definitely require superpolynomial average time and might require exponential average time. In the two regions marked with question marks it is

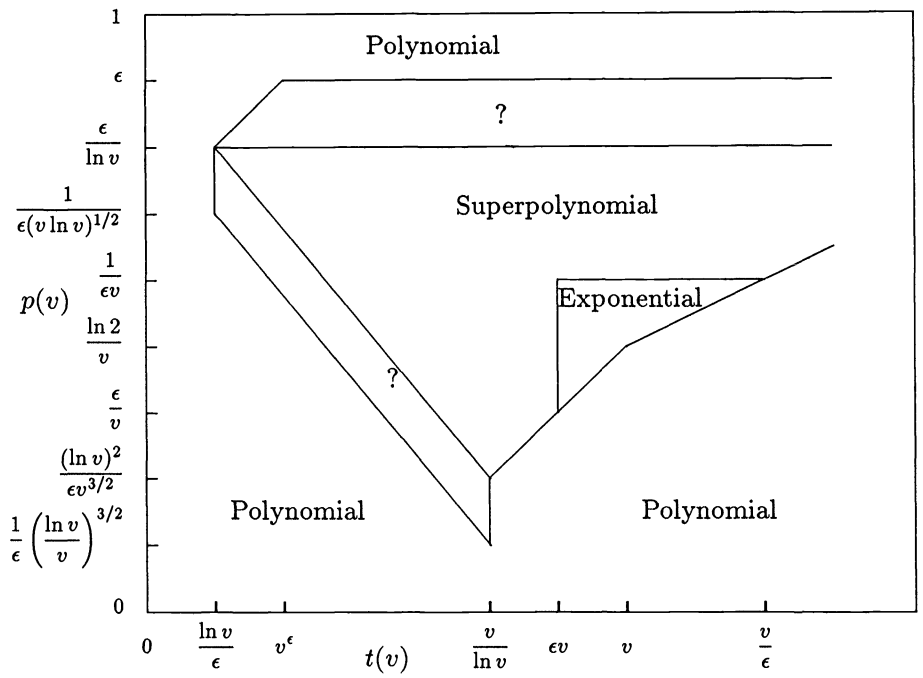


FIG. 1. *The regions where both backtracking and the pure literal rule need polynomial, superpolynomial, and exponential time are shown.*

not known whether the time for the pure literal rule is polynomial or superpolynomial. Backtracking takes exponential average time in all of the regions marked exponential, superpolynomial, or unknown. In most of these regions it is also the case that the average number of solutions per problem is exponentially large [9], but there is a region to the right of the $t = v$ line where the average number of solutions per problem is exponentially small (see the diagram in [9]).

Recently, Iwama [6] analyzed a SAT algorithm that is based on counting solutions. The average time of his algorithm is polynomial when $p > \sqrt{(\ln t)/v}$. Thus, Iwama's algorithm is better than the pure literal rule and backtracking when p is not small. The pure literal rule is better than (the upper limit analysis of) Iwama's algorithm when p is small, and it is better than backtracking when t is not too large. Finally, backtracking is better when t is large and p is small. In practice, these three techniques should be combined into a single algorithm, which would be fast where any one of the three is fast.

2. Random problems. We compute the average time required to find solutions to a predicate with v variables, t clauses, and with probability p that a literal is in a clause. The predicates are in *conjunctive normal form.* A random clause is formed by including each literal with probability p. A predicate consists of t independently selected random clauses. Since there are $2v$ literals altogether, and since each literal is included with probability p, the expected clause size is equal to $2vp$. The expected clause size is linear in v whenever p is fixed; it is fixed when p varies inversely with v, i.e., $p = \Theta(v^{-1})$. (The asymptotic notations Θ, O, Ω, o, and ω are given in [7].)

The Pure Literal Rule Algorithm generates a search tree, giving values to the variables in a fixed order. At each step it is working with a simplified predicate where

each clause with a *true* literal is eliminated and each *false* literal has been removed from all of its clauses. The simplified predicate contains only those literals whose variables do not yet have a value. If the simplified predicate contains the current variable in both positive and negative literals, then two subproblems are generated, one with the variable set to *true* and one with it set to *false*. If the variable only occurs positively, then only the subproblem with the variable set to *true* is generated; if the variable only occurs negatively, then only the subproblem with the variable set to *false* is generated. If the variable does not occur at all, then one subproblem identical to the current predicate is generated.

The average time for the Pure Literal Rule Algorithm is given by [4], [5]:

$$(1) \quad A(t,v) = tv + (1-p)^{2t}A(t,v-1) + 2\sum_{j\geqq 1}\binom{t}{j}p^j(1-p)^{t-j}A(t-j,v-1)$$

with boundary conditions $A(0,v) = A(t,0) = 0$. This implies that $A(1,v) \geqq v$ for $v \geqq 1$ and $A(t,1) = t$ for $t \geqq 1$. The first term in the recurrence corresponds to the time of one step of the recursion; the second term corresponds to the case where the selected variable does not occur in any clause of the predicate; the sum is the weighted sum of those cases where a literal that corresponds to the selected variable occurs j times in the predicate. Although there is no obvious way to solve the full history recurrence, the behavior of the solution can be determined from upper- and lower-bound methods.

3. Lower bound. Since the algorithm considers variables in fixed order, adding a clause to the predicate can only increase the size of the pure literal rule search tree (or leave it unchanged). Notice that the algorithm only looks for pure literals. If a predicate includes the pair of clauses $x \wedge \bar{x}$, then x will not be selected because x is not a pure literal. Thus, $A(t,v)$ is an increasing function of t. Also, it is an increasing function of v [9]. Therefore,

$$A(t,v) \geqq \max\{t,v\} \quad \text{for } t \geqq 1,\ v \geqq 1.$$

Since each term in the right-hand side of (1) is positive, we obtain a lower bound by taking selected terms (the tv term is dropped, and only the first n terms of the sum are retained):

$$A(t,v) \geqq (1-p)^{2t}A(t,v-1) + 2\sum_{1\leqq j\leqq n}\binom{t}{j}p^j(1-p)^{t-j}A(t-j,v-1).$$

This equation is still too complex to solve.

Since $A(t,v)$ is an increasing function of t, we replace $A(t-j,v)$ by $A(t-n,v)$ to obtain an equation that is easier to solve. This gives the lower bound

$$A(t,v) \quge A_n(t,v) \quad \text{for } t \geqq t_0,\ v \geqq v_0,$$

where

$$(2) \quad A_n(t,v) = \left[(1-p)^{2t} + 2\sum_{1\leqq j\leqq n}\binom{t}{j}p^j(1-p)^{t-j}\right]A_n(t-n,v-1),$$

and with boundary conditions such that

$$A_n(t_0, v) \leqq A(t, v) \quad \text{for } t \geqq t_0 \quad \text{and} \quad A_n(t, v_0) \leqq A(t, v) \quad \text{for } v \geqq v_0.$$

This equation is first-order linear in A_n. Let a be some number such that $a \leqq \min\{v-1, (t-1)/n\}$ (a is the depth of the recursion). Choose $t_0 = t - an$, $v_0 = v - a$, and $A_n(t_0, v) = A_n(t, v_0) = 1$. By iterating for a steps, the solution

$$A_n(t, v) = \prod_{0 \leqq k < a} \left[(1-p)^{2(t-nk)} + 2 \sum_{1 \leqq j \leqq n} \binom{t-nk}{j} p^j (1-p)^{t-nk-j} \right] \tag{3}$$

is obtained (see [10]). The function $A_n(t, v)$ depends on a as well as on n, t, and v, but we do not show the dependence on a because we intend to set a to obtain a large value for $A_n(t, v)$.

4. Asymptotic analysis. This section gives derivations of the asymptotic behavior of the bound given by (3), determining conditions where the average time is a rapidly increasing function of v. We consider p and t to be functions of v, but to avoid notational clutter, we call the functions p and t rather than $p(v)$ and $t(v)$. We determine conditions for which $A(t, v)$ is superpolynomial, i.e., greater than v^n for large but fixed n, and conditions for which $A(t, v)$ is exponential, i.e., greater than $(1+\epsilon)^v$ for small positive ϵ. Since $A(t, v)$ is polynomial if p is a fixed constant as v goes to infinity [9], we restrict the analysis to the case where p approaches zero.

4.1. Approximations. We will show that when pt is small enough, setting $n = 2$ includes enough terms to obtain an exponential lower bound (setting $n = 1$ does not lead to an exponential lower bound). When n is equal to 2, a lower bound is given by

$$A_2(t, v) = \prod_{0 \leqq k < a} \Big[(1-p)^{2(t-2k)} + 2(t-2k)p(1-p)^{t-2k-1} + (t-2k)(t-2k-1)p^2(1-p)^{t-2k-2} \Big], \tag{4}$$

where $a \leqq \min\{v-1, (t-1)/2\}$.

We show that $A_2(t, v)$ is large for some small pt by first showing that the factor in brackets is larger than 1 plus a small quantity, $f(v)$, over a range of k, $0 \leqq k \leqq g(v)$. The functions f and g must be in the ranges $0 \leqq f(v) \leqq 1$ and $0 \leqq g(v) < v$. That is, for the various values of $g(v)$ and $f(v)$, we want to show that

$$A_2(t, v) \geqq [1 + f(v)]^{g(v)} = v^{\Theta(f(v)g(v)/\ln v)},$$

so the bound is exponential [$(1+\epsilon)^v$ for some $\epsilon > 0$] if and only if

$$f(v) = \Theta(1) \quad \text{and} \quad g(v) = \Theta(v). \tag{5}$$

The function $A_2(t, v)$ is polynomial if $f(v)g(v) = O(\ln v)$. This can occur over a range of functions. For $g(v) = \Theta(v)$ [the largest value for $g(v)$], polynomial time results if and only if $f(v) = O([\ln v]/v)$. For $f(v) = \Theta(1)$ [the largest value for $f(v)$] polynomial time results if and only if $g(v) = O(\ln v)$. In what follows, let $N(v)$ be an increasing function that goes to infinity very slowly. So, $N(v) = \omega(1) = 1/o(1)$, but it is not too large. If

$$f(v)g(v) = N(v) \ln v, \tag{6}$$

then the bound is superpolynomial (larger than any polynomial).

We now find the conditions on p and t that lead to large running times. First concentrate on the size of the factor in square brackets from (4): replacing $t - 2k$ by τ gives

$$(1-p)^{2\tau} + 2p\tau(1-p)^{\tau-1} + p^2\tau(\tau-1)(1-p)^{\tau-2}. \tag{7}$$

Each term in (7) can now be expanded with the binomial theorem. Assume that there is an upper-bound on $p\tau$ and that τ goes to infinity as v goes to infinity. Then the terms with a power of p larger than the power of τ go to zero, and the following expansions are obtained:

$$\begin{aligned}(1-p)^{2\tau} &= 1 - 2p\tau + 2p^2\tau^2 - \frac{4}{3}p^3\tau^3 + \Theta(p^4\tau^4),\\ 2p\tau(1-p)^{\tau-1} &= 2p\tau - 2p^2\tau^2 + p^3\tau^3 - \Theta(p^4\tau^4),\\ p^2\tau(\tau-1)(1-p)^{\tau-2} &= p^2\tau^2 - p^3\tau^3 + \Theta(p^4\tau^4).\end{aligned}$$

Adding the terms gives

$$(1-p)^{2\tau} + 2p\tau(1-p)^{\tau-1} + p^2\tau(\tau-1)(1-p)^{\tau-2} = 1 + p^2\tau^2 - \Theta(p^3\tau^3).$$

Therefore, the factor in square brackets from (4) is larger than $1 + f(v)$ when

$$p^2\tau^2 - \Theta(p^3\tau^3) > f(v).$$

If C is the larger constant implied by the Θ so that $\Theta(p^3\tau^3) \leqq Cp^3\tau^3$, then this equation has a solution for $\sqrt{(2f(v))} < p\tau < 1/(2C)$. In particular, for $f(v) < 1/(128C^2)$ there exist solutions for a range of $p\tau$ where the upper-bound is at least four times the lower bound. So, if $f(v)$ is below the appropriate constant, we can find the lower bound on $p\tau$ by factoring out the $p^2\tau^2$ term from the left-hand side and taking the square root to obtain

$$p\tau > \sqrt{f(v)}[1 - Cp\tau]^{-1/2}.$$

Taylor's theorem with a remainder gives $(1-x)^{-1/2} = 1 + \frac{1}{2}x(1-c)^{-3/2}$ for some c in the range $0 < c < x$. For $0 < x < \frac{1}{2}$, this gives $(1-x)^{-1/2} < 1 + \sqrt{2}x$. Thus, for $p\tau < 1/(2C)$,

$$p\tau > \sqrt{f(v)}[1 + \sqrt{2}Cp\tau].$$

If we require $p\tau$ to be less than or equal to $N\sqrt{f(v)}$ for some constant N, then the rightmost term, $\sqrt{2}Cp\tau$, is no more than $\sqrt{2}CNf(v)$. So, a lower bound on $p\tau$ is

$$p\tau > \sqrt{f(v)}[1 + \Theta(\sqrt{f(v)})]. \tag{8}$$

For $f(v) < 1/(128C^2)$, the upper bound on $p\tau$ is at least four times larger than the lower bound.

Since p approaches zero, requiring (7) to be larger than $1 + \epsilon$ is equivalent to

$$\begin{aligned}(1-p)^{2\tau} + 2p\tau(1-p)^{\tau-1} + p^2\tau(\tau-1)(1-p)^{\tau-2} =&\\ (e^{-2p\tau} + 2p\tau e^{-p\tau} + p^2\tau^2 e^{-p\tau})[1 + O(p)] > 1 + \epsilon.&\end{aligned}$$

Solving this numerically gives a solution for all $p\tau$ such that

$$0 < p\tau < 2.304602987. \tag{9}$$

Equations (8) and (9) say that in order for $A_2(t,v)$ to be superpolynomial or exponential, $p\tau$ must be bounded from above and below.

4.2. Results and comparisons. Since $A(t,v)$ is an increasing function of t, the critical question is, for each p, how small can t be and still have $A(t,v)$ be large? If $A(t_0,v)$ is large, then $A(t,v)$ is at least as large for $t > t_0$. Since $\tau = t-2k$, $0 \leqq k \leqq a$, and τ must be greater than zero, we would like a to be as small as possible, but we cannot make it too small because then we would not have enough factors that would lead to a large bound. From (8), if $(t-2a)p$ is small (below $N\sqrt{f(v)}$ for some N) and slightly greater than $\sqrt{f(v)}$, and if pt is also small (below $N\sqrt{f(v)}$ for some N), then the term in square brackets from (4) is larger than $1+f(v)$ for $t-2a < t-2k \leqq t$. All these conditions are satisfied by setting

$$pt = 2\sqrt{f(v)} + 2ap \tag{10}$$

and requiring that $t \geqq 4a$ and that $f(v)$ be small (below $1/(128C^2)$). The lower bound on t ensures that pt and $p(t-2a)$ are about the same size ($pt \geqq p(t-2a) \geqq pt/2$). To keep the formulas simple, the number 2 is used in all places where a constant factor bigger than 1 is needed; slightly better results could be obtained by using smaller constants. Setting $a = g(v)$ in (4) gives us $g(v)$ terms, where each term is larger than $1+f(v)$.

Recall that (6) says that the bound is superpolynomial for $f(v)g(v) = N(v)\ln v$, so we set $g(v) = (N(v)\ln v)/f(v)$, giving superpolynomial time when

$$pt = 2\sqrt{f(v)} + \frac{2pN(v)\ln v}{f(v)} \tag{11}$$

for $t \geqq 4g(v)$. The right side of (11) is as small as possible when $f(v)$ is chosen so that

$$f(v) = [pN(v)\ln v]^{2/3}. \tag{12}$$

Applying (12) to (11) gives superpolynomial time for

$$t = 4[N(v)\ln v]^{1/3}p^{-2/3},$$

provided $pN(v)\ln v$ is bounded by a small constant (because $f(v)$ must be small) and provided that $[N(v)\ln v]^{1/3}p^{-2/3} < v$ (because $g(v)$ must be smaller than v).

The time for the algorithm is an increasing function of t, thus we have superpolynomial time whenever

$$t \geqq 4[N(v)\ln v]^{1/3}p^{-2/3}, \tag{13}$$

and

$$N(v)^{1/2}v^{-3/2}(\ln v)^{1/2} \leqq p \leqq \frac{1}{N(v)\ln v}. \tag{14}$$

Since $N(v) = \omega(1)$, (13) and (14) give superpolynomial time whenever

$$t = \omega((\ln v)^{1/3}p^{-2/3}), \qquad p = o(1/\ln v), \qquad \text{and} \qquad p = \omega(v^{-3/2}(\ln v)^{1/2}). \tag{15}$$

Equation (13) corresponds to the line in Fig. 1 that separates superpolynomial average time from the lower left unknown region, and (14) leads to the boundary that separates polynomial average time from the upper unknown region.

The small t upper-bound result of [9, Eq. (19)] shows that for any p the time for the Pure Literal Rule Algorithm is polynomial if $t = O(\ln v)$. This is essentially the same as the lower-bound result of (15) when p is just below $\Omega(1/\ln v)$. Thus, we conclude that for p of this form the time for the Pure Literal Rule Algorithm is polynomial if

$$t = O(\ln v),$$

and superpolynomial if

$$t = \omega(\ln v).$$

When p is below the lower bound of (14), the best results come from the case when $g(v) = v - 1$ and $f(v) = [N(v) \ln v]/v$, which gives

$$pt = 2\sqrt{\frac{N(v) \ln v}{v}} + 2pv - 2p. \tag{16}$$

The first term on the right side dominates for $p = O(v^{-3/2}(\ln v)^{1/2})$. The small pt upper bound of [9, Eq. (28)] is the same, except for the $N(v)$ factor, as the lower bound results of (16) for $p = O(v^{-3/2}(\ln v)^{1/2})$. Thus, for $p = O(v^{-3/2}(\ln v)^{1/2})$ the time for the Pure Literal Rule Algorithm is polynomial if

$$t = O\left(\frac{1}{p}\sqrt{\frac{\ln v}{v}}\right),$$

and it is superpolynomial if

$$t = \omega\left(\frac{1}{p}\sqrt{\frac{\ln v}{v}}\right).$$

There is no boundary line for (16) in Fig. 1, because it applies to a region where backtracking uses polynomial average time.

To show exponential time we need $a = g(v) = \alpha v$ and $f(v) = \epsilon$ (see (5)). Applying this to (10), we have exponential time when

$$pt = \epsilon^{1/2} + 2\alpha pv.$$

This equation has no solution unless $t > 2\alpha v$. Therefore, for $t = \beta v$ with $\beta > 2\alpha$, the bound is exponential whenever

$$p \leqq \frac{\epsilon^{1/2}}{(\beta - 2\alpha)v}.$$

If (19) and (28) of [9] are modified to obtain an exponential bound rather than a polynomial bound, then the same equation is obtained (within a constant factor). Thus, for $p = O(1/v)$ the time for the Pure Literal Rule Algorithm is exponential if

$$t = \Omega\left(\frac{1}{p}\right) \tag{17}$$

and subexponential if

$$t = o\left(\frac{1}{p}\right). \tag{18}$$

Equations (17) and (18) define the left boundary of the exponential average-time region, as shown in Fig. 1.

Our lower-bound analysis does not give a lower bound to match the large p upper bound of [9, Eq. (24)]. These results and the results from previous analyses of satisfiability algorithms are shown in schematic form in Fig. 1.

Numerical calculations similar to those used to derive (9) show that for the case when n is equal to 3 the lower bound is exponential whenever

$$0 < pt < 3.536962614.$$

This shows the tendency to extend the range of pt for which we have an exponential lower bound as n increases. (Since we must have $t - na \geqq 1$, increasing n decreases the range of t over which the resulting lower bound is exponential.) If this result is used in the previous analysis only the size of some constants change.

Finally, one should note that when pt is large, it is possible to use the technique of [1] to approximate the sum in (2). If n is slightly larger than pt the sum is near 2, whereas if n is slightly smaller than pt then the sum is near zero. Unfortunately, this approach does not lead to smaller bounds on t for large p [i.e., $p = \Omega(1/\ln v)$]. That is because $t - ng(v)$ must be larger than 1. If $n = pt$, this implies that $pg(v)$ is smaller than 1. Therefore, this approach does not produce interesting results for $p = \Omega(1/\ln v)$, since the smallest value for $g(v)$ that leads to superpolynomial time is $\omega(\ln v)$.

5. Conclusion. For the random clause model of generating random predicates, where there are v variables, t clauses, and probability p, each literal is in a clause; the average time of the Pure Literal Rule Algorithm is exponential in v when the average clause size is constant ($p = \Theta(1/v)$) and the average number of clauses per literal is constant or growing ($t = \Omega(1/p)$) (see Fig. 1). The average time grows faster than any polynomial function of v over a much larger region. For $p = O(1/\ln v)$ the results of the upper- and lower-bound analyses are the same except for the region between $p = o(1/\ln v)$ and $p = \omega((\ln v)/v^{3/2})$, and they are close in this region. More work is needed on the upper and lower bounds for $p = \omega(1/\ln v)$.

This work, combined with the analyses of backtracking algorithms [8], [11], show that the region where the average clause size is constant ($p = \Theta(1/v)$) and the average number of clauses is proportional to the number of variables ($t = \Theta(v)$), is the most difficult region for a satisfiability algorithm to handle. Backtracking algorithms can handle a portion of this region efficiently. The design of satisfiability algorithms should concentrate on algorithms that can handle more of this region efficiently.

REFERENCES

[1] D. ANGLUIN AND L. G. VALIANT, *Fast probabilistic algorithms for Hamiltonian cycles in non-oriented graphs*, J. Comput. System Sci., 19 (1979), pp. 155–193.

[2] K. BUGRARA AND P. W. PURDOM, JR., *An exponential lower bound for the Pure Literal Rule*, Inform. Process. Lett. 27 (1988), pp. 215–219.

[3] C. A. BROWN AND P. PURDOM, JR., *An average time analysis of backtracking*, SIAM J. Comput. 10 (1981), pp. 583–593.

[4] A. GOLDBERG, *On the complexity of the satisfiability problem*, Courant Computer Science Report No. 16, New York University, New York, NY, 1979.
[5] A. GOLDBERG, P. W. PURDOM, JR., AND C. A. BROWN, *Average time analysis of simplified Davis–Putnam procedures*, Inform. Process. Lett. 15 (1982), pp. 72–75. Erratum published in 16 (1983), p. 213.
[6] K. IWAMA, *CNF satisfiability by counting and polynomial average time*, Report KSU/ICS 88-01, Institute of Computer Science, Kyoto Sangyo University, Kyoto, Japan 1988.
[7] D. E. KNUTH, *Big omicron and big omega and big theta*, SIGACT News, April–June (1976), pp. 18–24.
[8] P. W. PURDOM, JR., *Search rearrangement backtracking*, Artificial Intelligence, 21 (1983), pp. 117–133.
[9] P. W. PURDOM, JR. AND C. A. BROWN, *The Pure Literal Rule and polynomial average time*, SIAM J. Comput., 14 (1985), pp. 943–953.
[10] ———, *The Analysis of Algorithms*, Holt, Rinehart and Winston, New York, 1985, p. 204.
[11] ———, *Polynomial average-time satisfiability problems*, Inform. Sci. 41 (1987), pp. 23–42.

SIAM J. COMPUT.
Vol. 18, No. 2, pp. 419–427, April 1989

017

A NEW PARALLEL ALGORITHM FOR THE MAXIMAL INDEPENDENT SET PROBLEM*

MARK GOLDBERG†‡ AND THOMAS SPENCER†§

Abstract. A new parallel algorithm for the maximal independent set problem is constructed. It runs in $O(\log^4 n)$ time when implemented on a linear number of EREW-processors. This is the first deterministic algorithm for the maximal independent set problem (MIS) whose running time is polylogarithmic and whose processor-time product is optimal up to a polylogarithmic factor.

Key words. parallel computation, $\mathcal{NC}$, efficient, deterministic, maximal independent set, matching

AMS (MOS) subject classification. 68R10

1. Introduction.

When researchers investigate the parallel complexity of a problem, one of the main questions they ask is whether a polylogarithmic running time is achievable on a PRAM containing a polynomial number of processors. If the answer is positive, then the problem and the corresponding algorithm are said to belong to class $\mathcal{NC}$ introduced in [22] (see also [8], [25]). Having constructed an $\mathcal{NC}$-algorithm for a given problem, it is natural to try to improve its computational complexity. The complexity of a parallel algorithm is characterized by the pair (T, P), where $T = T(N)$ is the worst-case running time, $P = P(N)$ is the number of processors used, and N is the size of the input. It has been traditional to consider the product $W = T(N)P(N)$ as a unified measure for different parallel algorithms solving the same problem. W represents the total amount of work that the parallel algorithm does; it is also the running time of the sequential algorithm into which the algorithm can be converted.

Let a sequential algorithm A_s with running time T_s and a parallel algorithm A_p solve the same problem. The *relative efficiency* $E(A_p, A_s)$ of A_p with respect to A_s measures the amount of extra work that A_p does, and it is given by $E = T_s/W_p$. The value of $E(A_p) = E(A_p, A_s)$, where A_s is the fastest known sequential algorithm for the problem, characterizes the efficiency of A_p. Thus, in general, this characteristic of a parallel algorithm depends on the progress in designing a sequential algorithm, but for the problems with a linear sequential algorithm, $E(A_p)$ is absolute. Clearly, the optimal algorithms introduced by Galil in [9] have the maximum relative efficiency of $O(1)$. We should expect that for many problems, achieving the speedup from a polynomial to a polylogarithmic running time is only possible if the efficiency $E(A_p)$ is a function that tends to 0 when $N \to \infty$. Consequently, the algorithms whose relative efficiency is large enough may be called *efficient*. In particular, we call an $\mathcal{NC}$-algorithm *efficient* if it has relative (to the fastest sequential algorithm) efficiency at least $\Omega(1/\log^l N)$, where l is a constant.

In this paper, we present an *efficient deterministic* parallel algorithm for the *Maximal Independent Set* problem (MIS). Recall that a subset I of the vertices of a graph G is *independent* if there are no edges between any two vertices in I. An

* Received by the editors February 9, 1987; accepted for publication (in revised form) April 26, 1988.

† Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180-3590.

‡ The work of this author was supported in part by the National Science Foundation under grant DCR-8520872.

§ The work of this author was supported in part by the National Science Foundation under grant CCR-8810609.

independent set I is *maximal* if it is not a proper subset of any other independent set. MIS is the problem of constructing a maximal independent set of a given graph.

Karp and Widgerson were the first to prove that MIS is in $\mathcal{NC}$. On graphs with n vertices and m edges, their algorithm [17] runs in $O(\log^4 n)$ time and uses $O(n^3/\log^3 n)$ processors. In [19] Luby constructed a probabilistic algorithm that runs in $O(\log n)$ time when implemented on a linear number of processors under the CRCW PRAM model of computation. In the EREW PRAM model, it runs in $O(\log^2 n)$ time. The deterministic version of the algorithm uses $O(n^2 m)$ processors. A still different probabilistic algorithm for MIS was described by Alon, Babai, and Itai in [4].

Since there is a trivial sequential algorithm that runs in linear time, every efficient algorithm for MIS must use at most a linear number of processors (up to a polylogarithmic factor). A deterministic algorithm that uses a linear number of processors was proposed by Goldberg in [12]; its running time is $O(n^\alpha)$, where $\alpha > \frac{1}{2}$ is arbitrary. Using the *deterministic coin-flipping technique* introduced by Cole and Vishkin in [6], Goldberg, Plotkin, and Shannon [11] developed algorithms for MIS as well as for the vertex-coloring problem, VC, which run in $O(\log^* n)$ time[1] on graphs with bounded maximum degree. Unfortunately, when the degree is allowed to grow, the algorithms become inefficient.

The algorithm we present in this paper runs in $O(\log^4 n)$ time on an EREW PRAM consisting of $O(m+n)$ synchronous processors that share a common memory [8], [22], [25], [26]. Each processor is a standard random access machine [2] capable of doing elementary operations on words of length $O(\log(n+m))$.

We follow the usual graph-theoretic terminology [7]. Our graphs are without loops or parallel edges. The vertices of a graph on n vertices are represented by integers $0, 1, \cdots, n-1$; the edges are given by a list of pairs $\{(i,j)\}$, where $0 \le i < j \le n-1$. Given a set T of vertices of a graph $G = (V, E)$, the neighborhood $N(T)$ is defined as the set of all vertices in V that are adjacent to at least one vertex in T. Thus, a set I is independent if $I \cap N(I) = \emptyset$. A subgraph of a graph G induced on a set A of vertices is denoted by $G[A]$. A matching is a collection of disjoint edges.

A partial coloring ϕ of a graph G is given by a collection of disjoint independent subsets $(C_1, \cdots, C_p)$ of G. We say that the vertices in C_i have color i $(1 \le i \le p)$; thus, the colors are always positive integers. If $V(G) = \cup_{i=1}^{p} C_i$, then ϕ is called complete. A trivial partial coloring is that for which $p = |V(G)|$; hence, for a trivial partial coloring, every vertex is its own color class. Given a partial coloring $\phi = (C_1, \cdots, C_p)$, we define matrix $D(\phi) = (d_{ij})$ and function $Q(\phi)$ by

$$d_{ij} = |N(C_i) \cap C_j|, \qquad (i, j = 1, \cdots, p);$$

$$Q(\phi) = \max_{1 \le i \le p} (|C_i| + |N(C_i)|).$$

For $h > 0$, we define a graph $B(\phi, h)$, on p vertices, by setting vertices i and j adjacent if and only if both $d_{ij} \ge h$ and $d_{ji} \ge h$ $(0 \le i, j \le p-1)$.

If L is a list of items sorted according to a key function f, then a maximal sublist of L with identical values of the key is called an interval of L. Every sorted list L can be viewed as the concatenation of its intervals.

A pair (r, s) is lexicographically smaller than another pair (r', s') if and only if either $r < r'$, or $r = r'$ and $s < s'$.

---

[1] We write $log\, n$ for $log_2 n$ and $log^*\, n$ for the minimum i such that the ith iteration of log function applied to n is < 2.

2. The algorithm. All parallel algorithms for MIS mentioned above as well as our algorithm have the same top-level description as the very first algorithm developed by Karp and Widgerson in [17].

```
begin
    I := ∅;      A := V(G);
    (* I is an independent set *)
    while A ≠ ∅ do
        begin
        C := FINDSET(A);
        I := I ∪ C;
        A := A − (C ∪ N(C))
    end;
end;
```

It is not hard to prove that an algorithm with such a structure will have a polylogarithmic running time if every application of FINDSET runs in polylogarithmic time and produces independent set C such that $|C \cup N(C)| = \Omega(|A|/\log^s |A|)$ for some fixed $s \geq 0$. Later, we will see that for our version of FINDSET, $s = 1$.

Informally, FINDSET works as follows. Starting with a trivial partial coloring ϕ_0 of graph $H = G[A]$, it constructs a sequence of partial colorings ϕ_j $(j \geq 0)$. For each $j \geq 0$, the procedure checks whether

$$(*) \qquad Q(\phi_j) > \frac{c_0 k}{\log k}$$

where $k = |A|$. If $(*)$ holds, FINDSET outputs the color class C' for which $|C'| + |N(C')| > c_0 k/\log k$; otherwise, it constructs a new partial coloring by decoloring some of the vertices and uniting some of the color classes. The construction is done by the procedure REDUCE. The input to REDUCE is a partial coloring ϕ and a matching M in the complement $\bar{B}$ of graph $B(\phi, h)$ (the selection of $h > 0$ is specified later). The matching M supplies a collection of pairs of color classes of ϕ. For each pair (C, C'), either the set $C \cap N(C')$ or the set $N(C) \cap C'$ (whichever is smaller) is decolored and the remaining vertices in $C \cup C'$ are declared to be a new color class. The new color classes obtained in this way, and the old color classes that were not changed, comprise the set of color classes of the new partial coloring.

A procedure MATCH finds a matching in $\bar{B}(\phi, h)$. It will be seen that the running time of the whole algorithm depends on the size of the matching delivered by this procedure as well as on its running time. In the context of our algorithm, each graph B to which MATCH is applied is such that its complement $\bar{B}$ contains a quadratic number $\Theta(|V(B)|^2)$ of edges even if the original graph G is sparse. This is our reason for not using any of the known algorithms for constructing a maximal matching (see [1], [10], [14], [15], [16], [18], [19], [20]). The subroutine MATCH runs in $O(\log n)$ time on an EREW PRAM with $O(n+m)$ processors. For graphs with a dense complement, MATCH constructs a matching of size $\Omega(|V(B)|)$, which is maximum up to a constant. On the other hand, it is not necessarily maximal.

Our algorithm uses $O(n + m)$ processors; every vertex and every edge of a graph has a processor associated with it; abusing the language, we identify a vertex or an edge with the corresponding processor. Each edge has, for each of its endpoints, a pointer to a record that stores the color of that vertex. If $\Delta(v)$ is the degree of a vertex v, then there are $\Delta(v)$ records containing the information related to v. Thus,

each edge can access its endpoints independently. An uncolored vertex has its color set to 0; a vertex that has been deleted by an earlier iteration of the top-level loop has its color set to -1.

A Pascal-like description of FINDSET is as follows:

```
function FINDSET(A);
begin
    k := |A|;      H := G[A];
    φ := trivial coloring of H;
    while Q(φ) < c0 k/log k do
        begin
            p:= the number of colors of φ_i;
            h := c1 k/(p log k);
            B := BUILD(φ, h);
            M := MATCH(B);
            φ := REDUCE(φ, h, M)
        end;
    C := a color class with Q(C) ≥ c0 k/log k;
    FINDSET := C
end;
```

In the description of FINDSET and MATCH we use two constants, c_0 and c_1; their values, which guarantee the necessary performance behavior, are defined in §3.

The function BUILD accepts a partial coloring ϕ and a number $h > 0$ and constructs the auxiliary graph $B(\phi, h)$. It does this by computing the values of the d_{ij} that are nonzero, and then finding out which d_{ij} and d_{ji} are both greater than h.

To calculate the d_{ij}, BUILD first creates a list of records containing, for each edge, its endpoints and their colors written in increasing order. Next, BUILD sorts this list lexicographically by color. Each interval of the resulting list consists of the edges with the endpoints colored by the same pair of colors. Let L_{ij} be the interval containing the edges whose endpoints are colored i and j. To calculate d_{ij}, BUILD sorts L_{ij} by the vertex colored j; the number of intervals of this list is the value of d_{ij}. Similarly, d_{ji} is the number of intervals that result when L_{ij} is sorted by the vertex colored i.

To find the intervals of a sorted list, every member of the list compares itself with the element on its right and the element on its left. This indicates the elements that are the ends of the intervals. Then, every other member assigns itself to the corresponding interval. This can be done in $O(\log n)$ time using the path-doubling technique of Wyllie [27].

```
function BUILD(φ, h);
begin
    L := a list of the edges with the colors of their endpoints
            listed in increasing order;
    sort L lexicographically by the colors of the endpoints;
    determine the set of intervals of L;
    for each interval L_ij in parallel do
        (* the subscripts i, j are the corresponding colors *)
        begin
            sort L_ij in increasing order of the endpoint colored by j;
            set d_ij to be the number of intervals of L_ij;
```

```
            sort L_ij in order of the endpoint colored by i;
            set d_ji to be the number of intervals of L_ij;
            if d_ij > h and d_ji > h then
                include (i, j) in E(B(φ, h));
        end;
end;
```

All sorts are done using Cole's algorithm [5]; thus, BUILD runs in $O(\log n)$ time on an EREW PRAM with a linear number of processors.

The idea of the procedure MATCH is as follows. Let the vertices of B be numbered by $0, 1, \cdots, p-1$, where $p = |V(B)|$, let K_p be the complete graph on $V(B)$, and let $\chi = p$ if p is odd, and $\chi = p-1$ if p is even. It is well known that there is a partition of the edges of K_p into χ matchings $P_0, P_1, \cdots, P_{\chi-1}$, each of size exactly $\lfloor p/2 \rfloor$. We give an explicit construction of such a partition in terms of the function "index" defined below, where the edge (i, j) is assigned to the matching $P_{\text{index}(i,j)}$. If the total number of edges of $\bar{B}$ is quadratic, the set $\bar{M}_t$ of the maximum size contains $\Omega(p)$ edges. For the same t, the size of M_t is a minimum. When such a t is found, we can check every of $\lfloor p/2 \rfloor$ pairs of $\bar{P}_t$ to compute $\bar{M}_t$.

```
function index(i, j, p);
begin
    if p is odd then
        index := (i + j) mod p
    else
        if j = p − 1 then
            index := 2i mod (p − 1)
        else
            index := index(i, j, p − 1)
        end;

function MATCH(B);
begin
    M := ∅; p := |V(B)|;
    if p is even then χ := p − 1 else χ := p;
    for each edge (i, j) of B in parallel
        compute index(i, j, p);
    for l := 0 to χ − 1 in parallel compute g(l)
        the number of edges (i, j) with index(i, j, p) = l;
    find t such that g(t) is minimized;
    compute M := P_t ∩ B̄;
    remove all but ⌈p(c_1 − c_0)/(2c_1)⌉ edges from M;
end;
```

Computing the number of the edges with a given index is done by sorting the edges of B according to their indices and then determining the lengths of all intervals. Finding a t for which the corresponding color class P_t contains the fewest number of edges can be done using Valiant's algorithm [24]. To determine $P_t \cap \bar{B}$, MATCH appends a list of edges in $P_t \cap B$ to the list of the edges in P_t. Then, the list is sorted lexicographically to bring the duplicates next to each other. If a pair (a, b) occurs in the list twice, then both occurrences are removed. The remaining pairs are a list of the edges in $\bar{B} \cap P_t$.

It turns out that the analysis is simpler if the matching returned by MATCH has a known size. We will show in §3 that, for our choice of c_0 and c_1, the size of $\bar{B} \cap P_t$ is at least $p(c_1 - c_0)/(2c_1)$. Thus, after removing a few edges from $\bar{B} \cap P_t$, MATCH returns a matching with exactly $\lceil p(c_1 - c_0)/(2c_1) \rceil$ edges. It is easy to see that every application of MATCH is executed on a linear number of processors in $O(\log n)$ time.

The matching M calculated by MATCH is used by REDUCE to construct a new partial coloring with a smaller number of color classes. In particular, REDUCE does the following three things:

(1) Decides which vertices are to be decolored;
(2) Merges the appropriate color classes; and
(3) Renumbers the color classes.

It is easy to implement REDUCE so that it runs in $O(\log n)$ time on a CRCW PRAM with a linear number of processors, which also yields an implementation on an EREW PRAM running in $O(\log^2 n)$ time. A more elaborate technique is needed to implement REDUCE so that it runs in $O(\log n)$ time on an EREW PRAM.

Intuitively, each vertex of color l needs to know which color, if any, is matched to l by M. We use a routine called BROADCAST to deliver this information. Specifically, BROADCAST is given a list L of ordered pairs of the form (l_i, m_i), where l_i is a color, m_i is a "message," and each color appears on the list at most once. The task of BROADCAST is to label each vertex of color l_i with the message m_i. For this purpose, BROADCAST first creates a new list L' with one record for each colored vertex. Each record is of the form (l_v, v), where l_v is the color of vertex v. Then, it sorts the concatenation of L' and L by color, that is by first coordinate; if two pairs from L and L', respectively, have the color, the pair from L is declared to be smaller. Next, BROADCAST uses the standard path-doubling technique to give m_i to each element of the list with color l_i. Finally, each element from L' that received a message labels its vertex with the message. Clearly, BROADCAST runs in $O(\log n)$ time and uses $O(n + m)$ processors.

In the context of REDUCE, the procedure BROADCAST is used to decide, for each edge $(i, j) \in M$, whether to decolor vertices with color i or with color j. Recall that the vertices of color j are decolored if and only if $d_{ij} \leq d_{ji}$. Both values d_{ij} and d_{ji} can either be obtained from BUILD, or REDUCE can calculate them itself. Having obtained these values, REDUCE orients each edge (i, j) of M so that $d_{ij} \leq d_{ji}$; hence the color of the vertices to be decolored is listed second. Then, REDUCE calls BROADCAST(M) to tell which vertices need to change color. If a vertex v receives a color l as a message, it checks to see if it has a neighbor of color l. If it does, it decolors itself; otherwise it changes its color to l.

Finally, REDUCE renumbers the surviving color classes by consecutive integers, starting with zero. This is necessary for the next application of MATCH to be done correctly. To do this, it sorts all the vertices by color and removes duplicates. The result is a list of all the colors in use. This list is then numbered, and the position of each color in the list is broadcast. Each colored vertex then changes its color to the color it receives.

A Pascal-like description of REDUCE is as follows:

```
function REDUCE(φ, h, M);
begin
    orient each edge (i, j) of M so that d_ij ≤ d_ji;
    BROADCAST(M);
    for each vertex v that received a color, l(v), in parallel do
```

```
            if v is adjacent to a vertex of color l(v)
                then decolor v
              else change the color of v to l(v);
        sort the vertices by their (new) colors;
        number the colors in use;
        make L, a list (l, n(l)), where n(l) is the number of l;
        BROADCAST(L);
        each colored vertex changes its color to the message it received;
    end;
```

3. Analysis. Our goal is to show that using $O(n+m)$ processors and in $O(\log^2 n)$ time FINDSET constructs an independent set C such that $|C \cup N(C)| > c_0 k/\log k$, where $|A| = k$ and c_0 is a constant. Once this is established, it is easy to see that FINDSET is called $O(\log^2 n)$ times and that the running time of the algorithm is $O(\log^4 n)$.

First, let us estimate the size of the set that MATCH returns. Recall that FINDSET calls MATCH on the graph $B(\phi, h)$. Let ϕ_i be the value of ϕ at the beginning of the ith iteration of the body of the *while* loop in FINDSET, and let p_i be the number of color classes in ϕ_i. If the body of the loop is executed then

$$Q(\phi_i) < \frac{c_0 k}{\log k}.$$

Let Δ_i be the maximum degree of a vertex in $B(\phi_i, h_i)$, where $h_i = c_1 k/(p_i \log k)$. If the degree, in B, of a vertex corresponding to a color class C is Δ_i, then $N(C)$ must contain at least $\Delta_i h_i$ vertices. On the other hand, $|N(C)| < c_0 k/\log k$. Therefore,

$$\Delta_i \frac{c_1 k}{p_i \log k} < \frac{c_0 k}{\log k}, \qquad \text{and} \qquad \Delta_i < \frac{c_0}{c_1} p_i.$$

This implies that the degree of every vertex of $\bar{B}(\phi_i, h_i)$, is at least $p_i(c_1 - c_0)/c_1$, yielding

$$|E(\bar{B}(\phi_i, h_i))| \geq \frac{c_1 - c_0}{2c_1} p_i^2.$$

Recall that MATCH divides the edges of $\bar{B}(\phi_i, h_i)$ into at most p classes and finds $P_t \cap \bar{B}$, the class with the most edges. Thus,

$$|P_t \cap \bar{B}| \geq \frac{c_1 - c_0}{2c_1} p_i,$$

and MATCH can discard edges from this set to return a matching M_i with

$$|M_i| = \frac{c_1 - c_0}{2c_1} p_i.$$

When REDUCE creates ϕ_{i+1}, it decolors at most

$$|M_i| \frac{c_1 k}{p_i \log k} = \frac{(c_1 - c_0)k}{2 \log k}$$

vertices and reduces the number of color classes to ap_i, where $a = (c_1 + c_0)/(2c_1)$. The initial partial coloring ϕ_0 has k color classes with a total of k vertices. Thus the

while loop in FINDSET will be executed at most $-\log k/\log a$ times. If none of the partial colorings ϕ_i with more than one color class satisfies

$$Q(\phi_i) \geq \frac{c_0 k}{\log k},$$

then FINDSET decolors at most

$$\frac{\log k}{-\log a}\frac{(c_1 - c_0)k}{2\log k}$$

vertices while reducing the number of color classes to one. Thus, the size of the only color class of the last coloring is at least

$$Q_0 = k - \frac{c_1 - c_0}{2}\frac{k}{\log k}\frac{\log k}{(-\log a)} = k - \frac{c_1 - c_0}{2}\frac{k}{(-\log a)}.$$

If we choose c_0 and c_1 such that

$$-\log a = \log\left(\frac{2c_1}{c_1 + c_0}\right) > (1+\epsilon)\frac{c_1 - c_0}{2c_1(1 - c_0)},$$

for some fixed $\epsilon > 0$, then

$$Q_0 > \frac{\epsilon}{1+\epsilon}k > \frac{c_0 k}{\log k},$$

for sufficiently large k. The criterion will be satisfied, for example, if $c_1 = 1$ and $c_0 = 1/3$.

Since every application of the *while* loop is executed in $O(\log n)$ time and the number of times the loop is iterated is $O(\log n)$, we have that the running time of FINDSET is $O(\log^2 n)$. It implies that the running time of the whole algorithm is $O(\log^4 n)$.

4. Open problems. This work addresses several interesting unresolved questions, including the following:

(1) The processor-time product for our algorithm is $O((n+m)\log^4 n)$. We would like to reduce the total amount of work that our algorithm does by reducing the number of processors it requires while not increasing its running time. We note that the technique developed by Miller and Reif in [20] does not seem to apply to this algorithm.

(2) There is a trivial sequential algorithm that colors a given graph G in at most $\Delta + 1$ colors, where Δ is the maximal degree of a vertex in G. Using the standard reduction of VC to MIS, we get an $\mathcal{NC}$-algorithm for $\Delta + 1$-coloring which is run on $O(n\Delta^2 + m\Delta)$ processors. However, no efficient $\mathcal{NC}$-algorithm for $\Delta + 1$-coloring is known.

(3) In [23] Túran proved that every graph with n vertices and m edges contains an independent set of size $\geq n^2/(2m+n)$. Such a set can be constructed by a linear sequential algorithm [13]. Can it be constructed by an $\mathcal{NC}$-algorithm using a linear number of processors? So far, the best approximation is achieved by an algorithm COLOR from [12]. It produces a coloring such that the size of at least one color class is $n/2$ if $m \leq n/4$, and $n^2/(32m)$ otherwise. The algorithm uses a linear number of processors and runs in $O(\log^3 n)$ time.

Acknowledgment. We are grateful to both referees for their helpful comments on the first version of this paper.

REFERENCES

[1] A. AGGARWAL AND R. ANDERSON, *A random NC-algorithm for depth first search,* in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 325–334.

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

[3] M. AJTAI, J. KOMLÓS, E. SZEMERÉDI, *An $O(n \log n)$ sorting network,* Combinatorica, 3 (1983), pp. 1–19.

[4] N. ALON, L. BABAI, AND A. ITAI, *A fast and simple randomized parallel algorithm for the maximal independent set problem,* J. Algorithms, 7 (1986), pp. 567–583.

[5] R. COLE, *Parallel merge sort,* in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 511–516.

[6] 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, 1986, pp. 206–219.

[7] G. CHARTRAND AND L. LESNIAK, *Graphs & Digraphs*, Wadsworth, 1986.

[8] S. A. COOK, *Taxonomy of problems with fast parallel algorithms,* Inform. and Control, 64 (1985), pp. 2–22.

[9] Z. GALIL, *Optimal parallel algorithms for string matching,* in Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 240–248.

[10] Z. GALIL AND V. PAN, *Improved processor bound for algebraic and combinatorial problems in* RNC, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 490–495.

[11] A. GOLDBERG, S. PLOTKIN, AND G. SHANNON, *Parallel symmetry–breaking in sparse graphs,* in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 315–324.

[12] M. GOLDBERG, *Parallel algorithms for three graph problems,* Congr. Numer., 54 (1986), pp. 111–121.

[13] M. GOLDBERG, S. LATH, AND J. ROBERTS, *Heuristics for the graph bisection problem,* Tech. Report TR 86-8, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY.

[14] A. ISRAELI AND A. ITAI, *A fast and simple randomized parallel algorithm for maximal matching,* Computer Science Department, Technion, Haifa, Israel, 1984.

[15] A. ISRAELI AND Y. SHILOACH, *An improved parallel algorithm for maximal matching,* Inform. Process. Lett., 22 (1986), pp. 57–60.

[16] R. M. KARP, E. UPFAL, AND A. WIDGERSON, *Constructing a perfect matching is in random NC,* in Proc. 17th Annual ACM Symposium on Theory of Computing, 1985, pp. 22–32.

[17] R. M. KARP AND A. WIDGERSON, *A fast parallel algorithm for the maximal independent set problem,* in Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 266–272.

[18] G. LEV, N. PIPPENGER, AND L. VALIANT, *A fast parallel algorithm for routing in permutation networks,* IEEE Trans. on Comp., 30 (1981), pp. 93–100.

[19] M. LUBY, *A simple parallel algorithm for the maximal independent set problem,* SIAM J. Comput., 15 (1986), pp. 1036–1053.

[20] G. L. MILLER AND J. H. REIF, *Parallel tree contraction and its applications,* in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 478–489.

[21] K. MULMULEY, U. V. VAZIRANI AND V. V. VAZIRANI, *Matching is as easy as matrix inversion,* in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 345–354.

[22] N. PIPPENGER, *On simultaneous resource bounds,* in Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, 1979, pp. 307–311.

[23] P. TÚRAN, *On the theory of graphs,* Colloq. Math., 3 (1954), pp. 19–30.

[24] L. G. VALIANT, *Parallelism in comparison problems,* SIAM J. Comput., 4 (1975), pp. 348–355.

[25] ———, *Parallel computation,* in Proc. 7th IBM Symposium on Mathematical Foundations of Computer Science, 1982.

[26] U. VISHKIN, *Synchronous parallel computation—a survey,* Preprint, Courant Institute, New York University, NY, 1983.

[27] J. C. WYLLIE, *The complexity of parallel computations,* Ph. D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1979.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 429-448, June 1989

A DESIGN THEORY FOR SOLVING THE ANOMALIES PROBLEM*

EDWARD P. F. CHAN†

Abstract. A theory is proposed for designing database schemes that are free of update anomalies. Unlike previous approaches, insertion and deletion anomalies are investigated in the context of a relation scheme, while replacement anomalies are studied in the context of a database scheme. Two simple models are developed for analyzing when a relation scheme is free of insertion and deletion anomalies. Techniques are also proposed for obtaining desirable decompositions that are free of insertion and deletion anomalies. A class of database schemes that is free of replacement anomalies is also proposed. This class of schemes is highly desirable with respect to constraint enforcement when attribute values of some tuple are being changed. By making different assumptions on the modifiable attributes, several important classes of database schemes that are free of replacement anomalies are characterized. Throughout, we assume update operations are performed on relation schemes at the conceptual level.

Key words. schema design, update anomalies, relational database, functional dependencies

AMS(MOS) subject classification. H21

1. Introduction.

Database design theory began with the pioneering work of Codd [Cod1], [Cod2]. Codd observed that in the presence of functional dependencies, updating a relation at the conceptual level may result in certain problems that are widely known as *update anomalies* [Cod2]. Codd argued that the cause of such problems is due to several independent facts being stored in the same relation, causing the relation to be semantically overloaded. He proposed normalization as a way of separating independent facts into different relations and reducing logical data duplication. Since then, normalization has generated a great deal of interest among researchers as well as practitioners [Da], [Ma], [TL], [U]. Various normal forms and other desirable properties related to normalization have been proposed [ABU], [B], [BDB], [Cod1], [Cod2], [Cod3], [F1], [F2], [Z]. To illustrate the problem of update anomalies, let us consider the following example from Codd [Cod2].

Example 1. Let us consider $R(Emp, Dept, Manager, Contract\text{-}type)$, and let the functional dependencies imposed on R be $\{Emp \rightarrow Dept,\ Dept \rightarrow Manager\ Contract\text{-}type\}$. With this set of functional dependencies, the primary key of R is *Emp*. In this relation, Codd assumed the following facts or relationships are being stored: The *Emp_Dept* relationship, which tells the department for whom an employee is working; the *Dept_Manager* relationship, which gives the manager of each department; and the *Dept_Contract-type* relationship, which indicates the type of contracts a department is handling. For semantic reasons, Codd further assumed that each inserted tuple cannot be null on the primary key of the relation [Cod2]. Hence under Codd's assumptions, partial tuples are allowed in a relation, but the tuples cannot be null on the primary key. With these assumptions, certain problems arise when the relation is being updated.

When a new department is set up to handle a particular contract type, the data cannot be entered into R unless an employee has been hired for the new department. The cause of this problem is the assumption that tuples in a relation cannot be null on the primary key. This phenomenon may be considered an anomaly because setting

* Received by the editors October 6, 1986; accepted for publication August 12, 1988. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

† Department of Computer Sciences, University of Alberta, Edmonton, Alberta, Canada T6G 2E9. Present address, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1.

up a new department should not imply that some employee has been hired for the department. On the other hand, when the last employee working in a department is fired, the tuple representing the employee is deleted. Consequently, the associated department information will cease to exist. This event may again be considered an anomaly since department information should be independent of employee information. The inability to insert data into a relation and the unexpected deletion of data are widely known as *insertion anomalies* and *deletion anomalies*, respectively.

Now consider that the manager of a department is being replaced. Such a change necessitates a series of modifications to the manager for each employee working in that department. This event may again be considered an anomaly, since modifying an attribute value of a tuple results in an unpredictable number of tuples being updated. This phenomenon is called a *replacement anomaly* in the literature [BG], [LP]. □

Codd argued that the cause of updated anomalies is due to several independent facts being stored in the same relation. Normalization is an attempt to separate independent facts into different relations and to reduce logical data duplication using constraint information, such as functional dependencies. Until Bernstein and Goodman [BG] questioned the benefit of normalization, it was widely held that normalization eliminates update anomalies. In fact, normalization has *never* proven to solve the anomalies problem [BG], [LP]. The following example shows that a relation scheme R in BCNF does not necessarily mean that R is free of insertion and deletion anomalies.

Example 2. Let $R(Course, Student\_name, Grade)$ and $F = \{Course\ Student\_name \rightarrow Grade\}$. R clearly is in BCNF. If we assume that *Student_name* information can be recorded independently, then R has insertion and deletion anomalies since *Student_name* information cannot be inserted and deleted independently of $\{Course, Student\_name, Grade\}$. □

The above example illustrates that basic facts or relationships stored in a relation may not be completely determined by constraints like functional dependencies. To capture a portion of database semantics that is not captured by functional dependencies, Sciore [Sc] introduced *objects* to define basic facts in a relation. Maier and Ullman [MU] argued that not all objects can be related, and they suggested *maximal objects* to limit the scope of connections among objects. Maier and Warren [MW] went a step further by proposing a semantically richer model called the association-object data model. *Associations* are basic facts recorded in a database and are nondecomposable. *Objects* are decomposable relationships defined from associations. Various semantically motivated considerations have been discussed to constrain how objects and associations can be syntactically related. Desai, Goyal, and Sadri [DGS1], [DGS2] studied the problem of insertion and deletion anomalies, nevertheless, with assumptions that are different from Codd's. They introduced the *primary fact structure* of a relation as a semantic structure describing nondecomposable information represented by a relation. Primary fact structures are essentially associations in the association-object data model. Having introduced primary fact structures, Desai et al. investigated how to insert and delete facts from a relation without any "side effects." An important difference between Codd's assumptions and the aforementioned authors' assumptions is that Codd assumed each relation has a primary key, while the concept of key is not inherent in other approaches. Since our work is primarily based on Codd's assumptions, the problem as well as results obtained here are different from those mentioned above.

In this paper, we present an alternative to the update anomalies problem. In the same way as Bernstein and Goodman [BG], we view insertion and deletion anomalies as a problem that is different from replacement anomalies. For instance, if we assume in Example 1 that tuples are only inserted and deleted, but not modified, then the

insertion and deletion anomalies still exist but the replacement anomalies as conceived by Codd will disappear. Similarly to Codd [Cod2], we regard insertion and deletion anomalies as a problem of the ability of a relation scheme to record independent facts, whereas replacement anomalies are considered as a problem of independent modification of tuple values in relations. As we will show later, the replacement anomaly problem is closely related to the constraint enforcement problem. Because of this difference, insertion and deletion anomalies will mainly be analyzed in the context of a single relation scheme. On the other hand, it is not meaningful to discuss replacement anomalies with respect to a relation scheme; instead, we have to study the problem in the context of a database scheme.

In § 2, we define the necessary notation needed throughout this paper. In § 3, we investigate insertion and deletion anomalies. Depending on the assumption of whether null values are allowed in an inserted tuple, two simple models are developed for analyzing when a relation scheme is free of insertion and deletion anomalies. Techniques are also proposed for constructing desirable decompositions that are free of insertion and deletion anomalies. In § 4, we study replacement anomalies and show that this problem closely relates to the constraint enforcement problem. In view of the desirability of replacement-anomaly-freedom, we characterize several important classes of database schemes that are free of replacement anomalies. Finally, we give our conclusion in § 5.

2. Definitions and notation.

2.1. Basics. Following standard notation [Ma], [U], we fix a finite set of attributes $U = \{A_1, \cdots, A_n\}$ and call it the *universe*. We use $Z, Y, X, \cdots$ to denote sets of attributes in U and $A, B, C, \cdots$ to denote attributes in U. A *relation scheme* R is a subset of U. A *database scheme* $\mathbf{R} = \{R_1, \cdots, R_k\}$ is a collection of relation schemes such that the union of the R_i's is U.

Associated with each attribute $A_i \in U$ is a set of constants called the *domain* of A_i or dom (A_i). A *tuple* t over $R_i = \{A_1, \cdots, A_m\}$ is an element of dom $(A_1) \times \cdots \times$ dom (A_m). A *relation* r_i over R_i is a set of tuples over R_i. A *database state* for a database scheme $\mathbf{R}$ is a function r that maps every relation scheme R_i in $\mathbf{R}$ to a relation on R_i; we write $r = \langle r_1, \cdots, r_k \rangle = \langle r(R_1), \cdots, r(R_k) \rangle$.

If t is a tuple over R_i and X is a subset of R_i, $t[X]$ is the *restriction* of t to the attributes X. If r_i is a relation over R_i, the *projection* of r_i onto X is

$$\pi_X(r_i) = \{t[X] \mid t \in r_i\}.$$

Given a relation I defined on U and a database scheme $\mathbf{R} = \{R_1, \cdots, R_k\}$, $\pi_{\mathbf{R}}(I) = \langle \pi_{R_1}(I), \cdots, \pi_{R_k}(I) \rangle$.

2.2. Tableaux. Tableaux were originally proposed by Aho, Sagiv, and Ullman to represent relational expressions [ASU]. A *tableau* consists of a *body* and a possibly empty *summary row*. The body of a tableau is a relation over $U' = U \cup \{TAG\}$. Each tuple in the body is simply called a *row*. The tableau domain of $A_i \in U$, tdom (A_i), is the disjoint union of dom (A_i), the set $\{a_i\}$, where a_i is called the *distinguished variable* (dv) for A_i, and a countable set Ndv (A_i) of *nondistinguished variable* (ndv's) for A_i. The tag domain tdom $(TAG) = \mathbf{R} \cup \{\mathbf{U}\}$. The elements of tdom (A_i), for all $A_i \in U$, are ordered by a partial order $\ll$ such that we have the following:

- all elements of dom (A_i) are pairwise incomparable.
- $c \ll v$, for $c \in$ dom (A_i), $v \in$ tdom $(A_i) -$ dom (A_i).
- $a \ll b$, for a the dv for A_i, $b \in$ Ndv (A_i).
- Ndv (A_i) is a linear order set under $\ll$.

The summary row of a tableau is a tuple over a subset of U called the *target relation scheme.* Where it is defined, the summary row may contain only dv's and constants that appear in the body of the tableau. Let $\mathbf{R}=\{R_1, \cdots, R_k\}$. The tableau for $\mathbf{R}$, denoted by $T_{\mathbf{R}}$, is a tableau of k rows $t_1, \cdots, t_k$ such that $t_i[A]$ is a dv exactly when $A \in R_i$ and distinct ndv otherwise, for all $1 \leqq i \leqq k$. The summary row of $T_{\mathbf{R}}$ is a tuple of all dv's $\langle a_1, \cdots, a_n \rangle$.

Given a database state $r=\langle r_1, \cdots, r_k \rangle$, we define a tableau T_r on $U \cup \{TAG\}$ and call it *the tableau for database state r.* For each relation $r_i \in r$, and for each tuple $t \in r_i$, there is a row u in T_r corresponding to it. The tuple u is said to *originate* from r_i or R_i and is defined as follows:

- $u[R_i]=t$.
- $u[A]=b_{ij}$, b_{ij} is an ndv that appears nowhere else in T_r, $A \in U-R_i$.
- $u[TAG]=R_i$.

The summary of T_r is empty.

2.3. Dependencies and chasing. The kinds of constraints considered are functional dependencies (fd's) and join dependencies (jd's). Associated with each fd or jd is an F-rule or J-rule, respectively. Given a tableau T and a set of fd's and jd's Σ, we can use the J-rules to infer additional tuples that must be in T if it is to satisfy Σ, and the F-rules to infer equalities among symbols of T for the same reason. These transformation rules are defined as follows and their properties are described in [MMS].

F-rule: For each fd $X \to A$, there is an F-rule corresponding to it. Suppose a tableau T has rows t_1, t_2 that agree in all X-columns. Let v_1, v_2 be the values in the A-column of t_1, t_2, respectively. Furthermore, $v_1 \neq v_2$. Applying the F-rule corresponding to $X \to A$ to rows t_1, t_2 of T yields a transformed tableau S. S is the same as T except v_1, v_2 are replaced as follows. If one of v_1 or v_2 is a constant (or a dv) and the other is not, then replace all occurrences of the other by the constant (or the dv, respectively). If both are ndv's, then replace all occurrences of the variable with the higher subscript by the variable with the lower subscript. If both are distinct constants, the result of applying the rule is usually defined to be the empty tableau and an *inconsistency* is said to be found.

J-rule: Let $\mathbf{S}=\{S_1, \cdots, S_k\}$, with the union of S_i's yielding U. Rows $t_1, \cdots, t_k$ of T (not necessarily distinct) are joinable on $\mathbf{S}$ if there exists a row w not in T that agrees with t_i on S_i, $1 \leqq i \leqq k$. Row w is the result of joining t_i's. An application of the J-rule corresponding to the jd $\|\times\|\mathbf{S}$ allows us to take rows $t_1, \cdots, t_k$ of T that are joinable on $\mathbf{S}$ and to add their result w to T. The added tuple is assumed to have $\mathbf{U}$ as its tag.

Suppose Σ is a set of fd's and/or jd's. $CHASE_{\Sigma}(T)$ is the tableau obtained by applying the F-rules and/or J-rules corresponding to the members of Σ exhaustively to T.

Given a set of dependencies Σ, there are additional dependencies implied by this set in the sense that any relation that satisfies this set must also satisfy the additional dependencies. The set of dependencies that is logically implied by Σ is the *closure* of Σ, denoted by Σ^+. Σ is said to be *equivalent* to a set of dependencies Ψ if $\Sigma^+=\Psi^+$. A set G of fd's is a *cover* of F if G is equivalent to F. Given a set of attributes X, the *closure* of X with respect to Σ, denoted by X_{Σ}^+, is the set of attributes $\{A \mid X \to A \in \Sigma^+\}$. We shall use X^+, instead of X_{Σ}^+ if Σ is understood. Let Σ be a set of fd's F. If $A \in X^+$, then there is a sequence of fd's $Y_1 \to A_1, \cdots, Y_n \to A_n = A$ such that $Y_i \to A_i \in F$, for all i.

Let Σ be a set of dependencies defined on a relation scheme R. An fd $X \to A \in \Sigma^+$ is *nontrivial* if $A \notin X$. A set of fd's is *nontrivial* if each of its members is. An fd $X \to A \in \Sigma^+$ is *left-reduced* if $X \to A$ is nontrivial and there is no proper subset Z of X with $Z \to A \in \Sigma^+$. A set of fd's is said to be *left-reduced* if each of its elements is. $K \subseteq R$ is a *candidate key* of R if $K \to R \in \Sigma^+$ and there is no proper subset of K that has this property. $X \subseteq R$ is a *superkey* of R if X contains a candidate key of R. A relation scheme R is said to be in BCNF with respect to Σ if whenever $X \to A \in \Sigma^+$ is nontrivial implies X is a superkey of R. A relation scheme R is *single-key* with respect to Σ if X and Y are candidate keys of R implies $X = Y$.

Let Σ be a set of dependencies on the universe U and let $\mathbf{R} = \{R_1, \cdots, R_k\}$ be a database scheme on U. $\Sigma^+ | R_i$ denotes the set of dependencies in Σ^+ that is defined on R_i, for some $R_i \in \mathbf{R}$. $\Sigma^+ | R_i$ is the set of *projected* dependencies onto R_i. $\mathbf{R}$ is said to be *single-key* with respect to Σ if each of its relation schemes R_i is single-key with respect to $\Sigma^+ | R_i$. $\mathbf{R}$ is in BCNF with respect to Σ if each $R_i \in \mathbf{R}$ is in BCNF with respect to $\Sigma^+ | R_i$. $\mathbf{R}$ is said to be *cover embedding with respect to a set* fd's F if there is a cover G of F such that every $g \in G$ is embedded in some relation scheme in $\mathbf{R}$. G is said to be an *embedded cover* of $\mathbf{R}$. $\mathbf{R}$ is said to be *lossless* with respect to Σ if $CHASE_\Sigma(T_\mathbf{R})$ has a row of dv's. $\mathbf{R}$ is said to be *dependency preserving* or to *preserve a set of* fd's F if for any relation I on U, I satisfies F implies $\pi_{R_1}(I) \| \times \| \cdots \| \times \| \pi_{R_k}(I)$ also satisfies F [BMSU]. It has been proved that $\mathbf{R}$ is cover embedding implies $\mathbf{R}$ is dependency preserving [BMSU].

2.4. Consistency of data in a database. Let r be a state for a database scheme $\mathbf{R} = \{R_1, \cdots, R_k\}$. Let I be an instance defined on U. Then I is a *weak instance* for r with respect to a set of dependencies Σ if

- $\pi_{R_i}(I) \supseteq r_i$, for each $1 \leqq i \leqq k$.
- I satisfies Σ.

Under the *weak instance model*, a database state r is said to be *consistent* with a set of dependencies Σ if a weak instance exists for the state with respect to Σ [GMV], [H], [V]. Otherwise r is *inconsistent* with respect to Σ. It has been shown that $CHASE_\Sigma(T_r)$ is nonempty if and only if r is a consistent state [GMV]. $CHASE_\Sigma(T_r)$ is called *the representative instance for state r.*

3. Insertion and deletion anomalies. In the Introduction, we have argued that whether a relation exhibits insertion and deletion anomalies depends on the types of partial tuples allowed in the relation. Following other authors [BMSU], [DGS1], [DGS2], [LP], [MW], [MU], [Sc], we need an additional concept to capture the idea of "basic facts" in a relation. We shall borrow the concept of *insertion sets* [LP] to denote basic facts we want to store in a relation. Insertion sets are similar to update sets in [BMSU], except that insertion sets are basic units of insertion and deletion, but not of update. So insertion sets are primarily associations or primary fact structures referenced elsewhere in the literature [DGS1], [DGS2], [MW]. We also assume each tuple in a relation represents an identifiable object in the real world. Codd suggested that an identifiable object in a relation is represented by a primary key value. Consequently Codd's assumptions on relational systems are as follows [Cod2], [Cod4]. A relation scheme R is assumed to have a candidate key designated as the primary key of R. A relational system is assumed to allow users to insert partial tuples into a relation with the restriction that the primary key cannot contain any null value. On the other hand, when a tuple is deleted, the whole tuple will be removed.

Most existing relational systems, for instance SQL-lookalike systems, support Codd's assumptions. However, several researchers [LP], [M] have felt the assumption

that a partial tuple inserted into a relation must have non-null values on the primary key is too restrictive. Instead, they proposed that this assumption should be relaxed by requiring inserted tuples to be non-null on a candidate key only.

These two different sets of assumptions lead to two different models for analyzing insertion and deletion anomalies and they are presented, respectively, in the following two sections. In both cases, a definition of anomalies-freedom is first given. We then address the question of how to obtain a desirable anomalies-free decomposition from an anomalous relation scheme.

Since insertion and deletion anomalies deal with the problem of whether some facts could be inserted and deleted independently in a relation, we shall consider this problem within the context of a relation scheme in the remainder of this section.

3.1. The case of non-null primary key values. In this section, we study the case that when a partial tuple is inserted into a relation R, it must be non-null on the primary key of R. We first define when a relation scheme is free of insertion and deletion anomalies. We then investigate how to find a decomposition that is free of insertion and deletion anomalies from a relation scheme.

3.1.1. A definition of anomalies-free schemes. In this section, we first define when a relation scheme is free of insertion and deletion anomalies under our assumptions. We then compare our approach with classical normalization theory by showing the condition under which normalization guarantees the resulting decomposition is free of insertion and deletion anomalies.

We have shown in the Introduction that independent facts cannot always be represented by constraints such as functional dependencies. To solve the problem of insertion and deletion anomalies formally, we need to define explicitly the meaning of independent facts. Let X be a subset of a relation scheme R. We call X an *insertion set* (on R) if X denotes a nondecomposable relationship by which data may be recorded. Several authors have studied the problem of how to find insertion sets [DGS1], [DGS2], [MW], [Sc], while others have addressed the question of how to convert a decomposable fact into a collection of insertion sets [B], [BDB]. In this paper, we assume insertion sets have been identified and we shall concentrate on how to find desirable decompositions that are free of the anomalies problem.

Informally, a relation scheme R is free of insertion anomalies if tuples on any insertion set on R can be inserted independently into R. R is free of deletion anomalies if no fact on an insertion set on R is "unexpectedly" deleted. Following LeDoux and Parker [LP], we capture these ideas formally as follows. A relation scheme R is *anomalies-free* if every insertion set on R contains the primary key of R. First we want to argue that R is anomalies-free exactly when R does not exhibit insertion and deletion anomalies illustrated in Example 1.

From this definition, R is anomalies-free if and only if R is free of insertion anomalies. To show that R is anomalies-free exactly when it is free of deletion anomalies, let us consider the following. If R is anomalies-free, then deleting a tuple from R will delete facts that contain the same primary key value. Under the assumption of this model, a primary key value of a tuple represents an object and the nonprimary attribute values are properties of the object. Therefore deleting a tuple results in deleting an object and its associated properties from the database. Hence no fact on an insertion set on R is "unexpectedly" deleted. On the other hand, if R is not anomalies-free, then R has an insertion set I defined on it which does not contain the primary key of R. Since the primary keys of the relation schemes represent objects in the real world, I is an insertion set which is different from any insertion set on R containing the

primary key. As a result, deleting a tuple on R will result in deleting a fact on I that is supposed to have independent existence from any insertion sets containing the primary key of R. So some fact on I is deleted "unexpectedly" when the last tuple containing the fact is deleted from R. Therefore in the sense of Codd [Cod2], R is not free of deletion anomalies. Hence R is anomalies-free exactly when R is free of deletion anomalies. This shows that our definition captures the essence of insertion and deletion anomalies. A database scheme or a decomposition **R** is *anomalies-free* if each of its relation schemes R_i is anomalies-free.

In this model, determining if a relation scheme R exhibits insertion and deletion anomalies is simple once the primary key and the insertion sets are identified. Although constraints such as fd's do not play an explicit role in this model, it is often the case that fd's convey some information about insertion sets, and this is assumed in the older literature [BMSU], [B], [Cod2]. So it would be interesting to see how this relates to our work. Under this model, the following result characterizes when a relation scheme is anomalies-free if every nontrivial fd is assumed to be an insertion set on a relation scheme.

THEOREM 1. *Let R and Σ be a relation scheme and a set of constraints on R, respectively. Suppose every nontrivial* fd *represents an insertion set on R. R is anomalies-free if and only if for every nontrivial* fd $X \to A \in \Sigma^+$, *X does not contain the primary key implies $XA = R$.*

Proof. This theorem is proved by showing that every nontrivial fd embeds the primary key of R if and only if for every nontrivial fd $X \to A \in \Sigma^+$ such that X does not contain the primary key implies $XA = R$.

"If." The proof is trivial.

"Only if." Assume there exists some nontrivial fd $X \to A \in \Sigma^+$ such that X does not contain the primary key and $R - XA \neq \emptyset$. If XA does not contain the primary key, what we want to prove follows trivially. If XA embeds the primary key, then A is a primary key attribute since X does not contain the primary key. Since $R - XA \neq \emptyset$, let $B \in R - XA$. Since XA contains the primary key, $X \to B \in \Sigma^+$. By assumption, X does not embed the primary key and A is an attribute of the primary key, $X \to B \in \Sigma^+$ is a nontrivial fd that does not contain the primary key. Therefore in all cases, there is a nontrivial fd that does not embed the primary key of R. □

In many cases, an fd in a nontrivial cover denotes an insertion set [BMSU]: the following result characterizes when R is anomalies-free under such an assumption.

THEOREM 2. *Let R and F be a relation scheme and a set of nontrivial* fd's *on R, respectively. If every* fd $X \to A \in F$ *represents an insertion set on R, then R is anomalies-free if and only if for every $X \to A \in F$, XA embeds the primary key of R.*

Proof. The proof is trivial. □

From Theorems 1 and 2, if we assume nontrivial fd's denote insertion sets on a relation scheme, then the classes of anomalies-free relation schemes are subclasses of BCNF relation schemes.

3.1.2. Synthesizing anomalies-free decompositions by grouping. From § 3.1.1, there is a simple test to determine if a relation scheme is anomalies-free once the primary key and its insertion sets are identified. In this section, we consider the problem of constructing an anomalies-free decomposition from a relation scheme R using the insertion sets. We implicitly assume that candidate keys of insertion sets are given and we are free to choose any candidate key to be the primary key in the design of anomalies-free decompositions. The technique proposed here is to group together insertion sets with a common candidate key.

Given a relation scheme R and associated insertion sets D, then clearly D forms an anomalies-free decomposition of R. However, there are cases where such a design is poor. For example, suppose we have insertion sets AB and ABC, both with key A. Then all we really need is a relation over ABC. Similarly, if we have insertion sets AB and AC both with key A, it makes sense to combine them into a single relation ABC. Thus our goal is to decompose an anomalous relation scheme R into a *minimal* collection of anomalies-free schemes.

Minimality and anomalies-freedom are important criteria for a decomposition, as is the property of preserving insertion sets. An insertion set I is *preserved* in a decomposition if I is defined on some relation scheme. However, if I is defined on more than one relation scheme, then whenever tuples on I are inserted into the database, more than one relation is updated. Storing duplicate data at the conceptual level is not desirable in general; hence each insertion set should be defined on exactly one relation scheme in the decomposition.

D is an *insertion-set-preserving* decomposition of R if every insertion set on R is defined on exactly one element in **D**. A decomposition **D** of R is *minimal-insertion-set-preserving* if for all decompositions **E** of R with anomalies-free and insertion-set-preserving properties, $|\mathbf{D}| \leqq |\mathbf{E}|$. It is worth mentioning that the cardinality of a minimal-insertion-set-preserving decomposition of R is one if and only if R is anomalies-free.

In view of the desirability of minimal-insertion-set-preserving decompositions, it is interesting to know if a minimal-insertion-set-preserving decomposition **D** of R can be found efficiently. A straightforward solution is to identify all subsets S of candidate keys of the insertion sets such that every insertion set has at least one candidate key in the subset S. The candidate keys in such subset S partition the insertion sets into groups such that insertion sets in each group share a common candidate key. The union of elements in each group forms a relation scheme in an anomalies-free decomposition with the common candidate key designated as the primary key of the relation scheme. Each insertion set is defined on the relation scheme corresponding to the partition in which it is included. It is easy to verify that a decomposition formed this way is anomalies-free and insertion-set-preserving. We call this technique of finding minimal-insertion-set-preserving decompositions the *grouping* technique. Having found all such subsets, we select a subset S with minimal cardinality and the decomposition resulting from S is a minimal-insertion-set-preserving decomposition of R.

Example 3. Let $R(A, B, C, D)$ and the insertion sets be AB, AC, BC, and BD. Suppose every attribute functionally determines the others and therefore each attribute is a candidate key. Then a minimal subset of candidate keys such that the subset contains at least one candidate key from each insertion set is $S = \{B, C\}$. A partition induced by S is $\{AB, BD\}$ and $\{AC, BC\}$. Note that the partition induced is not unique. For instance, $\{AB, BD, BC\}$ and $\{AC\}$ is another partition induced by S. The resulting anomalies-free relation schemes from the former partition are ABD and ABC, with B and C as the primary keys of the relation schemes, respectively. The sets of insertion sets $\{AB, BD\}$ and $\{AC, BC\}$ are assumed to be defined on ABD and ABC, respectively. It is easy to see that $\{ABD, ABC\}$ is a minimal-insertion-set-preserving decomposition of R. If the partition $\{AB, BD, BC\}$ and $\{AC\}$ is used instead, the resulting minimal-insertion-set-preserving decomposition would be $\{ABCD, AC\}$. □

The above method of finding minimal-insertion-set-preserving decompositions requires exhaustively considering all subsets of candidate keys of the insertion sets; hence it may take exponential time. Unfortunately, there does not seem to be a polynomial-time algorithm for finding a minimal-insertion-set-preserving decomposi-

tion in general even if we assume the set of candidate keys can be found for each insertion set efficiently. This fact follows from the following theorem.

THEOREM 3. *Given a database scheme* $\mathbf{D} = \{\langle R_i, K_i\rangle \mid R_i$ *is a relation scheme and* K_i *is the set of candidate keys of* $R_i\}$. *Then the problem "Does there exist a subset* S *of* $\bigcup_i K_i$ *with size* n *such that for each* K_i, S *contains at least one element from* K_i?" *is* NP-*complete.*

Proof. The problem is obviously in NP, since we have only to select nondeterministically a subset S of size n from $\bigcup_i K_i$ and verify that S contains at least one element from each K_i. Clearly, the verification process can be done efficiently. We now present a polynomial-time reduction of a known NP-complete problem called the hitting set to our problem. The hitting set problem is formulated as follows. Given a family $\{V_1, \cdots, V_q\}$ subsets of $T = \{t_1, \cdots, t_p\}$, we must decide if there exists a subset W of T of size n such that for each V_i, W contains at least one element from V_i. Such a set is called a *hitting set of size n.* This problem can be found in [GJ] and was proved to be NP-complete in [K]. We now construct a polynomial-time algorithm that maps each instance of the hitting set problem to a corresponding instance of our problem.

Let T be the universe. For each i, $R_i = V_i$ and $K_i = V_i$. In other words, every attribute in a relation scheme is a candidate key of the relation scheme. This can easily be constructed by assuming each attribute in T functionally determines all other attributes. We want to show that $\mathbf{D}$ has a subset S of $\bigcup_i K_i$ with size n such that for each K_i, S contains at least one element from K_i if and only if T contains a hitting set of size n.

"If." Let W be a hitting set of size n. Let $S = W$. We want to show that S contains at least one element from each K_i. Since W is a hitting set, for each V_i, W contains at least one element from V_i. Since $V_i = K_i$, S contains at least one element from each K_i.

"Only if." Assuming that there is a subset S of $\bigcup_i K_i$ of size n such that for each K_i, S contains at least one element from K_i. Let $W = S$. Since $K_i = V_i$, W contains at least one element from each V_i. So W is a hitting set of size n. □

So given a set of insertion sets on R, even if we can find the candidate keys of each insertion set efficiently, it is highly unlikely that there is an efficient algorithm for finding a minimal-insertion-set-preserving decomposition of R. This is because solving this problem efficiently necessarily implies the problem in Theorem 3 can be solved efficiently. It is worth noting that the key-finding problem has also been proved to be NP-hard [BB]. In fact, we do not even know if the problem of finding a minimal-insertion-set-preserving decomposition is in NP.

COROLLARY 1. *Let* $\mathbf{S} = \{\langle S_i, K_i\rangle \mid S_i$ *is an insertion set and* K_i *is the set of candidate keys of* $S_i\}$ *be the set of insertion sets for a relation scheme* R. *Then given* $\mathbf{S}$, *the problem of finding a minimal-insertion-set-preserving decomposition of* R *is* NP-*hard.*

Proof. The proof follows directly from the above argument. □

Up to now, we only require an anomalies-free decomposition of R to be minimal. There are some other properties we may want a decomposition to have. For example, if we assume it is meaningful to have a tuple defined on the original relation scheme R, we may want the minimal-insertion-set-preserving decomposition to be lossless [ABU]. In view of existence of nonunique minimal-insertion-set-preserving decompositions, it is interesting to know if every such decomposition is equally desirable with respect to the losslessness criterion of the original scheme R. It should be clear that it is not always possible to construct a lossless decomposition from the insertion sets. For example, if the constraints are fd's and the insertion sets on R are disjoint, then

any insertion-set-preserving decomposition is lossy. So the question is: under what condition(s) do we have a lossless decomposition of R. It turns out there is a simple answer if the decomposition is anomalies-free and insertion-set-preserving.

THEOREM 4. *Let* **S** *be a set of insertion sets on a relation scheme* R, *and let* **D** *be any insertion-set-preserving and anomalies-free decomposition of* R. *Let* Σ *be the set of constraints on* R. **D** *is lossless with respect to* Σ *if and only if* **S** *is lossless with respect to* Σ.

Proof. First observe that since **D** is anomalies-free and insertion-set-preserving, every element D_i in **D** is a union of elements in **S** that share a common candidate key. So we can chase the tableau $T_{\mathbf{S}}$ to give an intermediate tableau that is equivalent to $T_{\mathbf{D}}$. It follows that $\Sigma \models \|\times\|\mathbf{R}$ if and only if $\Sigma \models \|\times\|\mathbf{S}$. □

Theorem 4 says that whether an insertion-set-preserving and anomalies-free decomposition is lossless depends solely on the given insertion sets. So any insertion-set-preserving and anomalies-free decomposition is equally desirable with respect to the losslessness criterion of R.

3.2. The case of non-null values on a candidate key. In § 3.1, an inserted tuple is assumed to have nonnull values on the primary key of the relation scheme. This basically is the assumption made by Codd on relational systems [Cod2], [Cod4]. Some authors [LP], [M] have felt that this assumption on an inserted tuple is too restrictive and they proposed an inserted tuple should be non-null on a candidate key only. This less restrictive assumption still allows tuples in a relation to be uniquely identified by a candidate key and yet this is more flexible than Codd's assumption with respect to tuple insertions. We study the insertion and deletion anomalies problem under this assumption in this section. We first give a definition of anomalies-free schemes. We then characterize when a relation scheme is anomalies-free if every functional relationship is assumed to be an insertion set on a relation scheme. Then, we investigate how a minimal-insertion-set-preserving decomposition is obtained from a relation scheme.

3.2.1. A definition of anomalies-free schemes. Since we now assume every tuple inserted into a relation must be non-null on a candidate key, we implicitly assume that candidate keys in a relation scheme are semantically equivalent and that they all represent the same object the relation is supposed to model. For example, assume the attributes *SIN* and *Emp-no* are both candidate keys in the EMP relation. Then, either *SIN* or *Emp-no* could be used to represent an employee in the company. Therefore, as long as an inserted tuple on EMP is nonnull on one of these two attributes, it is legal to insert it into the relation.

Using an analysis similar to the one given in § 3.1.1, the following definition captures the essence of anomalies-freedom. A relation scheme R is *anomalies-free* if every insertion set on R contains a candidate key of R. With this definition, the following result characterizes when a relation scheme is anomalies-free if every functional relationship on a relation scheme is assumed to be an insertion set.

THEOREM 5. *Let* R *and* Σ *be a relation scheme and a set of dependencies on* R, *respectively. Suppose every nontrivial* fd *on* R *represents an insertion set on* R. *Then* R *is anomalies-free if and only if* R *is in* BCNF *with respect to* Σ.

Proof. R is anomalies-free if and only if every nontrivial fd $X \to A \in \Sigma^+$ embeds a candidate key of R. Every nontrivial fd $X \to A \in \Sigma^+$ embeds a candidate key of R if and only if every nontrivial fd $X \to A \in \Sigma^+$ implies X contains a candidate key of R, and hence is a superkey of R. □

If every fd in a nontrivial cover F is assumed to represent an insertion set, the following characterizes when R is anomalies-free.

THEOREM 6. *Let R and F be a relation scheme and a set of nontrivial* fd*'s on R, respectively. Suppose every $X \to A \in F$ represents an insertion set on R. Then R is anomalies-free if and only if R is* BCNF *with respect to F.*

Proof. This follows from the fact that R is BCNF with respect to F exactly when every fd $X \to A \in F$ embeds a superkey of R. □

It is interesting to note that under a different model, Bernstein and Goodman [BG] have shown that BCNF relation schemes are exactly the class of anomalies-free schemes when fd's are given as constraints.

3.2.2. Synthesizing anomalies-free decompositions by merging. Suppose after the set of insertion sets on a relation scheme R is identified, R exhibits insertion and deletion anomalies. Then, as was argued in § 3.1.2, we would like to find a minimal-insertion-set-preserving decomposition for R. Notice that the definition of minimal-insertion-set-preserving decomposition is the same as the one given in § 3.1.2, except that the definition of anomalies-freedom is the one given in § 3.2.1. Because the definition of anomalies-frredom in § 3.1.1 implies the definition of anomalies-freedom in this section (but not vice versa), any insertion-set-preserving and anomalies-free decomposition under the previous model is an insertion-set-preserving and anomalies-free decomposition in this model. However, the converse does not hold.

Example 4. Let $R(A, B, C, D)$ and let the insertion sets be AB, ABC, CD with fd's $\{A \to BC, C \to AD\}$. The attributes A and C are the candidate keys of the insertion sets. Under the previous model, AB and ABC could be grouped together and a minimal-insertion-set-preserving decomposition is $\{ABC, CD\}$ with A and C as the primary keys in the corresponding relation schemes. Under the current model, $\{ABC, CD\}$ is an anomalies-free and insertion-set-preserving decomposition. The minimal-insertion-set-preserving decomposition is R, which clearly is not anomalies-free under the previous model. □

Example 5. Let $R(A, B, C, D)$, $F = \{A \to B, B \to C, C \to BD\}$ and the insertion sets be AB, BC, and CD. The only candidate key of R is A and since BC and CD do not contain A, R has the insertion and deletion anomalies problem. The decomposition that consists of AB (with candidate key A) and BCD (with candidate keys B and C) is a minimal-insertion-set-preserving decomposition of R. The sets of insertion sets $\{AB\}$ and $\{BC, CD\}$ are assumed to be defined on AB and BCD, respectively. □

It turns out that a minimal-insertion-set-preserving decomposition in this model can be found easily if the closure of a set of attributes with respect to Σ can be computed efficiently. The following algorithm is a method for obtaining such a decomposition and it is obtained by merging insertion sets.

ALGORITHM Merge.
Input: R, Σ and the insertion sets $I = \{I_1, \cdots, I_k\}$ on R.
Output: A minimal-insertion-set-preserving decomposition of R.
Method:
(1) Compute I_j^+ with respect to Σ, for all $1 \leqq j \leqq k$.
(2) Define a binary relation $\approx$ on I as follows. $I_i \approx I_j$ if $I_i^+ = I_j^+$, for all I_i and I_j in I. Clearly the relation $\approx$ is an equivalence relation. Let $[I_i]$ be the equivalence class of I_i with respect to $\approx$. Then $\cup [I_i]$ is a relation scheme in the decomposition. The insertion set I_i is assumed to be defined on $\cup [I_j]$.
(3) Output $\mathbf{D} = \{\cup [I_j] \mid I_j \in I\}$.

By step (2), the decomposition **D** output preserves insertion sets and is anomalies-free. **D** is insertion-set-preserving since the equivalence partitions I. **D** is anomalies-free

since each insertion set embeds a candidate key of the relation scheme on which it is defined. Let $\mathbf{S} = \{S_1, \cdots, S_m\}$ be a minimal-insertion-set-preserving decomposition for R. We want to show $\mathbf{D} = \mathbf{S}$. Let S_{j_1} and S_{j_2} be two insertion sets on S_j, for some $1 \leqq j \leqq m$. By the definition of anomalies-freedom, $S_{j_1}^+ = S_{j_2}^+$. Since $\mathbf{S}$ preserves insertion sets, let $I_i \in I$ be defined on S_j, for some $1 \leqq j \leqq m$. Then, elements in $[I_i]$ are defined on S_j, otherwise $\mathbf{S}$ is not minimal. By the definition of $[I_i]$ and the fact that $S_{j_1}^+ = S_{j_2}^+$, for all insertion sets S_{j_1} and S_{j_2} defined on S_j, it follows that $[I_i]$ is exactly the set of insertion sets defined on S_j. This shows that $\mathbf{D} = \mathbf{S}$. Therefore the minimal-insertion-set-preserving decomposition is unique. Once the insertion sets on a relation scheme are identified, whether the minimal-insertion-set-preserving decomposition can be obtained efficiently depends on the time complexity for computing the closure of a set of attributes with respect to Σ.

THEOREM 7. *The algorithm Merge correctly obtains a minimal-insertion-set-preserving decomposition for R and the decomposition is unique for R given the input.*

Proof. The proof follows directly from the above analysis. □

It is worth noting that relation schemes in a decomposition produced by Bernstein's synthesizing algorithm are anomalies-free if each embedded key dependency of a relation scheme denotes an insertion set on the relation scheme. Hence the synthesizing algorithm could be used in converting an anomalous relation scheme into an anomalies-free decomposition.

4. Replacement anomalies. Example 1 illustrated replacement anomalies, as conceived by Codd [Cod2]. In that example, a relation exhibits replacement anomalies if modification of a tuple on some attribute causes some constraints to be violated. Whether or not a relation exhibits replacement anomalies depends on its constraints as well as the set of attributes allowed to be modified in the relation. An attribute in a relation scheme that is allowed to be modified is said to be *updatable.*

We first give a definition that captures the essence of the replacement anomaly problem as illustrated in Example 1. Then we argue that unlike insertion and deletion anomalies, it is not meaningful to discuss replacement anomalies in the context of a single relation. Instead, this problem should be addressed in the context of a database scheme. As we will show later, the replacement anomaly problem is closely related to the constraint enforcement problem. We then give a more intuitively correct definition of replacement-anomaly-frredom. By varying the assumptions on the updatable attributes, several important classes of replacement-anomaly-free database schemes are characterized.

4.1. A definition of replacement-anomaly-free schemes. From Example 1, a relation scheme R that exhibits replacement anomalies implies that when some value of a tuple on R is changed, other tuples in the relation may be modified on the updated attribute. Consequently, an update on a tuple results in an unpredictable number of tuples being retrieved and changed. The following definition captures the essence of replacement-anomaly-freedom when a relation is considered. A relation scheme R is *replacement-anomaly-free* with respect to Σ if the resulting relation from a modification of a tuple in a satisfying relation is also satisfying with respect to Σ.

However, just requiring each relation to satisfy its projected or local dependencies is inadequate.

Example 6. Let $\mathbf{R} = \{R_1(\textit{Supplier}, \textit{City}), R_2(\textit{Supplier}, \textit{Status})\}$, and $F = \{\textit{Supplier} \rightarrow \textit{City}, \textit{City} \rightarrow \textit{Status}\}$. The *Supplier* attribute is the primary key of both relations. Suppose the only updatable attributes for R_1 and R_2 are *City* and *Status*, respectively. Because of the updatable attributes and since each relation is in BCNF, each relation scheme

is replacement-anomaly-free with respect to its projected fd's. Let us consider the following state:

Supplier	City	Supplier	Status
s_1	c	s_1	t_1
s_2	c	s_2	t_1

The given state is consistent with respect to the given fd's [GMV], [H]. Suppose now we change the status of supplier "s_1" located in city "c" from "t_1" to "t_2." To maintain the consistency of the state, all "t_1" values in r_2 should be changed to "t_2." Since more than one tuple is retrieved and updated, this database scheme is not free of replacement anomalies. □

Intuitively, a database scheme **R** is replacement-anomaly-free with respect to Σ if every update on a consistent state is legal and the database system is not required to retrieve any tuple from the state as a result of the update. In view of this, we now give a more appropriate definition that captures the intuition on replacement-anomaly-freedom.

A database scheme **R** is *replacement-anomaly-free* with respect to Σ if for every consistent state r, whenever a tuple t_i on some $R_i \in \mathbf{R}$ is modified on some updatable attribute A, the resulting state is also consistent with respect to Σ. So if a scheme is replacement-anomaly-free, then nothing needs to be done by the system with respect to consistency whenever an update is performed on a consistent state. In the remainder of this section, we first study some properties of replacement-anomaly-free schemes, we then characterize several important classes of replacement-anomaly-free database schemes.

4.2. Properties and characterizations of replacement-anomaly-free schemes.

4.2.1. Some necessary conditions and properties for replacement-anomaly-free schemes. In this section, we first identify some necessary conditions for a database scheme to be replacement-anomaly-free when a set of dependencies is given. In view of the importance of fd's and the full jd $\|\times\|\mathbf{R}$ [FMU], we then describe a condition under which the augmentation of fd's with the full jd $\|\times\|\mathbf{R}$ does not change the characterization of replacement-anomaly-free schemes.

A database scheme **R** is said to satisfy the *nondetermining and key-determined* condition with respect to Σ if for every $R_i \in \mathbf{R}$, and for every left-reduced fd $X \to A \in \Sigma^+ \mid R_i$, $B \in XA$ is updatable implies $A = B$, X is a candidate key of R_i and there is no other R_j that embeds XA, $i \neq j$. The following two examples illustrate why the above condition is necessary for replacement-anomaly-freedom.

Example 7. Let $\mathbf{R} = \{S(Project, Part, Cost)\}$ and $F = \{Part \to Cost\}$. The relation scheme S tells us what parts are used in which projects and at what prices. Suppose the updatable attribute is *Part*, we claim that **R** is not replacement-anomaly-free with respect to F. Consider the following consistent state $r = \langle s = \{t_1 = \langle pj_1, p_1, c_1\rangle, t_2 = \langle pj_2, p_2, c_2\rangle\}\rangle$. Suppose we change $t_1[Part]$ from p_1 to p_2, then the update gives rise to a violation of the fd $Part \to Cost$. On the other hand, if *Cost* is the only updatable attribute in **R**, then **R** still has replacement anomalies. Let us consider the following consistent state $r = \langle s = \{t_1 = \langle pj_1, p_1, c_1\rangle, t_2 = \langle pj_2, p_1, c_1\rangle\}\rangle$. If $t_1[Cost]$ is changed from c_1 to c_2, then the update maps r into an inconsistent state. This example illustrates that if **R** is replacement-anomaly-free, then for every nontrivial fd $X \to A \in \Sigma^+ \mid R_i$, $B \in XA$ and B is updatable necessarily implies $B = A$ and X is a candidate key of R_i. □

Example 8. Let $\mathbf{R} = \{S(Project, Part, Cost), T(Part, Cost)\}$ and $F = \{Part \rightarrow Cost\}$. Let us assume *Cost* in T is the only updatable attribute. Note that the attribute *Cost* is the right-hand side of the fd $Part \rightarrow Cost$ and *Part* is a candidate key of T. However, $\mathbf{R}$ is not replacement-anomaly-free since the relationship *Part_Cost* is also embedded in S and therefore we can easily construct a consistent state on $\mathbf{R}$ such that changing the *Cost*-component of some tuple in T will map the state into a state that violates the fd $Part \rightarrow Cost$. □

The following theorem proves that the nondetermining and key-determined condition is necessary for replacement-anomaly-freedom.

THEOREM 8. *Let* Σ *be a set of dependencies on* $\mathbf{R}$. *If* $\mathbf{R}$ *does not satisfy the nondetermining and key-determined condition with respect to* Σ, *then* $\mathbf{R}$ *is not replacement-anomaly-free with respect to* Σ.

Proof. If $\mathbf{R}$ violates the condition, then there is a $R_i \in \mathbf{R}$ that embeds a left-reduced fd $X \rightarrow A \in \Sigma^+$ and $B \in XA$ is updatable but $A \neq B$, or X is not a candidate key of R_i, or XA is embedded in some other relation scheme R_j. There are three possible cases to be considered.

Case 1. $A \neq B$. That is, there is some updatable attribute $B \in X$ in R_i. Let $I = \{t_1, t_2\}$ be such that t_1 and t_2 agree exactly on $\{X - \{B\}\}^+$ and distinct constants otherwise. Then chase I with respect to Σ to obtain $CHASE_\Sigma(I)$. By considering $CHASE_\Sigma(I)$ as an instance, $CHASE_\Sigma(I)$ is a satisfying relation with respect to Σ [BV], [MMS]. Since Σ does not imply the fd $(X - \{B\}) \rightarrow A$, t_1 and t_2 in $CHASE_\Sigma(I)$ disagree on A and B but agree on $(X - \{B\})$. Let $r = \pi_{\mathbf{R}}(CHASE_\Sigma(I))$. The state r is consistent with respect to Σ since $CHASE_\Sigma(I)$ is a weak instance for r. Note that $t_1[R_i]$ and $t_2[R_i]$ are two tuples in $r_i \in r$ and they agree on $(X - \{B\})$ but disagree on A and B. Since by assumption B is updatable in R_i, we change the B-component of $t_1[R_i]$ to the B-component of $t_2[R_i]$. Hence the tuples $t_1[R_i]$ and $t_2[R_i]$ in r_i violate the fd $X \rightarrow A \in \Sigma^+ \mid R_i$. This shows that $\mathbf{R}$ is not replacement-anomaly-free with respect to Σ.

Case 2. $A = B$ and X is not a candidate key of R_i. Let $I = \{t_1, t_2\}$ be such that t_1 and t_2 agree exactly on X^+ and distinct constants otherwise. As in Case 1, we first obtain $CHASE_\Sigma(I)$. By considering $CHASE_\Sigma(I)$ as a satisfying instance, let $r = \pi_{\mathbf{R}}(CHASE_\Sigma(I))$ and r is a consistent state with respect to Σ. By assumption, X is not a candidate key of R_i. This implies $t_1[R_i]$ and $t_2[R_i]$ are two distinct tuples in $r_i \in r$. By the construction of $CHASE_\Sigma(I)$, $t_1[R_i]$ and $t_2[R_i]$ agree on X. Since A is updatable in R_i, change the A-component of $t_1[R_i]$ to a constant that appears nowhere else. The resulting relation violates $X \rightarrow A \in \Sigma^+ \mid R_i$ and therefore the updated state is inconsistent with respect to Σ. Hence $\mathbf{R}$ is not replacement-anomaly-free with respect to Σ.

Case 3. $A = B$ and XA is embedded in some other relation scheme R_j. Let r be the consistent state constructed in Case 2. Since XA is embedded in both R_i and R_j, $t_1[R_i]$ and $t_1[R_j]$ are tuples in r_i and r_j, respectively, and they agree on XA. Since A is updatable in R_i, change the A-component of $t_1[R_i]$ to a constant that appears nowhere else. Clearly the state is inconsistent with respect to Σ. Hence $\mathbf{R}$ is not replacement-anomaly-free with respect to Σ. □

The following example illustrates another necessary condition for replacement-anomaly-freedom.

Example 9. Let $\mathbf{R} = \{R_1(Patient, Hospital), R_2(Patient, Doctor), R_3(Doctor, Hospital)\}$, and $F = \{Patient \rightarrow Hospital\ Doctor,\ Doctor \rightarrow Hospital\}$. R_1 records the registration information, and each patient registers in a unique hospital. R_2 stores the in-charge-of relationship, and each patient is assigned to a unique doctor. R_3 tells the

affiliation information, and each doctor is assumed to be associated with at most one hospital.

First observe that R_2^+ with respect to $\{Doctor \rightarrow Hospital\}$ contains R_1, and hence there are two different derivations of the functional relationship between *Patient* and *Hospital*. Let $r = \langle r_1 = \{\langle p, h\rangle\}, r_2 = \{\langle p, d\rangle\}, r_3 = \{\langle d, h\rangle\}\rangle$ be a consistent state on $\mathbf{R}$. If the attribute *Hospital* in R_1 is updatable, then the state r can be transformed into an inconsistent state by changing the *Hospital*-component of the tuple in r_1 to some other constant. This shows that $\mathbf{R}$ is not replacement-anomaly-free with respect to F. □

The following theorem formally states a necessary condition for replacement-anomaly-freedom illustrated in the example above.

THEOREM 9. *Let F be a set of* fd's. *Let $Z_1 \rightarrow A_1, \cdots, Z_n \rightarrow A_n$ be a sequence of* fd's *used in computing (partially or totally) the closure of $R_i \in \mathbf{R}$ such that each $Z_j \rightarrow A_j \in F^+$ and is embedded in some $R_{i_j} \in \mathbf{R}$. Suppose $R_i \cup Z_1A_1 \cup \cdots \cup Z_nA_n$ contains XB, where $X \rightarrow B \in F^+$ is a left-reduced* fd *embedded in some $R_p \in \mathbf{R}$, $p \neq i$. Moreover, for all $1 \leqq j \leqq n$, Z_jA_j embedded in $R_{i_j} = R_p$ implies Z_j is not a superkey of R_p. If some attribute of XB in R_p is updatable, then $\mathbf{R}$ is not replacement-anomaly-free with respect to F.*

Proof. By assumption that $X \rightarrow B$ is left-reduced, if X is not a candidate key of R_p or B is not the only updatable attribute of XB in R_p, then by Theorem 8, $\mathbf{R}$ is not replacement-anomaly-free with respect to F. In the remainder of this proof, we assume X is a candidate key of R_p and B is the only updatable attribute of XB in R_p.

We first construct a tagged satisfying universal relation T with $t_0, \cdots, t_{n+1}$ as its rows as follows: t_0 has zero exactly in R_i^+ with tag R_i. For each $Z_j \rightarrow A_j$, $1 \leqq j \leqq n$, there is exactly one row t_j in T with tag R_{i_j}. The tuple t_j has the constant zero exactly in Z_j^+ and distinct constants that otherwise appear nowhere else. The tuple t_{n+1} with tag R_p has zero exactly in X^+ and distinct constants that appear nowhere else otherwise. It can be shown that $\pi_U(T)$ is a satisfying relation with respect to F [GY].

Let us construct a consistent state r from T as follows. For each $R_k \in \mathbf{R}$, $r_k = \{t[R_k] \mid t \in T \text{ and } t[Tag] = R_k\}$. Since $\pi_U(T)$ is a satisfying relation, and hence a weak instance for r, r is consistent with respect to F. Next we want to show that r can be mapped into an inconsistent state with a modification on a tuple. First observe that by assumption on Z_jA_j, $t_{n+1}[R_p] \neq t_j[R_p]$, for any Z_jA_j embedded in R_p. By construction of r, $t_{n+1}[R_p] \in r_p$. Since B is updatable in R_p, change the B-component of $t_{n+1}[R_p]$ from zero to a constant w that appears nowhere else. Let s be the updated state. By assumption, $Z_1 \rightarrow A_1, \cdots, Z_n \rightarrow A_n$ is a sequence of fd's used in computing the closure of R_i and the closure contains XB, it is easy to see that during the chase process the tuple in the state tableau corresponding to $t_0[R_i]$ in s has the constants zero and w simultaneously assigned to $t_0[B]$. Hence s is not consistent with respect to F. Therefore $\mathbf{R}$ is not replacement-anomaly-free with respect to F. □

The following theorem describes a condition under which the augmentation of fd's with the full jd does not change the characterization of replacement-anomaly-free schemes. We need a result from [CM] before we give the theorem.

THEOREM 10. *Let $\mathbf{R}$ be dependency preserving with respect to F and $\Sigma = F \cup \{\|\times\|\mathbf{R}\}$. Let r be a state on $\mathbf{R}$. Then r is consistent with respect to F if and only if r is consistent with respect to Σ.*

Proof. See [CM] for the proof. □

THEOREM 11. *Let $\mathbf{R}$ be dependency preserving with respect to F and $\Sigma = F \cup \{\|\times\|\mathbf{R}\}$. Let Z_i be the set of updatable attributes in R_i, for all $R_i \in \mathbf{R}$. Then $\mathbf{R}$ is replacement-anomaly-free with respect to F if and only if $\mathbf{R}$ is replacement-anomaly-free with respect to Σ.*

Proof. "If." Let r be a consistent state with respect to F. By Theorem 10, r is consistent with respect to Σ. Suppose s is the state after a tuple in r has been updated

on some updatable attribute $A \in Z_i$, for some $R_i \in \mathbf{R}$. Since $\mathbf{R}$ is replacement-anomaly-free with respect to Σ, the updated state s is consistent with respect to Σ. By Theorem 10, s is consistent with respect to F. Hence $\mathbf{R}$ is replacement-anomaly-free with respect to F.

"Only if." Using an argument similar to the "if" part, it is easy to show that this implication also holds. □

4.2.2. Independent and γ-acyclic cover embedding BCNF schemes. In this section, we characterize two important classes of replacement-anomaly-free schemes. We show this result by proving that the nondetermining and key-determined condition is also sufficient for replacement-anomaly-freedom when the class of independent schemes and the class of γ-acyclic cover embedding BCNF schemes are considered.

Constraint enforcement is an important function in any database system. A class of database schemes known as independent schemes was proposed to allow efficient enforcement of constraints imposed on a database. Independent schemes were proposed independently by several researchers [GY], [IIK], [S1], [S3] and have been shown to be desirable with respect to query answering [AC], [IIK], [S3]. Let $\Sigma = F \cup \{\|\times\|\mathbf{R}\}$, where F is a set of fd's. A database scheme $\mathbf{R}$ is *independent with respect to* Σ if ensuring every relation $r_i \in r$ satisfies $\Sigma^+ | R_i$ guarantees that the state r is globally consistent with respect to Σ. It has been shown that $\Sigma^+ | R_i$ is a set of fd's embedded in R_i, for every $R_i \in \mathbf{R}$ [GY]. The following theorem shows that the nondetermining and key-determined condition characterizes replacement-anomaly-frredom when independence is assumed.

THEOREM 12. *Let* $\mathbf{R}$ *be independent with respect to* $\Sigma = F \cup \{\|\times\|\mathbf{R}\}$. $\mathbf{R}$ *is replacement-anomaly-free with respect to* Σ *if and only if* $\mathbf{R}$ *satisfies the nondetermining and key-determined condition with respect to* Σ.

Proof. "Only if." The "only if" part follows directly from Theorem 8.

"If." Let r be a consistent state with respect to Σ. Let $\Sigma^+ | R_i = F_i$ be the set of fd's embedded in R_i. Without loss of generality, we assume F_i is left-reduced. Suppose some tuple in $r_i \in r$ is updated on A and the updated relation violates some left-reduced fd $Y \to B \in F_i$. Clearly $A \in YB$. Since $\mathbf{R}$ is nondetermining and key-determined with respect to Σ, Y is a candidate key of R_i and $A = B$. Since the Y-column of r_i is not changed by the update, no two tuples in the updated relation agree on Y. We can conclude that the updated relation cannot violate the fd $Y \to B$. Hence after each update on some r_i, the resulting relation satisfies F_i. Since $\mathbf{R}$ is independent with respect to Σ, updates preserve the consistency of data in a database. Therefore $\mathbf{R}$ is replacement-anomaly-free with respect to Σ. □

Recently, the class of γ-acyclic cover embedding BCNF database schemes has been shown to be highly desirable with respect to query processing and constraint enforcement [CH]. With this class of schemes, Algorithm Enforce, shown in Fig. 1, is an incremental algorithm used to enforce satisfaction of fd's efficiently.

Let $\mathbf{R}$ be a γ-acyclic cover embedding BCNF database scheme with respect to F and let r be a consistent state of $\mathbf{R}$. Let r_p be a relation that is being changed by an insertion of a tuple t, where $r_p \in r$. Let $\{K_{p_1}, \cdots, K_{p_m}\}$ be the set of nontrivial candidate keys of R_p. A candidate key K of R is *nontrivial* if there is a nontrivial fd $K \to A$ embedded in R. It was proved in [CH] that Algorithm Enforce correctly determines if an updated state is consistent with respect to F.

Given this class of database schemes, the nondetermining and key-determined condition again characterizes the class of replacement-anomaly-free schemes with respect to $\Sigma = F \cup \{\|\times\|\mathbf{R}\}$.

Input: A consistent state r of a γ-acyclic cover embedding BCNF database scheme $\mathbf{R}$ with respect to F.
A tuple t to be inserted in $r_p \in r$, $R_p \in \mathbf{R}$.
Output: *No*, if $r \cup \{t\}$ is not consistent with respect to F; *yes* otherwise.
Notation: $\{K_{p_1}, \cdots, K_{p_m}\}$ is the set of nontrivial candidate keys of R_p.

```
(1) for each K_{p_i} do begin
(2)     for each A ∈ R_p − K_{p_i} do begin
(3)         for all R_q ∈ R such that R_q ⊇ K_{p_i}A do begin
(4)             if π_{K_{p_i}A}(r_q) ∪ π_{K_{p_i}A}({t}) does not satisfy K_{p_i} → A then do begin
(5)                 print no; halt end
(6)         end
(7)     end
(8) end
(9) print yes
```

FIG. 1. *Algorithm Enforce.*

THEOREM 13. *Let* $\mathbf{R}$ *be cover embedding* BCNF *with respect to* F *and is* γ*-acyclic.* $\mathbf{R}$ *is replacement-anomaly-free with respect to* F *if and only if* $\mathbf{R}$ *satisfies the nondetermining and key-determined condition with respect to* F.

Proof. "Only if." Follows from Theorem 8.

"If." Suppose we update a tuple of r_p in a consistent state on an attribute A. We can consider the update as a deletion followed by an insertion. Since under our assumption deletion does not map a consistent state to an inconsistent one, let us consider the insertion. If after the insertion some nontrivial key dependency $K_{p_i} \to B \in F^+ \mid R_p$ is violated, then by the nondetermining and key-determined condition, $A = B$. Hence the update did not change any value of K_{p_i} in r_p. By statement (4) in Algorithm Enforce (see Fig. 1), the updated relation r_p cannot violate $K_{p_i} \to B$. Hence after the insertion, the updated relation r_p satisfies the set of nontrivial key dependencies. Again by the nondetermining and key-determined condition, $K_{p_i} \to A$ is embedded in no other relation scheme. This implies the updated state is consistent with respect to F. Hence $\mathbf{R}$ is replacement-anomaly-free with respect to F. □

COROLLARY 2. *Let* $\mathbf{R}$ *be cover embedding* BCNF *with respect to* F *and be* γ*-acyclic. Let* $\Sigma = F \cup \{\|\times\|\mathbf{R}\}$. $\mathbf{R}$ *is replacement-anomaly-free with respect to* Σ *if and only if* $\mathbf{R}$ *satisfies the nondetermining and key-determined condition with respect to* F.

Proof. The proof follows directly from Theorems 11 and 13. □

4.2.3. A natural case. In § 4.2.1 and 4.2.2, no assumption is made on the updatable attributes. In many cases, a candidate key of a relation scheme is designated as the primary key of the relation scheme. The primary key of a relation scheme is used to represent an object or entity in the real world. Hence the primary key values cannot be modified by an application program [Cod4], [TL]. In this case, we assume every attribute in $R_i - P_i$ is updatable, where P_i is the primary key of R_i, for all $R_i \in \mathbf{R}$. In this section, we want to characterize when a cover embedding database scheme $\mathbf{R}$ is replacement-anomaly-free with respect to F. By Theorem 11, the characterization is also applicable when $F \cup \{\|\times\|\mathbf{R}\}$ is considered. We need the following results before we prove the characterization.

THEOREM 14. *Let* Σ *be a set of dependencies. Assume further that* $R_i - P_i$ *is the set of updatable attributes in* R_i, *where* P_i *is a candidate key of* R_i, *for all* $R_i \in \mathbf{R}$. *If* $\mathbf{R}$ *is replacement-anomaly-free with respect to* Σ, *then* $\mathbf{R}$ *is single-key with respect to* Σ.

Proof. Suppose there is $R_i \in \mathbf{R}$ such that R_i embeds two candidate keys. Let the candidate keys be X and Y. Without loss of generality, let X be P_i. Since X and Y

are two distinct candidate keys, there is an $A \in X - Y$. Since Y is a candidate key, $Y \to A$ is a nontrivial fd embedded in R_i. This implies there is a subset Z of Y such that $Z \to A$ is left-reduced. Since X cannot embed a nontrivial fd and $A \in X$, there is an attribute $B \in Z - X$. By assumption, B is updatable. Since $Z \to A$ is left-reduced, B is updatable and $A \neq B$, **R** violates the nondetermining and key-determined condition. By Theorem 8, **R** is not replacement-anomaly-free with respect to Σ. □

THEOREM 15. *Let Σ be a set of dependencies. Assume further that $R_i - P_i$ is the set of updatable attributes in R_i, where P_i is a candidate key of R_i, for all $R_i \in \mathbf{R}$. If* **R** *is replacement-anomaly-free with respect to Σ, then* **R** *is* BCNF *with respect to Σ.*

Proof. Suppose there is $R_i \in \mathbf{R}$, which is not in BCNF with respect to $\Sigma^+ | R_i$. That is, there is some nontrivial fd $X \to A \in \Sigma^+$ embedded in R_i but X is not a superkey, and hence is not a candidate key of R_i. Without loss of generality, we assume $X \to A$ is left-reduced. By assumption on the updatable attributes and the fact that the candidate key P_i cannot contain a nontrivial fd, some attribute $B \in XA$ is updatable. This implies **R** does not satisfy the nondetermining and key-determined condition. By Theorem 8, **R** is not replacement-anomaly-free with respect to Σ. □

For the class of cover embedding BCNF database schemes with respect to F, Sagiv has shown that independence is characterized by a condition known as the *uniqueness* condition [S1], [S2]. A database scheme **R** is said to satisfy the *uniqueness* condition if there is no R_i and R_j in **R**, $i \neq j$, such that $(R_i)^+_{F-F_j}$ embeds a nontrivial fd $K_j \to A \in F_j$, where F_j is the set of projected fd's on R_j. The following shows that under certain assumptions, independence is necessary for replacement-anomaly-freedom.

THEOREM 16. *Let* **R** *be cover embedding with respect to F. Assume further that $R_i - P_i$ is the set of updatable attributes in R_i, where P_i is a candidate key of R_i, for all $R_i \in \mathbf{R}$. If* **R** *is single-key* BCNF *and replacement-anomaly-free with respect to F, then* **R** *is independent with respect to F.*

Proof. Suppose **R** is not independent with respect to F. Since **R** is cover embedding BCNF and nonindependent with respect to F, **R** violates the uniqueness condition. That is, there are R_i and R_j, $i \neq j$, such that $(R_i)^+_{F-F_j}$ embeds a nontrivial fd $K_j \to A \in F_j = F^+ | R_j$, where K_j is a superkey of R_j. Since **R** is single-key with respect to Σ, A is updatable in R_j. Let $X_1 \to A_1, \cdots, X_p \to A_p$ be a sequence of fd's in F used in computing $(R_i)^+_{F-F_j}$. By the cover embedding property, we can assume each $X_q \to A_q$ embedded in some S_q, where $S_q \neq R_j$, for all $1 \leqq q \leqq p$. By Theorem 9, **R** is not replacement-anomaly-free with respect to F. □

THEOREM 17. *Let* **R** *be cover embedding with respect to F. Assume further that $R_i - P_i$ is the set of updatable attributes in R_i, where P_i is a candidate key of R_i, for all $R_i \in \mathbf{R}$.* **R** *is replacement-anomaly-free with respect to F if and only if* **R** *is single-key,* BCNF *and independent with respect to F.*

Proof. "If." Let t_i be a tuple of R_i in some consistent state that is being modified on some updatable attribute A. Since A is updatable, $A \in R_i - P_i$. Since **R** is single-key BCNF and independent with respect to F, the modified state is consistent with respect to F. Hence **R** is replacement-anomaly-free with respect to F.

"Only if." By Theorems 14 and 15, **R** is single-key and BCNF with respect to F. By Theorem 16, **R** is independent with respect to F. □

THEOREM 18. *Let* **R** *be cover embedding with respect to F and $\Sigma = F \cup \{\| \times \| \mathbf{R}\}$. Assume further that $R_i - P_i$ is the set of updatable attributes in R_i, for all $R_i \in \mathbf{R}$.* **R** *is replacement-anomaly-free with respect to Σ if and only if* **R** *is single-key,* BCNF *and independent with respect to F.*

Proof. First observe that the sets of fd's implied by F and Σ are identical [GY]. Then the theorem follows directly from Theorems 11 and 17. □

5. Conclusion. We have argued that normalization does not necessarily solve the problem of update anomalies. In view of this, we have studied this problem and proposed a theory for designing database schemes that are free of update anomalies.

Unlike Codd, LeDoux, and Parker [Cod2], [Cod3], [LP], we have viewed insertion and deletion anomalies as a problem that is different from replacement anomalies. Unlike Bernstein and Goodman [BG], we have regarded insertion and deletion anomalies as a problem of recording data in a relation, and replacement anomalies as a problem of independent modification of attribute values of a tuple in a consistent state. Because of the nature of these problems, we have studied insertion and deletion anomalies in the context of a single relation scheme. We have proposed two simple models for analyzing and designing desirable database schemes that are free of insertion and deletion anomalies. With the assumption that every embedded nontrivial fd represents a basic relationship in a database, we have shown that BCNF is a necessary condition for a relation scheme to be free of insertion and deletion anomalies. For replacement anomalies, it is closely related to the constraint enforcement problem and therefore it was analyzed in the context of a database scheme. Assuming any attribute is updatable, we have given some necessary conditions for a database scheme to be free of replacement anomalies. We also have characterized when an independent scheme or a γ-acyclic BCNF scheme is replacement-anomaly-free. In many cases, a candidate key of a relation scheme is assumed to be nonupdatable. Under this assumption, we proved that single-key, BCNF and independence characterize replacement-anomaly-freedom when the constraints considered are a set of embedded fd's. This characterization is also applicable when an embedded cover is augmented with the full jd $\|\times\|\mathbf{R}$.

Acknowledgments. The author is grateful to Héctor Hernández for his helpful comments on an initial draft of this paper. The author also thanks an anonymous referee, D. Wood, and J. D. Ullman for their constructive comments that greatly enhanced the readability of this paper.

REFERENCES

[ABU] A. V. AHO, C. BEERI, AND J. D. ULLMAN, *The theory of joins in relational data-bases*, ACM Trans. Database Systems, 4 (1979), pp. 297-314.

[AC] P. ATZENI AND E. P. F. CHAN, *Efficient query answering in the representative instance approach*, Proc. 4th ACM Symposium on Principles of Database Systems, 1985, pp. 181-188.

[ASU] A. V. AHO, Y. SAGIV, AND J. D. ULLMAN, *Equivalence of relational expressions*, SIAM J. Comput., 8 (1979), pp. 218-246.

[BB] C. BEERI AND P. A. BERNSTEIN, *Computational problems related to the design of normal form relational database schemas*, ACM Trans. Database Systems, 4 (1979), pp. 30-59.

[BMSU] C. BEERI, A. O. MENDELZON, Y. SAGIV, AND J. D. ULLMAN, *Equivalence of relational database schemas*, SIAM J. Comput., 10 (1981), pp. 352-370.

[BV] C. BEERI AND M. Y. VARDI, *A proof procedure for data dependencies*, J. Assoc. Comput. Mach., 31 (1984), pp. 718-741.

[B] P. A. BERNSTEIN, *Synthesizing third normal form relations from functional dependencies*, ACM Trans. Database Systems, 1 (1976), pp. 277-298.

[BG] P. A. BERNSTEIN AND N. GOODMAN, *What does Boyce-Codd normal form do?* Proc. 6th International Conference on Very Large Data Bases, 1980, pp. 245-259.

[BDB] J. BISKUP, U. DAYAL, AND P. A. BERNSTEIN, *Synthesizing independent database schemas*, Proc. ACM SIGMOD Conference, 1979, pp. 143-152.

[CH] E. P. F. CHAN AND H. J. HERNANDEZ, *On the Desirability of γ-Acyclic* BCNF *Database Schemas*, Lecture Notes in Computer Science 243, Springer-Verlag, Berlin, New York, 1986, pp. 105-122.

[CM] E. P. F. CHAN AND A. O. MENDELZON, *Independent and separable database schemes*, SIAM J. Comput., 16 (1987), pp. 841-851.

[Cod1] E. F. CODD, *A relational model for large shared data banks*, Comm. ACM, 13 (1970), pp. 377-387.

[Cod2] ———, *Further normalization of the data base relational model*, in Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 33-64.

[Cod3] ———, *Recent investigations in relational database systems*, International Federation of Information Processing Conference, 1974, pp. 1017-1021.

[Cod4] ———, *Extending the database relational model to capture more meaning*, ACM Trans. Database Systems, 4 (1979), pp. 397-434.

[Da] C. J. DATE, *An Introduction to Database Systems*, Addison-Wesley, Reading, MA, 1986.

[DGS1] B. C. DESAI, P. GOYAL, AND F. SADRI, *Updates in relational databases*, Proc. AFIPS National Computer Conference, 1986, pp. 237-248.

[DGS2] ———, *Fact structure and its application to update in relational databases*, Inform. Systems, 12 (1987), pp. 215-221.

[F1] R. FAGIN, *Multivalued dependencies and a new normal form for relational databases*, ACM Trans. Database Systems, 2 (1977), pp. 262-278.

[F2] ———, *Normal forms and relational database operators*, Proc. ACM SIGMOD Conference, 1979, pp. 153-160.

[FMU] R. FAGIN, A. O. MENDELZON, AND J. D. ULLMAN, *A simplified universal relation assumption*, ACM Trans. Database Systems, 7 (1984), pp. 343-360.

[GJ] 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.

[GMV] M. H. GRAHAM, A. O. MENDELZON, AND M. Y. VARDI, *Notions of dependency satisfaction*, J. Assoc. Comput. Mach., 33 (1986), pp. 105-129.

[GY] M. H. GRAHAM AND M. YANNAKAKIS, *Independent database schemas*, J. Comput. System Sci., 28 (1984), pp. 121-141.

[H] P. HONEYMAN, *Testing satisfaction of functional dependencies*, J. Assoc. Comput. Mach., 29 (1982), pp. 668-677.

[IIK] M. ITO, M. IWASAKI, AND T. KASAMI, *Some results on the representative instance in relational databases*, SIAM J. Comput., 14 (1985), pp. 334-354.

[K] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, Plenum Press, New York, 1972.

[LP] C. H. LEDOUX AND D. S. PARKER, *Reflections on Boyce-Codd normal form*, Proc. 8th International Conference on Very Large Data Bases, 1982, pp. 131-141.

[Ma] D. MAIER, *The Theory of Relational Databases*, Computer Science Press, Potomac, 1983.

[MMS] D. MAIER, A. O. MENDELZON, AND Y. SAGIV, *Testing implications of data dependencies*, ACM Trans. Database Systems, 4 (1979), pp. 455-469.

[MU] D. MAIER AND J. D. ULLMAN, *Maximal objects and the semantics of universal relation database*, ACM Trans. Database Systems, 8 (1983), pp. 1-14.

[MW] D. MAIER AND D. WARREN, *Specifying connections for a universal relation*, Proc. ACM SIGMOD Conference, 1982, pp. 1-7.

[M] A. O. MENDELZON, *Database states and their tableaux*, ACM Trans. Database Systems, 9 (1984), pp. 264-282.

[S1] Y. SAGIV, *Can we use the universal instance assumption without using nulls?* Proc. ACM SIGMOD Conference, 1981, pp. 108-120.

[S2] ———, *A characterization of globally consistent databases and their correct access paths*, ACM Trans. Database Systems, 8 (1983), pp. 266-286.

[S3] ———, *Evaluation of queries in independent database schemes*, unpublished manuscript, 1984.

[Sc] E. SCIORE, *The universal interface and database design*, Ph.D. dissertation, Princeton University, Princeton, NJ, October 1980.

[TL] D. C. TSICHRITZIS AND F. H. LOCHOVSKY, *Data Models*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[U] J. D. ULLMAN, *Principles of Database Systems*, Computer Science Press, Potomac, 1982.

[V] Y. VASSILIOU, *A formal treatment of imperfect information in data management*, Computer Systems Research Group/University of Toronto, Toronto, Ontario, Canada, TR-123, November 1980.

[Y] M. YANNAKAKIS, *Algorithms for acyclic database schemes*, Proc. 7th International Conference on Very Large Data Bases, 1981, pp. 82-94.

[Z] C. ZANIOLO, *A new normal form for the design of relational database schemata*, ACM Trans. Database Systems, 7 (1982), pp. 489-499.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 449-462, June 1989

ON TALLY RELATIVIZATIONS OF *BP*-COMPLEXITY CLASSES*

SHOUWEN TANG† AND OSAMU WATANABE‡

Abstract. It is known that $AM = BP \cdot NP$. Babai [Proc. 17th Annual ACM Symposium Theory of Computing, 1985, pp. 421-429] and Goldwasser and Sipser [Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 59-68] asked whether $BP \cdot NP$ is equal to $\{A \mid$ for almost every set B, $A \in NP(B)\}$. This question is still open. In this paper it is shown that (1) for every $k \geqq 0$ and every set A, $A \in BP \cdot \Sigma_k^P$ if and only if for almost every tally set T, $A \in \Sigma_k^P(T)$, and (2) for every $k \geqq 0$ and almost every tally set T, $BP \cdot \Sigma_k^P(T) = \Sigma_k^P(T)$. From them are obtained some properties of the "*BP*-polynomial-time hierarchy" studied by Schöning [Proc. 2nd Annual Conference on Structure in Complexity Theory, 1987, pp. 2-8]. That is, the *BP*-polynomial-time hierarchy has the properties that are precisely parallel to those of the polynomial-time hierarchy. The proofs of these results provide examples of the use of properties of complexity classes specified by relativizations to obtain properties of unrelativized complexity classes.

Key words. probabilistic complexity classes, tally relativizations, random oracle sets

AMS(MOS) subject classifications. 68Q15, 68Q30, 03D15

1. Introduction. Baker, Gill, and Solovay [BGS75] have proved that there exists an oracle set A separating P and $NP: P(A) \neq NP(A)$. It is natural to ask about the number of oracle sets with this separating property. One way to measure the size of this class of sets has been used by Bennett and Gill [BG81]. The characteristic function of a language is an infinite string. Every language can be identified with a real number in the unit interval $[0, 1]$, and a class of languages is identified with a subset of the unit interval. The Lebesgue measure ν of that subset provides a natural way to measure the size of the corresponding class of languages. This measure corresponds to the Bernoulli independent testing sequence: a random language (infinite string) is the sequence that results from tossing an unbiased coin infinitely often. Bennett and Gill have shown that $\nu(\{B \mid P(B) \neq NP(B)\}) = 1$ and $\nu(\{C \mid P(C) = BPP(C)\}) = 1$; this can be described as that for a random oracle set B (or for "almost every" oracle set B), $P(B) \neq NP(B)$, and for a random oracle set C (or for "almost every" oracle set C), $P(C) = BPP(C)$. Following Yao's proof [Ya85] of the existence of an oracle set D such that $PH(D) \neq \text{PSPACE}(D)$, Cai [Ca86] and Babai [Ba87] have shown that for a random oracle set D, $PH(D) \neq \text{PSPACE}(D)$.

When studying probabilistic complexity classes and lowness properties, Schöning [Sc87] introduced the notion of the "*BP*-operator" and developed some properties of the "*BP*-polynomial-time hierarchy," that is, the variation of the polynomial-time hierarchy obtained by applying the "*BP*" operator to classes in that hierarchy. Babai [Ba85] introduced the class AM and, in addition, claimed that $BPP = \{A \mid$ for almost every set B, $A \in P(B)\}$. Ambos-Spies [Am86] showed that (i) $A \in P$ if and only if $A \leqq_m^P B$ for almost every B; (ii) $A \in BPP$ if and only if $A \leqq_T^P B$ for almost every B; and (iii) for almost every pair (B_1, B_2), $BPP = P(B_1) \cap P(B_2)$. (Kurtz [Ku87] also proved (ii) and (iii).) Babai [Ba85] and Goldwasser and Sipser [GS86] proposed the

* Received by the editors February 3, 1988; accepted for publication August 16, 1988. This research was performed while the authors were visiting the Department of Mathematics, University of California, Santa Barbara, California, and was supported in part by the National Science Foundation under grants CCR-8611980. A preliminary version of this paper was presented at the 3rd Annual Structure in Complexity Theory Conference [Proc. 3rd Annual Conference on Structure in Complexity Theory, 1988].

† Department of Computer Science, Beijing Computer Institute, Beijing, People's Republic of China.

‡ Department of Computer Science, Tokyo Institute of Technology, Tokyo 152, Japan.

following open problem: While $AM = BP \cdot NP$, is it the case that $AM = \{A \mid \text{for almost every set } B,\ A \in NP(B)\}$?

Consider the following two questions.

QUESTION 1. Can $BP \cdot \Sigma_k^P$ be characterized by means of oracles? That is, is it true that $A \in BP \cdot \Sigma_k^P$ if and only if for almost every oracle B, $A \in \Sigma_k^P(B)$?

QUESTION 2. What is the relation between relativizations of $BP \cdot \Sigma_k^P$ and Σ_k^P? Specifically, is it true that for almost every oracle B, $BP \cdot \Sigma_k^P(B) = \Sigma_k^P(B)$?

For Question 1, case $k = 0$ was solved by Ambos-Spies [Am86] and case $k \geqq 1$ is still open.[1] For Question 2, case $k = 0$ was solved by Bennett and Gill [BG81] and case $k \geqq 1$ was solved by Tang [Ta87].

In this paper, we introduce the concept of random tally sets and both Question 1 and Question 2 are solved for the tally oracle case (see Theorems 5.2 and 5.4). That is, we establish the following results.

(1) $A \in BP \cdot \Sigma_k^P$ if and only if for almost every tally oracle T, $A \in \Sigma_k^P(T)$;

(2) For almost every tally oracle T, $BP \cdot \Sigma_k^P(T) = \Sigma_k^P(T)$.

Note that in many cases the results for random *tally* oracle sets are independent from those for random oracle sets. For example, the above statement (2) with $k = 0$ is not a direct consequence of the result by Bennett and Gill [BG81] that $BPP(B) = P(B)$ for almost every oracle B. Also the statement (1) with $k = 1$ is not a complete answer to the original question of Babai and of Goldwasser and Sipser.

Compared with general relativizations, relativization with respect to a tally oracle set is "much closer" to the corresponding unrelativized case. For example, $P \neq NP$ if and only if there is a tally set T such that $P(T) \neq NP(T)$ [LS86]: thus, the existence of even one tally oracle that separates NP from P is not known, whereas $NP(B) \neq P(B)$ for almost every oracle B [BG81]. Similarly PH collapses if and only if there is a tally set T such that $PH(T)$ collapses; and $PH = \text{PSPACE}$ if and only if there is a tally set T such that $PH(T) = \text{PSPACE}(T)$ [BBS86], [LS86]. This suggests that the tally oracle is of special significance for the unrelativized case and the role of random tally sets as oracle sets should be investigated. Indeed, using the above two statements, we show that the BP-polynomial-time hierarchy has properties faithfully reflecting those of the polynomial-time hierarchy (see Theorems 5.6–5.9). That is, the properties established by relativization yield results for unrelativized cases.

The characterization of $BP \cdot \Sigma_k^P$ by statement (1) explicitly tells the difference between $\Sigma_k^P/poly$ and $BP \cdot \Sigma_k^P$. Take $k = 0$ as an example. A set L has polynomial-size circuits (belongs to $P/poly$: see, e.g., [BH77]) if and only if there is one tally set T such that L is in $P(T)$. On the other hand, a set L is in BPP if and only if for almost every tally set T, L is in $P(T)$. (Kämper [Kä87] also has observed this in his framework.)

This paper is organized as follows: In § 2 preliminaries are given. In § 3 the concept of random tally sets is introduced so that we can use the phrase "for almost every tally set." In § 4 we solve Questions 1 and 2 in a general setting. As a consequence of § 4, we get the desired results on BP-polynomial-time hierarchy in § 5.

2. Preliminaries. Let $\Sigma = \{0, 1\}$ so that Σ^* denotes the set of all finite strings over $\{0, 1\}$, with the empty word being denoted by ε. Let Σ^ω denote the set of all infinite sequences over $\{0, 1\}$. All languages will be assumed to be taken over Σ. The length of a string w is denoted by $|w|$. The cardinality of a set S is denoted by $\|S\|$. A *tally* set is a set of strings over $\{0\}^*$. We denote the class of all tally sets by $TALLY$. In this paper, let T with or without subscript always denote a tally set, and let $\mathcal{T}$ denote a

[1] Recently Nisan and Wigderson [NW88] solved this problem.

class of tally sets. If A is a language over Σ, then $\bar{A}$ denotes $\Sigma^* - A$, but if T is a tally set, then $\bar{T}$ denotes $\{0\}^* - T$. For a class $\mathscr{A}$ of languages over Σ and a class $\mathscr{T}$ of tally languages, we denote $\{\bar{A} \mid A \in \mathscr{A}\}$ by $\mathrm{co}-\mathscr{A}$ and $\{\bar{T} \mid T \in \mathscr{T}\}$ by $\mathrm{co}-\mathscr{T}$. In addition, $\mathscr{A}^c$ denotes $\{A \mid A \text{ is not in } \mathscr{A}\}$ and $\mathscr{T}^c$ denotes $\{T \in TALLY \mid T \text{ is not in } \mathscr{T}\}$.

Some fixed pairing function computable in polynomial time is assumed and is denoted by $\langle\ ,\ \rangle$. Furthermore, it is assumed that $\langle\ ,\ \rangle$ restricted to $\{0\}^* \times \{0\}^*$ takes values in $\{0\}^*$.

For an oracle machine M and a set A, M^A is the language accepted relative to A by M. For any set A, the class of functions that can be computed relative to A by polynomial time-bounded deterministic oracle transducers is denoted by $PF(A)$, and PF denotes the class $PF(\varnothing)$. Frequently, we will say "$f \in PF(-)$" to mean that f denotes a polynomial time-bounded oracle transducer. If $f \in PF(-)$, then for any set A, $f^A \in PF(A)$ denotes the function computed relative to A by the polynomial time-bounded transducer f. We will say "$f \in PF$" to mean that f denotes a polynomial time-bounded transducer or the function computed by such a transducer.

Classes in the polynomial-time hierarchy are denoted in the usual way: $\Sigma_0^P = \Delta_1^P = \Pi_0^P = P$, $\Sigma_{k+1}^P = NP(\Sigma_k^P)$, $\Delta_{k+1}^P = P(\Sigma_k^P)$, $\Pi_k^P = \mathrm{co}-\Sigma_k^P$, and $PH = \bigcup_{n \geqq 0} \Sigma_n^P$ [St77], [Wr77].

If $\mathscr{C}$ is a class of languages on Σ and $\leqq_R$ is a reducibility, we say that $\mathscr{C}$ is *closed under* $\leqq_R$ if $A \leqq_R B$ and $B \in \mathscr{C}$ imply $A \in \mathscr{C}$. There are two specific reducibilities that are of interest here.

(a) For sets A and B, define $A \leqq_{\mathrm{pos}}^P B$ (also written $A \in P_{\mathrm{pos}}(B)$) if $A \in P(B)$ is witnessed by a polynomial time-bounded oracle machine M with the property that $X \subseteq Y$ implies $M^X \subseteq M^Y$.

(b) For sets A and B, define $A \leqq_{\mathrm{maj}}^P B$ if there exists a function $f \in PF$ such that if $f(x) = y_1 \# y_2 \# \cdots \# y_t$, then $x \in A$ if and only if the majority of y_i's are in B.

It is easy to see that for any set A, all of $\Sigma_k^P(A)$, $\Pi_k^P(A)$, $\Delta_k^P(A)$, $PH(A)$, and PSPACE (A) are closed under $\leqq_{\mathrm{pos}}^P$. Note that $A \leqq_{\mathrm{maj}}^P B$ implies $A \leqq_{\mathrm{pos}}^P B$. If a class $\mathscr{C}$ is closed under $\leqq_{\mathrm{pos}}^P$, then $\mathscr{C}$ is also closed under $\leqq_{\mathrm{maj}}^P$. Thus, we see that for any set A, all of $\Sigma_k^P(A)$, $\Pi_k^P(A)$, $\Delta_k^P(A)$, $PH(A)$, and PSPACE (A) are closed under $\leqq_{\mathrm{maj}}^P$. Also, if $\mathscr{C}$ is closed under $\leqq_{\mathrm{maj}}^P$, then $\mathscr{C}$ is closed under $\leqq_m^P$.

In this paper, we first develop our results in general framework; then using this machinery, the classes in the polynomial-time hierarchy are observed. In order to discuss relativized complexity classes in general setting, we introduce the following notion and notation.

For language A, if $x \in A$, then $A(x) = 1$, else $A(x) = 0$. For language A and total function f, $A \circ f$ is a language B such that for all x, $B(x) = A \circ f(x)$: i.e., $B = A \circ f$ means $B \leqq_m A$ via f, or equivalently, $B = f^{-1}(A)$. So, for sets D and A, and oracle transducer f, $D \circ f^A$ is the language $\{x \mid D \circ f^A(x) = 1\}$.

DEFINITION 2.1. For language class $\mathscr{C}$ and class $\mathscr{F}$ of total functions, $\mathscr{C} \circ \mathscr{F}$ is the language class $\{A \circ f \mid A \in \mathscr{C}, f \in \mathscr{F}\}$.

It is clear from the definitions that the following hold:

(a) $\mathscr{C}$ is closed under $\leqq_m^P$ if and only if $\mathscr{C} \circ PF = \mathscr{C}$.

(b) $(\mathrm{co}-\mathscr{C}) \circ \mathscr{F} = \mathrm{co}-(\mathscr{C} \circ \mathscr{F})$.

(c) $(\mathscr{C} \circ \mathscr{F}_1) \circ \mathscr{F}_2 = \mathscr{C} \circ (\mathscr{F}_1 \circ \mathscr{F}_2)$.

For any set A and language class $\mathscr{C}$, we regard $\mathscr{C} \circ PF(A)$ as the relativized class of $\mathscr{C}$ with respect to A. Let us take $NP \circ PF(A)$ as an example. The decision problem in $NP \circ PF(A)$ can be done as follows: For a word x, first run a polynomial-time transducer f on x relative to A to get the output $y = f^A(x)$; then run an NP-machine M on y (without oracle); $x \in L$ if and only if M accepts y. Hence, $NP \circ PF(A) \subseteq NP(A)$.

However, the question of whether $NP \circ PF(A) = NP(A)$ is still open. In general we have that $\Sigma_k^P \circ PF(A) \subseteq \Sigma_k^P(A)$, but the equality problem is open. Nevertheless, $\Sigma_k^P \circ PF(T) = \Sigma_k^P(T)$ for every tally set T (see Lemma 5.1); hence, when considering tally oracles, the above notion is a reasonable abstraction for relativized complexity classes.

For any class $\mathscr{C}$ of languages, define $\mathscr{C}/poly$ to be the class of languages A such that for some $C \in \mathscr{C}$, some polynomial length-bounded function $h: \{0\}^* \to \Sigma^*$, and all x, $x \in A$ if and only if $\langle x, h(0^{|x|})\rangle \in C$.

The following relation is a generalization of a characterization of $P/poly$, and the proof is essentially the same as the one for $P/poly$ (see, e.g., [BH77]).

LEMMA 2.2. *If $\mathscr{C}$ is closed under $\leqq_m^P$, then $\mathscr{C}/poly = \mathscr{C} \circ PF(TALLY)$.*

Proof. Suppose that $A \in C/poly$. Then there exist $C \in \mathscr{C}$ and a polynomial length-bounded function h such that $A(x) = C(\langle x, h(0^{|x|})\rangle)$ for all x.

Let $T_h = \{\langle 0^n, 0^i, 0^b\rangle | b = 0 \text{ or } 1, \text{ the } i\text{th bit of } h(0^n) \text{ is } b\}$, and let $f^{T_h}(x) = \langle x, h(0^{|x|})\rangle$. Then $A = C \circ f^{T_h} \in \mathscr{C} \circ PF(TALLY)$.

Suppose that $A \in \mathscr{C} \circ PF(TALLY)$. Then there exists $C \in \mathscr{C}$, tally set T, and $f \in PF(-)$ such that $A \in C \circ f^T$. Let $p(n)$ be a polynomial that bounds the running time of f. Let $h(0^n) = T(\varepsilon)T(0)T(0^2) \cdots T(0^{p(n)})$. For any $z \in \Sigma^*$, let T_z denote the tally language with characteristic sequence $z0^\omega$. Let $g(\langle x, y\rangle) = f^{T_y}(x)$. Then $g \in PF$, h is polynomial length-bounded, and $f^T(x) = g(\langle x, h(0^{|x|})\rangle)$. Thus, $A(x) = C \circ f^T(x) = C \circ g(\langle x, h(0^{|x|})\rangle)$. Let $D = C \circ g$. Then $D \in \mathscr{C} \circ PF = \mathscr{C}$. Hence, $A(x) = D(\langle x, h(0^{|x|})\rangle)$ for all x, i.e., $A \in \mathscr{C}/poly$. □

For predicate P and natural number m, $\Pr_m[y: P(y)]$ is the conditional probability $\Pr[P/\Sigma^M] = 2^{-m} \times \|\{y \mid P(y) \text{ and } |y| = m\}\|$.

In this paper we focus on the operator "BP" defined by Schöning [Sc87] as follows: For language class $\mathscr{C}$, $BP \cdot \mathscr{C}$ is the class of languages A such that for some $C \in \mathscr{C}$, and polynomial $p(n)$, and all $x \in \Sigma^*$,

$$\Pr_{p(|x|)}[y: A(x) = C(\langle x, y\rangle)] > \tfrac{3}{4}.$$

It is clear that $BP \cdot P = BPP$ and $BP \cdot NP = AM$ (see [Sc87], [Ba85] for discussion about AM). However, we had better clarify the relativized case. Recall that for any sets A and L, $L \in BPP(A)$ if and only if there exists a nondeterministic oracle machine M that runs in polynomial time, has fan-out two, and has the following properties:

(i) $x \in L \Rightarrow$ more than $\frac{3}{4}$ of M's computations on x relative to A are accepting.

(ii) $x \notin L \Rightarrow$ less than $\frac{1}{4}$ of M's computations on x relative to A are accepting.

On the other hand, for a set A, $BP \cdot P(A)$ is the class of sets L such that there exist $D \in P(A)$ and polynomial $p(n)$ such that for all x

$$Pr_{p(|x|)}[y: L(x) = D(\langle x, y\rangle)] > \tfrac{3}{4}.$$

It is easy to prove that for every set A, $BPP(A) = BP \cdot P(A)$.

The following facts are immediate from the definitions:

(a) $BP \cdot \text{co}-\mathscr{C} = \text{co}-(BP \cdot \mathscr{C})$.

(b) $\mathscr{C} \subseteq \mathscr{D}$ implies $BP \cdot \mathscr{C} \subseteq BP \cdot \mathscr{D}$.

(c) If $\mathscr{C}$ is closed under padding (i.e., $A \in \mathscr{C}$ implies $\langle A, \Sigma^*\rangle \in \mathscr{C}$), then $\mathscr{C} \subseteq BP \cdot \mathscr{C}$.

Thus, we see that if $\mathscr{C}$ is closed under $\leqq_m^P$, i.e., $\mathscr{C} \circ PF = \mathscr{C}$, then $\mathscr{C} \subseteq BP \cdot \mathscr{C}$. From the above remarks, we see that if $\mathscr{C}$ is closed under $\leqq_{\text{pos}}^P$, then $\mathscr{C} \subseteq BP \cdot \mathscr{C}$.

Now we have the Amplification Lemma. While Schöning proved this under the hypothesis that $\mathscr{C}$ is closed under $\leqq_{\text{pos}}^P$, we state the result in terms of $\mathscr{C}$ being closed under $\leqq_{\text{maj}}^P$. The proof is essentially the same as that of Schöning and so is omitted.

LEMMA 2.3 (Amplification). *If $\mathscr{C}$ is closed under $\leqq^P_{\text{maj}}$, then for all $A \in BP \cdot \mathscr{C}$ and all polynomials $q(n)$, there is a set $B \in \mathscr{C}$ and a polynomial $p(n)$ such that for all $n \geqq 0$*

$$\Pr_{p(n)}[y: \forall x_{|x| \leqq n}(A(x) = B(\langle x, y\rangle))] > 1 - 2^{-q(n)}.$$

PROPOSITION 2.4. *If $\mathscr{C}$ is closed under $\leqq^P_{\text{maj}}$, then $BP \cdot \mathscr{C} \subseteq \mathscr{C}/poly$; thus $BP \cdot \mathscr{C} \subseteq \mathscr{C} \circ PF(TALLY)$.*

Proof. The proof is immediate from Lemmas 2.2 and 2.3. □

The following facts will be used or re-proved in later sections.

PROPOSITION 2.5 [Sc87].

(a) $\forall k \geqq 1$, $BP \cdot \Sigma^P_k \subseteq \Pi^P_{k+1}$.

(b) $\forall k \geqq 1$, $\Pi^P_k \subseteq BP \cdot \Sigma^P_k$ *implies* $PH = \Sigma^P_{k+1}$.

(c) $\forall k \geqq 0$, $\Sigma^P_2(BP \cdot \Sigma^P_k) = \Sigma^P_{k+2}$.

(d) $\forall k \geqq 1$, $\Sigma^P_2(BP \cdot \Sigma^P_k \cap BP \cdot \Pi^P_k) = \Sigma^P_{k+1}$.

(e) $PH = \bigcup_{k \geqq 0} BP \cdot \Sigma^P_k$.

Relationships between some of the above-mentioned classes are described in Fig. 1.

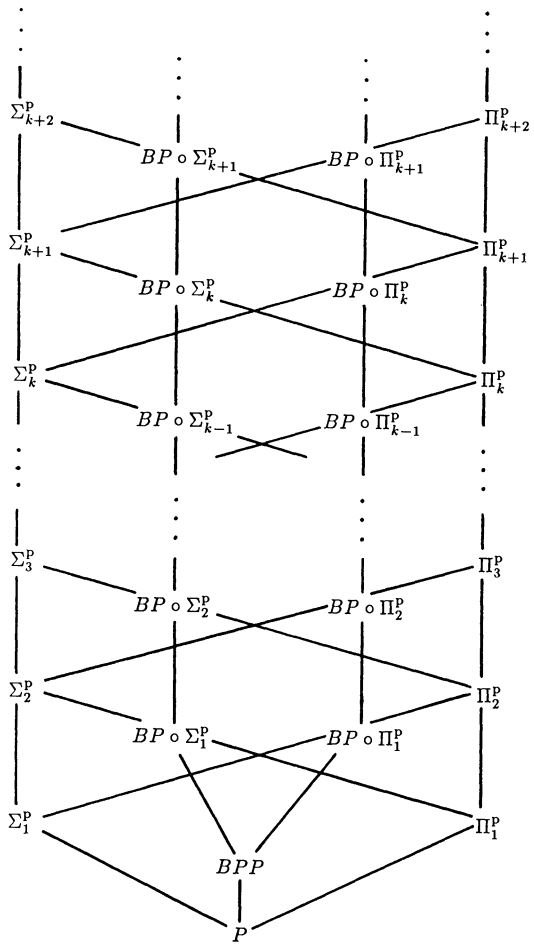


FIG. 1. *Polynomial hierarchy.*

PROPOSITION 2.6 [Ko82].
(a) $BPP(BPP) = BPP$.
(b) $NP \subseteq BPP$ *implies* $PH = BPP$.
PROPOSITION 2.7 [Za86], [ZF87].
(a) $\Sigma_1^P(BPP) \subseteq BP \cdot \Sigma_1^P \subseteq BPP(\Sigma_1^P)$.
(b) $BP \cdot \Sigma_1^P(BPP)\ (= AM(BPP)) = BP \cdot \Sigma_1^P$.

3. Random tally sets. One of the important tools in this paper is the notion of "random" tally sets. For every tally language $T \subseteq \{0\}^*$, let $\tau_T = b_0 b_1 b_2 b_3 \cdots \in \Sigma^\omega$ denote the characteristic sequence of T: $b_i = 1$ if and only if $0^i \in T$. Dually, for every $\alpha \in \Sigma^\omega$, let T_α denote the tally language with characteristic sequence α. For $z \in \Sigma^*$, T_z denotes T_α, where $\alpha = z0^\omega$. For $z \in \Sigma^*$ and $\alpha \in \Sigma^\omega$, $z \otimes \alpha$ denotes $z\beta$, where $\alpha = x\beta$ and $|x| = |z|$.

In the following, we will identify a tally language and its characteristic sequence: the power set of $\{0\}^*$ is identified with Σ^ω, and then any subset of Σ^ω is identified with a class of tally languages. Thus, $z \otimes T$ is well defined for word z and tally language T. For word z, let $\mathcal{R}_z = \{z\alpha \mid \alpha \in \Sigma^\omega\}$; thus, $\mathcal{R}_z$ is a "rectangular" infinite product in Σ^ω: $\mathcal{R}_z = \{z\}\Sigma^\omega = \{T \mid \tau_T = z\alpha \text{ for some } \alpha \in \Sigma^\omega\}$. Such an infinite product will be referred to as a *basic rectangle.*

In Σ, we define $\mu(\{0\}) = \mu(\{1\}) = \frac{1}{2}$. By taking the completion of the infinite product of this discrete measure (probabilistic) space, we have the measure of probability μ in Σ^ω. By identifying elements in Σ^ω and real numbers in the unit interval, μ can be interpreted as the Lebesgue measure in the unit interval. This means that for any fixed n, 0^n belongs to a random tally set T with probability $\frac{1}{2}$: $\mu(\{T \mid 0^n \in T\}) = \frac{1}{2}$. That is, we produce a random tally set T by an independent series of tosses of an unbiased coin: if the result of the nth tossing is "heads," then put 0^{n-1} into T; otherwise, do not put 0^{n-1} into T.

Since we identify a tally set and its characteristic sequence, we have $\mu(\mathcal{T}) = \mu(\{\alpha \in \Sigma^\omega \mid T_\alpha \in \mathcal{T}\})$ for a class $\mathcal{T}$ of tally sets. We will freely use either or both of these formulations in any given context.

For any string z of length n, $\mu(\mathcal{R}_z) = \frac{1}{2^n}$. For every n and m, $n \neq m$,

$$\begin{aligned}\mu(\{T \mid 0^n \in T, 0^m \in T\}) &= \mu(\{T \mid 0^n \in T, 0^m \notin T\}) \\ &= \mu(\{T \mid 0^n \notin T, 0^m \in T\}) \\ &= \mu(\{T \mid 0^n \notin T, 0^m \notin T\}) = \tfrac{1}{4},\end{aligned}$$

and for fixed T_0,

$$\mu(\{T \mid T_0 \subseteq T\}) = \mu(\{T \mid T_0 \cap T = \varnothing\}) = \begin{cases} 1/2^n & \text{when } \|T_0\| = n, \\ 0 & \text{when } \|T_0\| = \infty. \end{cases}$$

Recalling the notation for conditional probability, we have the following fact.

PROPOSITION 3.1. *For every integer* $m > 0$ *and predicate* P, $\Pr_m[y : P(y)] = \mu(\{y\alpha \mid |y| = m, \alpha \in \Sigma^\omega, \text{ and } P(y)\})$.

Recall that a class $\mathcal{C}$ of languages is closed under finite variation if for any $C \in \mathcal{C}$ and any finite set F, both $C \cup F$ and $C - F$ are in $\mathcal{C}$. Bennett and Gill [BG81] have claimed the following useful fact. (The following two facts are well known and can be found in any standard textbook on measure theory, e.g., [Ha50].)

PROPOSITION 3.2 (0-1 Law). *If* $\mathcal{T}$ *is a measurable class of tally sets that is closed under finite variation, then* $\mu(\mathcal{T}) = 0$ *or* $\mu(\mathcal{T}) = 1$.

Proof. Because $\mathcal{T}$ is closed under finite variation, we see that if $|y| = |z|$, then $\forall \alpha[y\alpha \in \mathcal{T} \Leftrightarrow z\alpha \in \mathcal{T}]$. Hence, $\mu(\mathcal{T} \cap \mathcal{R}_y) = \mu(\mathcal{T} \cap \mathcal{R}_z)$ if $|y| = |z|$; and so, $\mu(\mathcal{T} \cap \mathcal{R}_z) = \mu(\mathcal{T})\mu(\mathcal{R}_z)$ for all z. Thus, if $\mathcal{S}$ is a countable union of basic rectangles, then

$\mu(\mathcal{T}\cap\mathcal{S})=\mu(\mathcal{T})\mu(\mathcal{S})$. Since $\mathcal{T}$ is measurable, there exists a sequence $\mathcal{S}_1,\mathcal{S}_2,\cdots$ of countable unions of basic rectangles such that $\mathcal{S}_1\supseteq\mathcal{S}_2\supseteq\cdots\supseteq\mathcal{T}$ and $\mu(\mathcal{S}_n)\to\mu(\mathcal{T})$ as $n\to\infty$. Now, $\mu(\mathcal{T})=\mu(\mathcal{T}\cap\mathcal{S}_n)=\mu(\mathcal{T})\mu(\mathcal{S}_n)\to\mu(\mathcal{T})\mu(\mathcal{T})$ as $n\to\infty$. Thus, $\mu(\mathcal{T})=\mu(\mathcal{T})^2$. Therefore, $\mu(\mathcal{T})=0$ or $\mu(\mathcal{T})=1$. □

We will also use the following fact.

PROPOSITION 3.3. *If $\mathcal{T}$ is a class of tally sets such that $\mu(\mathcal{T})>0$, then there is a basic rectangle $\mathcal{R}_z$ such that $\mu(\mathcal{T}\cap\mathcal{R}_z)>(\frac{3}{4})\mu(\mathcal{R}_z)$.*

Proof. (Suppose that for all z, $\mu(\mathcal{T}\cap\mathcal{R}_z)\leqq(\frac{3}{4})\mu(\mathcal{R}_z)$. If $\mathcal{S}$ is a countable union of basic rectangles, then we have $\mu(\mathcal{T}\cap\mathcal{S})\leqq(\frac{3}{4})\mu(\mathcal{S})$. As in the proof of Proposition 3.2, $\mu(\mathcal{T})=\mu(\mathcal{T}\cap\mathcal{S}_n)\leqq(\frac{3}{4})\mu(\mathcal{S}_n)\to(\frac{3}{4})\mu(\mathcal{T})$ as $n\to\infty$. Hence, $\mu(\mathcal{T})\leqq(\frac{3}{4})\mu(\mathcal{T})$. Therefore $\mu(\mathcal{T})=0$, contradicting the hypothesis. □

Let $\mathcal{T}$ be a class of tally sets, or equivalently, a property for tally sets. If $\mu(\mathcal{T})=1$, we say that property $\mathcal{T}$ holds *for almost every tally set T.*

4. $BP\cdot\mathcal{C}$. In this section we investigate properties of classes of the form $BP\cdot\mathcal{C}$. Our first main result, Theorem 4.2, characterizes the assertion "$A\in BP\cdot\mathcal{C}$." Next, in Theorem 4.9, we show that $BP\cdot(\mathcal{C}\circ PF(T))=\mathcal{C}\circ PF(T)$ for almost every tally set T.

In the next section we apply these results to the situation where the class $\mathcal{C}$ is one of the well-studied complexity classes: P, NP, Σ_k^P, Δ_k^P, Π_k^P, PH, and PSPACE. Since we prove the results in this section with very few restrictions on the class $\mathcal{C}$, the results in the next section follow as simple corollaries with little or no additional proof needed.

As noted in § 2, we identify a tally set and its characteristic sequence so that we have $\mu(\mathcal{T})=\mu(\{\alpha\in\Sigma^\omega\mid T_\alpha\in\mathcal{T}\})$.

LEMMA 4.1. *Let $\mathcal{B}$ and $\mathcal{C}$ be countably infinite classes. For any set A, both of the classes $\{T\mid A\in\mathcal{C}\circ PF(T)\}$ and $\{T\mid\mathcal{B}\subseteq\mathcal{C}\cdot PF(T)\}$ are measurable and are closed under finite variation.*

Proof. It is easy to see that both of the classes are closed under finite variation. There are countably many sets in $\mathcal{C}$ and countably many oracle transducers so that it is sufficient to prove that for every $D\in\mathcal{D}$ and every $f\in PF(-)$, $\mathcal{T}=\{T\mid A=D\circ f^T\}$ is measurable.

Let $\mathcal{T}_n=\{\mathcal{T}\mid\forall x_{|x|\leqq n}[A(x)=D\circ f^T(x)]\}$. Then $\mathcal{T}_n$ is a finite union of basic rectangles since for all x with $|x|\leqq n$, $D\circ f^T(x)$ depends on only a finite prefix of T. Thus, $\mathcal{T}_n$ is measurable for all n. Note that $\mathcal{T}=\bigcap_{n\geqq0}\mathcal{T}_n$. Hence, $\mathcal{T}$ is measurable as desired.

That $\{\mathcal{T}\mid\mathcal{B}\subseteq\mathcal{C}\circ PF(T)\}$ is measurable follows from the fact that it is a countable intersection of classes $\{T\mid A\in\mathcal{C}\circ PF(T)\}$ for all $A\in\mathcal{B}$, and, as shown in the last paragraph, each such class is measurable. □

THEOREM 4.2. *If $\mathcal{C}$ is a countably infinite class that is closed under $\leqq^P_{\text{maj}}$, then $A\in BP\cdot\mathcal{C}$ if and only if $\mu(\{T\mid A\in\mathcal{C}\circ PF(T)\})=1$, that is, $A\in BP\cdot\mathcal{C}$ if and only if for almost every tally set T, $A\in\mathcal{C}\circ PF(T)$.*

Proof. Only if. Since $A\in BP\cdot\mathcal{C}$ and $\mathcal{C}$ is closed under $\leqq^P_{\text{maj}}$, the amplification lemma (Lemma 2.3) shows that there exist a polynomial $p(n)$ and a set $\mathcal{D}\in\mathcal{C}$ such that $\Pr_{p(n)}\{y\mid\forall x_{|x|=n}[A(x)=D(\langle x,y\rangle)]\}>1-2^{-(n+2)}$. Without loss of generality, we may assume that $p(n)\leqq p(n+1)$ and that if $|y|=p(|x|)$, then for all z, $D(\langle x,y\rangle)=D(\langle x,yz\rangle)$. Let $\mathcal{T}_n=\{\mathcal{T}_{y\alpha}\mid|y|=p(n),\quad\alpha\in\Sigma^\omega,\quad\forall x_{|x|=n}[A(x)=D(\langle x,y\rangle)]\}$. By Proposition 2.3,

$$\mu(\mathcal{T}_n)=\Pr_{p(n)}[y\mid\forall x_{|x|=n}[A(x)=D(\langle x,y\rangle)]]>1-2^{-(n+2)}.$$

Letting $\mathcal{T}=\bigcap_{n\geqq0}\mathcal{T}_n$, we have

$$\mathcal{T}=\bigcap_{n\geqq0}\mathcal{T}_n$$

$$= \{T_\alpha \mid \alpha \in \Sigma^\omega \text{ and } \forall x[A(x)$$
$$= D(\langle x, y\rangle)], \text{ where } y \text{ is the prefix of } \alpha \text{ of length } p(|x|)\}$$
$$= \{T \mid \forall x[A(x) = D(\langle x, T(\varepsilon)T(0)T(0^2)\cdots T(0^m)\rangle)], \text{ where } m = p(|x|)-1\}.$$

Then we have $\mu(\mathscr{T}^c) = \mu(\bigcup_{n\geq 0}\mathscr{T}_n^c) \leqq \sum_{n\geq 0}\mu(\mathscr{T}_n^c) < \sum_{n\geq 0}2^{-(n+2)} = \frac{1}{2}$. (Recall that $\mathscr{T}^c$ denotes $\{T \in TALLY \mid T \text{ is not in } \mathscr{T}\}$.) Thus, $\mu(\mathscr{T}) \geqq \frac{1}{2}$.

Let f be a polynomial time-bounded oracle transducer that has value $f^T(x) = \langle x, T(\varepsilon)T(0)T(0^2)\cdots T(0^m)\rangle$, where $m = p(|x|)-1$, i.e., $y = T(\varepsilon)T(0)T(0^2)\cdots T(0^m)$ is the prefix of length $m+1 = p(|x|)$ of τ_T. If $T \in \mathscr{T}$, then $\forall x[A(x) = D \circ f^T(x)]$; i.e., $A = D \circ f^T$. Hence, $\mu(\{T \mid A \in \mathscr{C} \circ PF(T)\}) \geqq \mu(\{T \mid A = D \circ f^T\}) \geqq \mu(\mathscr{T}) \geqq \frac{1}{2}$. By the 0-1 Law, this means that $\mu(\{T \mid A \in \mathscr{C} \circ PF(T)\}) = 1$.

If. Suppose that $\mu(\{T \mid A \in \mathscr{C} \circ PF(T)\}) = 1$. Then there exist a $D \in \mathscr{C}$ and a polynomial time-bounded oracle transducer f such that $\mu(\{T \mid A = D \circ f^T\}) > 0$.

Let $\mathscr{T} = \{T \mid A = D \circ f^T\}$. By Proposition 3.3, there is a basic rectangle $\mathscr{R}_z$ such that $\mu(\mathscr{T} \cap \mathscr{R}_z) > (\frac{3}{4})\mu(\mathscr{R}_z)$. Without loss of generality, we may assume that $\mu(\mathscr{T}) > \frac{3}{4}$ (for, consider transducer f' defined by $f'^T(x) = f^{z\otimes T}(x)$). Suppose that the running time of f is bounded above by the polynomial $p(n)$.

Let $C(\langle x, y\rangle) = D \circ f^{T_y}(x)$. Then $C \in \mathscr{C}$ because if g is defined by $g(\langle x\, y\rangle) = f^{T_y}(x)$, then $g \in PF$ and $C = D \circ g \in \mathscr{C} \circ PF = \mathscr{C}$. Thus, for all $x \in \Sigma^*$,

$$\begin{aligned}
&\Pr_{p(|x|)}[y \mid A(x) = C(\langle x, y\rangle)] \\
&\quad = \mu(\{T_{y\alpha} \mid |y| = p(|x|),\ \alpha \in \Sigma^\omega,\ A(x) = D \circ f^{T_y}(x) = D \circ f^{T_{y\alpha}}(x)\}) \\
&\quad = \mu(\{T_\beta \mid A(x) = D \circ f^{T_\beta}(x)\}) \geqq \mu(\{T_\beta \mid A = D \circ f^{T_\beta}\}) \\
&\quad = \mu(\{T \mid A = D \circ f^T\}) = \mu(\mathscr{T}) > \tfrac{3}{4}.
\end{aligned}$$

Hence, $A \in BP\cdot\mathscr{C}$. □

Recall that $A \in \mathscr{C}/\text{poly}$ if and only if $A \in \mathscr{C} \circ PF(T)$ for some tally set T (Lemma 2.2) and that $BP\cdot\mathscr{C} \subseteq \mathscr{C}/\text{poly}$ (Lemma 2.4). Comparing this fact with the above theorem, we see the difference between $BP\cdot\mathscr{C}$ and $\mathscr{C}/\text{poly}$.

COROLLARY 4.3. *If $\mathscr{C}$ is a countably infinite class and closed under $\leqq^P_{\text{maj}}$, then each of the following hold*:

(a) *$A \in BP\cdot\mathscr{C}$ if and only if $\mu(\{T \mid A \in \mathscr{C} \circ PF(T)\}) = 1$ if and only if $\mu(\{T \mid A \in \mathscr{C} \circ PF(T)\}) > 0$. That is, $A \in BP\cdot\mathscr{C}$ if and only if for almost every tally set T, $A \in \mathscr{C} \circ PF(T)$, and $A \notin BP\cdot\mathscr{C}$ if and only if for almost every tally set T, $A \notin \mathscr{C} \circ PF(T)$.*

(b) *$A \in PB\cdot\mathscr{C}$ if and only if there exists $C \in \mathscr{C}$ and $f \in PF(-)$ such that $\mu(\{T \mid A = C \circ f^T\}) > 0$.*

(c) *For any class $\mathscr{D}$, $\mathscr{D} \subseteq BP\cdot\mathscr{C}$ if and only if $\mu(\{T \mid \mathscr{D} \subseteq \mathscr{C} \circ PF(T)\}) = 1$ if and only if $\mu(\{T \mid \mathscr{D} \subseteq \mathscr{C} \circ PF(T)\}) > 0$; that is, $\mathscr{D} \subseteq BP\cdot\mathscr{C}$ if and only if for almost every tally set T, $\mathscr{D} \subseteq \mathscr{C} \circ PF(T)$.*

COROLLARY 4.4. *If $\mathscr{C}$ is a countably infinite class that is closed under $\leqq^P_{\text{maj}}$ and $\mathscr{D}$ is a countably infinite class, then $BP\cdot\mathscr{C} \subseteq \mathscr{D}$ implies $\mu(\{T \mid BP\cdot\mathscr{C} = \mathscr{D} \cap \mathscr{C} \circ PF(T)\}) = 1$.*

Proof. Let X denote the class $\mathscr{D} - BP \circ \mathscr{C}$. Since $BP\cdot\mathscr{C} \subseteq \mathscr{D}$, we have

$$\begin{aligned}
\{T \mid BP\cdot\mathscr{C} = \mathscr{D} \cap \mathscr{C} \circ PF(T)\} &= \bigcap_{A\in X}\{T \mid A \notin \mathscr{C} \circ PF(T) \text{ and } BP\cdot C \subseteq C \circ PF(T)\} \\
&= \bigcap_{A\in X}\{T \mid A \notin \mathscr{C} \circ PF(T)\} \cap \{T \mid BP\cdot\mathscr{C} \subseteq \mathscr{C} \circ PF(T)\}.
\end{aligned}$$

It follows from Corollary 4.3(a) and (c) that $\mu(\{T \mid A \notin \mathscr{C} \circ PF(T)\}) = 1$ for every $A \notin BP\cdot\mathscr{C}$ and that $\mu(\{T \mid BP\cdot\mathscr{C} \subseteq \mathscr{C} \circ PF(T)\}) = 1$. That is, the above class is a countable intersection of classes of measure one; hence, it has measure one. □

COROLLARY 4.5. *If $\mathscr{C}$ is countably infinite and closed under $\leqq^P_{\text{maj}}$, then* $(\mu \times \mu) \times (\{(T_1, T_2) \mid BP \cdot \mathscr{C} = \mathscr{C} \circ PF(T_1) \cap \mathscr{C} \circ PF(T_2)\}) = 1$.

Proof. From Corollary 4.3(c), $\mu(\{T_1 \mid BP \cdot \mathscr{C} \subseteq \mathscr{C} \circ PF(T_1)\}) = 1$; and from Corollary 4.4, for any T_1 such that $BP \cdot \mathscr{C} \subseteq \mathscr{C} \circ PF(T_1)$, $\mu(\{T_2 \mid BP \cdot \mathscr{C} = \mathscr{C} \circ PF(T_1) \cap C \circ PF(T_2)\}) = 1$. Then by Fubini's theorem [Ha50], $(\mu \times \mu) \times (\{T_1, T_2) \mid BP \cdot \mathscr{C} = \mathscr{C} \circ PF(T_1) \cap C \circ PF(T_2)\}) = 1$. □

Now we move to the next main theorem. In the proof of this theorem, we make use of the following technical lemma.

LEMMA 4.6. *For any T, let $L(T) \subseteq \Sigma^*$ denote a set determined by T. Let C and f be a set and a function in $PF(-)$, respectively. If for every T, $L(T) \leqq^P_{\text{maj}} C \circ f^T$ via some fixed $g \in PF$, then $L(T) = D \circ k^T$ for some $k \in PF(-)$ and $D \leqq^P_{\text{maj}} C$.*

Proof. Note that $L(T) \leqq^P_{\text{maj}} C \circ f^T$ via g means that if $g(x) = y_1 \# y_2 \# \cdots \# y_t$, then $x \in L(T)$ if and only if the majority of $f^T(y_1), f^T(y_2), \cdots, f^T(y_t)$ are in C.

Suppose that $q(n)$ and $r(n)$ are polynomials that bound the running times of f and g, respectively. Let $p(n) = q(r(n))$, so for each i and each T, $|f^T(y_i)| \leqq q(r(|x|)) = p(|x|)$. Let h be defined by $h(\langle x, z\rangle) = f^{T_z}(y_1) \# f^{T_z}(y_2) \# \cdots \# f^{T_z}(y_t)$, where $g(x) = y_1 \# y_2 \# \cdots \# y_t$. Then, $h \in PF$.

Let k be a polynomial time-bounded oracle transducer that on input x relative to oracle set T computes $\langle x, T(\varepsilon)T(0)T(0^2) \cdots T(0^m)\rangle$, where $m = p(|x|)$. Let $D = \{\langle x, z\rangle \mid$ the majority of the words in $h(\langle x, z\rangle)$ are in $C\} = \{\langle x, z\rangle \mid$ the majority of $f^{T_z}(y_1), \cdots, f^{T_z}(y_t)$ are in $C\}$. Then $D \leqq^P_{\text{maj}} C$ via h.

It is not difficult to show that for all T, $L(T) = D \circ k^T$: if $g(x) = y_1 \# y_2 \# \cdots \# y_t$, then $x \in D \circ k^T \Leftrightarrow k^T(x) \in D \Leftrightarrow \langle x, z\rangle = \langle x, T(\varepsilon)T(0)T(0^2) \cdots T(0^m)\rangle \in D$ (where $m = p(|x|)) \Leftrightarrow$ majority of $f^{T_z}(y_1) = f^T(y_1), \cdots, f^{T_z}(y_t) = f^T(y_t)$ are in $C \Leftrightarrow$ majority of $y_1, \cdots, y_t$ are in $C \circ f^T \Leftrightarrow x \in L(T)$. □

It is worth mentioning here that this lemma also proves that the closure property under $\leqq^P_{\text{maj}}$ of $\mathscr{C}$ is inherited by $\mathscr{C} \circ PF(T)$ and $BP \cdot \mathscr{C}$.

COROLLARY 4.7. *If $\mathscr{C}$ is closed under $\leqq^P_{\text{maj}}$, then so is $\mathscr{C} \circ PF(T)$ for every tally set T.*

Proof. Suppose that $A \leqq^P_{\text{maj}} C \circ f^T \in \mathscr{C} \circ PF(T)$ via $g \in PF$. For every tally set T_1, let $L(T_1) = \{x \mid$ the majority of words in $g(x)$ are in $C \circ f^{T_1}\}$. Hence, $A = L(T)$. By Lemma 4.6, there exist D and $k \in PF(-)$ such that $D \leqq^P_{\text{maj}} C$; hence $D \in \mathscr{C}$, and for every T_1, $L(T_1) = D \circ k^{T_1}$. Thus, $A = L(T) = D \circ k^T \in \mathscr{C} \circ PF(T)$, i.e., $\mathscr{C} \circ PF(T)$ is closed under $\leqq^P_{\text{maj}}$. □

COROLLARY 4.8. *If $\mathscr{C}$ is a countably infinite class that is closed under $\leqq^P_{\text{maj}}$, then so is $BP \cdot \mathscr{C}$.*

Proof. Suppose that $A \leqq^P_{\text{maj}} B$ for some $B \in BP \cdot \mathscr{C}$. Then it follows from Theorem 4.2 that $B \in \mathscr{C} \circ PF(T)$ for almost every tally set T. Since $A \leqq^P_{\text{maj}} B$ and $\mathscr{C} \circ PF(T)$ is closed under $\leqq^P_{\text{maj}}$, $A \in \mathscr{C} \circ PF(T)$ for almost every tally set T; again from Theorem 4.2, $A \in BP \cdot \mathscr{C}$. □

Now we have Theorem 4.9, our next main result of this section concerning the power of the BP-operator in tally relativizations. It is shown that the BP-operator does not increase the complexity of $\mathscr{C} \circ PF(T)$ for almost every tally set T. In other words, for almost every tally set T, $\mathscr{C} \circ PF(T)$ is a "fixed point" of the BP-operator.

THEOREM 4.9. *If $\mathscr{C}$ is countably infinite and closed under $\leqq^P_{\text{maj}}$, then $\mu(\{T \mid BP \cdot (\mathscr{C} \circ PF(T)) = \mathscr{C} \circ PF(T)\}) = 1$; that is, for almost every tally set T, $BP \cdot (\mathscr{C} \circ PF(T)) = \mathscr{C} \circ PF(T)$.*

Proof. By Corollary 4.7, we have that $\mathscr{C} \circ PF(T)$ is closed under $\leqq^P_{\text{maj}}$, for every tally set T; so, it is closed under padding. Thus, it follows from the fact concerning the BP-operator (see § 2) that $\mathscr{C} \circ PF(T) \subseteq BP \cdot (\mathscr{C} \circ PF(T))$.

In order to prove the reverse containment, let us first give some notation. For any set A and any polynomial $p(n)$, define $L_{\mathrm{maj}}(A, p(n))$ and $e(x, A, p(n))$ by

$$L_{\mathrm{maj}}(A, p(n)) = \{x \mid \Pr_{p(|x|)}[y: \langle x, y\rangle \in A] \geqq \tfrac{1}{2}\},$$
$$e(x, A, p(n)) = \min\{\Pr_{p(|x|)}[y: \langle x, y\rangle \in A], 1 - \Pr_{p(|x|)}[y: \langle x, y\rangle \in A]\}.$$

We call $e(x, A, p(n))$ the *error probability*. Note that for any tally set T,

$$BP\cdot(\mathscr{C} \circ PF(T)) = \bigcup_{p:\mathrm{poly}} \{L_{\mathrm{maj}}(C \circ f^T, p(n)) \mid C \in \mathscr{C}, f \in PF(-), \text{ and } \forall x[e(x, C \circ f^T, p(n)) < \tfrac{1}{4}]\}.$$

What we must prove is that for any $C \in \mathscr{C}$, any $f \in PF(-)$, and any polynomial $p(n)$,

$$\mu(\{T \mid \forall x[e(x, C \circ f^T, p(n)) < \tfrac{1}{4}] \Rightarrow [L_{\mathrm{maj}}(C \circ f^T, p(n)) \in \mathscr{C} \circ PF(T)]\}) = 1.$$

In the following, we shall see that this class is measurable.

For any choice of $C \in \mathscr{C}$, $f \in PF(-)$, and polynomial $p(n)$, we have the following fact.

CLAIM 1. For any integer $c > 0$, there exist $D \in \mathscr{C}$ and $g \in PF(-)$ such that for all x,

$$\mu(\{T \mid [e(x, C \circ f^T, p(n)) < \tfrac{1}{4}] \Rightarrow [x \in L_{\mathrm{maj}}(C \circ f^T, p(n)) \Leftrightarrow x \in D \circ g^T]\}) > 1 - 2^{-(c+2|x|)}.$$

Proof of Claim 1. Let $L(T) = \{\langle x, y_1, y_2, \cdots, y_t\rangle \mid t = 5(c+2|x|), \forall i_{1 \leqq i \leqq t} |y_i| = p(|x|),$ and the majority of the $\langle x, y_i\rangle$ are in $C \circ f^T\}$.

Obviously, $L(T) \leqq^P_{\mathrm{maj}} C \circ f^T$ via a fixed function in PF. By Lemma 4.6, there exist $D \in \mathscr{C}$ and $k \in PF(-)$ such that $L(T) = D \circ k^T$ for all T. It is easy to prove that for all x and T,

$$e(x, C \circ f^T, p(n)) < \tfrac{1}{4} \Rightarrow e(x, D \circ k^T, 5(c+2n)p(n)) < 2^{-(c+2|x|)}.$$

Let $q(n) = 5(c+2n)p(n)$. Suppose that $r(n)$ is a polynomial large enough such that we have the following:

(i) For all x and y, if $|x| = n$ and $|y| = q(n)$, then $k^T(\langle x, y\rangle)$ never queries the oracle about words of length greater than $r(n)$,

(ii) For all x and y, if $|x| = n$ and $|y| = p(n)$, then $f^T(\langle x, y\rangle)$ never queries the oracle about words of length greater than $r(n)$.

Let $h \in PF(-)$ behave as follows:

$$h^T(x) = \langle x, T(0^{(n)+1})T(0^{r(n)+2}) \cdots T(0^{r(n)+q(n)})\rangle \quad \text{where } n = |x|.$$

Let $g^T = k^T \circ h^T$. So $g \in PF(-)$.

We shall prove that D and g meet the requirement of Claim 1.

Note that $D \circ g^T(x) = D \circ k^T(\langle x, z\rangle)$ where $z = T(0^{r(n)+1})T(0^{r(n)+2}) \cdots T(0^{r(n)+q(n)})$, $n = |x|$.

Because $r(n)$ is large enough, we see that $e(x, C \circ f^T, p(n))$ and $e(x, D \circ k^T, q(n))$ depend only on $T^{\leqq r(|x|)}$. Thus, for any x, if we let $\mathscr{T} = \{T \mid [e(x, C \circ f^T, p(n)) < \tfrac{1}{4}] \Rightarrow [x \in L_{\mathrm{maj}}(C \circ f^T, p(n)) \Leftrightarrow x \in D \circ g^T]\}$, then we see that $\mathscr{T}$ is a finite union of basic rectangles. Since for any x and any T, $e(x, C \circ f^T, p(n)) < \tfrac{1}{4} \Rightarrow e(x, D \circ k^T, q(n)) < 2^{-(c+2|x|)}$, it follows that if $T^{\leqq r(|x|)}$ does not satisfy $e(x, C \circ f^T, (n)) < \tfrac{1}{4}$, then $T \in \mathscr{T}$. Furthermore, if $T^{\leqq r(|x|)}$ does satisfy $e(x, C \circ f^T, p(n)) < \tfrac{1}{4}$, then the next $q(|x|)$ bits in τ_T will determine whether $L_{\mathrm{maj}}(C \circ f^T, p(n))$ and $D \circ g^T$ agree on x. Only those situations that have the error probability $e(x, D \circ k^T, q(n))$ cause disagreement. Because $e(x, D \circ k^T, q(n)) < 2^{-(c+2|x|)}$, the ratio of the number of T's that agree (i.e., belong to $\mathscr{T}$) to the total number is greater than $1 - 2^{-(c+2|x|)}$.

Hence, $\mu(\mathscr{T}) > 1 - 2^{-(c+2|x|)}$. This concludes the proof of Claim 1.

CLAIM 2. For any integer $c > 0$, there exist $D \in C$ and $g \in PF(-)$ such that

$$\mu(\{T \mid \forall x[e(x, C \circ f^T, p(n)) < \tfrac{1}{4}] \Rightarrow [L_{\mathrm{maj}}(C \circ f^T, p(n)) = D \circ g^T]\}) > 1 - 2^{-c}.$$

Proof of Claim 2. The set $\{T \mid \forall x[e(x, C \circ f^T, p(n)) < \frac{1}{4}] \Rightarrow [L_{\text{maj}}(C \circ f^T, p(n)) = D \circ g^T]\}$ contains the intersection over all x of all of the classes studied in Claim 1. Thus, we have Claim 2.

To complete the proof of the theorem, note that for any integer $c > 0$,

$$\mu(\{T \mid \forall x[e(x, C \circ f^T, p(n)) < \tfrac{1}{4}] \Rightarrow [L_{\text{maj}}(C \circ f^T, p(n)) \in \mathscr{C} \circ PF(T)]\}) > 1 - 2^{-c}.$$

Since c is chosen arbitrarily, we have

$$\mu(\{T \mid \forall x[e(x, C \circ f^T, p(n)) < \tfrac{1}{4}] \Rightarrow [L_{\text{maj}}(C \circ f^T, p(n)) \in \mathscr{C} \circ PF(T)]\}) = 1. \qquad \square$$

COROLLARY 4.10. *If $\mathscr{C}$ is countably infinite and closed under $\leqq^P_{\text{maj}}$, then for almost every tally set T, $(BP \cdot \mathscr{C}) \circ PF(T) = BP \cdot (\mathscr{C} \circ PF(T))$.*

Proof. First, $(BP \cdot \mathscr{C}) \circ PF(T) \subseteq (\mathscr{C} \circ PF(T_1)) \circ PF(T)$ for every tally set T and almost every tally set T_1, since $BP \cdot \mathscr{C} \subseteq \mathscr{C} \circ PF(T_1)$ for almost every tally set T_1.

Second, $(\mathscr{C} \circ PF(T_1)) \circ PF(T) = (\mathscr{C} \circ PF(T)) \circ PF(T_1)$ for every tally set T_1 and T, since $PF(T_1) \circ PF(T_2) = PF(T_1 \oplus T_2)$ for every tally set T_1 and T_2.

By Corollary 4.3(c), we have $(BP \cdot \mathscr{C}) \circ PF(T) \subseteq BP \cdot (\mathscr{C} \circ PF(T))$ for every tally set T. Thus, $\mathscr{C} \circ PF(T) \subseteq (BP \cdot \mathscr{C}) \circ PF(T) \subseteq BP \cdot (\mathscr{C} \circ PF(T))$ for every tally set T. By Theorem 4.9, for almost every tally set T, $\mathscr{C} \circ PF(T) = BP \circ (\mathscr{C} \circ PF(T))$. $\square$

Although we have the equality $(BP \cdot \mathscr{C}) \circ PF(T) = BP \cdot (\mathscr{C} \circ PF(T))$ for almost every tally set T, it is left open whether this equality holds for all tally sets.

5. Polynomial complexity classes. In this section we have the main results on classes such as Σ^P_k, Π^P_k, Δ^P_k, *PH*, and PSPACE. To a large extent the results follow from Theorem 5.2 below, and the proof of that result is immediate from the corollaries of Theorems 4.2 and 4.9 and Lemma 5.1. The proof of Lemma 5.1 is immediate from the definition of the $\mathscr{C} \circ \mathscr{F}$ and the fact that every tally set is "self-P-printable," i.e., for every tally set T, there is a polynomial time-bounded oracle transducer that on input 0^n, $n \geqq 0$, will compute relative to T the list of all strings in T of length at most n.

LEMMA 5.1. *Let T be a tally set and let $\mathscr{C}$ be any of the classes Σ^P_k, Π^P_k, Δ^P_k, PH, or* PSPACE, *$k \geqq 0$. For any set L, $L \in \mathscr{C}(T)$ if and only if there exists $f \in PF(T)$ and $D \in \mathscr{C}$ such that $(\forall x)[x \in L \Leftrightarrow f(x) \in D]$. That is, for every tally set T, $\mathscr{C}(T) = \mathscr{C} \circ PF(T)$.*

Because each of the classes Σ^P_k, Π^P_k, Δ^P_k, *PH*, and PSPACE is countable and closed under $\leqq^P_{\text{pos}}$, and hence is closed under $\leqq^P_{\text{maj}}$; all of the results in § 4 can be applied to these classes. Thus, we can focus our attention on the interpretation of the results developed in § 4 in the context of these classes. The main result is the next two theorems.

THEOREM 5.2. *For every integer $k \geqq 0$, each of the following hold*:

(a) *For every set A, $A \in BP \cdot \Sigma^P_k$ if and only if for almost every tally set T, $A \in \Sigma^P_k(T)$.*

(b) *For any countable class $\mathscr{D}$, $\mathscr{D} \subseteq BP \cdot \Sigma^P_k$ if and only if for almost every tally set T, $\mathscr{D} \subseteq \Sigma^P_k(T)$.*

(c) *For almost every pair (T_1, T_2) of tally sets, $BP \cdot \Sigma^P_k = \Sigma^P_k(T_1) \cap \Sigma^P_k(T_2)$.*

(d) *For any countable class $\mathscr{D}$ such that $BP \cdot \Sigma^P_k \subseteq \mathscr{D}$, $BP \cdot \Sigma^P_k = \mathscr{D} \cap \Sigma^P_k(T)$ for almost every tally set.*

Proof. In each case the proof follows from Lemma 5.1 and the corresponding part of Corollary 4.3. $\square$

The above statements characterize the class $BP \cdot \Sigma^P_k$ from several points of view. Parts (a) and (b) of this theorem are basic characterizations and are used in the following discussion. Part (c) concerns the problem of finding a minimal pair [Ro67] in the context of complexity theory. Ambos-Spies [Am86] and Kurtz [Ku87] have shown the existence of a minimal pair for the class *BPP* (with respect to Turing reductions). Part (c) states the existence of a *tally* minimal pair for each $BP \cdot \Sigma^P_k$ with respect to Σ^P_k-reductions.

For example, by letting $k=0$ or 1 in this theorem, we have the following characterizations of BPP $(=BP\cdot P)$ and AM $(=BP\cdot NP)$.

COROLLARY 5.3. (a) *For every set A, $A\in BPP$ if and only if for almost every tally set T, $A\in P(T)$.*

(b) *For almost every pair (T_1, T_2) of tally sets, $BPP=P(T_1)\cap P(T_2)$, so (T_1, T_2) is a $\leqq_T^P$-minimal pair for BPP.*

(c) *For any countable class $\mathscr{D}$ such that $BPP\subseteq\mathscr{D}$, $BPP=\mathscr{D}\cap P(T)$ for almost every tally set.*

(d) *For every set A, $A\in AM$ if and only if for almost every tally set T, $A\in NP(T)$.*

(e) *For almost every pair (T_1, T_2) of tally sets, $AM=NP(T_1)\cap NP(T_2)$, so (T_1, T_2) is a $\leqq_T^{NP}$-minimal pair for AM.*

(f) *For any countable class $\mathscr{D}$ such that $AM\subseteq\mathscr{D}$, $AM=\mathscr{D}\cap NP(T)$ for almost every tally set.*

THEOREM 5.4. *Let k be any nonnegative integer. For almost every tally set T, $BP\cdot\Sigma_k^P(T)=\Sigma_k^P(T)$.*

Proof. The proof is immediate from Lemma 5.1 and Theorem 4.9. □

COROLLARY 5.5. (a) *For almost every tally set T, $BPP(T)=P(T)$.*

(b) *For almost every tally set T, $AM(T)=NP(T)$.*

In Theorems 5.2 and 5.4, we can uniformly replace Σ_k^P by Π_k^P or by Δ_k^P, and all of the resulting statements will be true.

What follows is applications of the above two theorems. Here we improve or generalize the known facts (see Propositions 2.5–2.7 for the summary of previous works).

It is known that $BPP(BPP)=BPP$ [Ko82] and that $BP\cdot NP(BPP)=BP\cdot NP$, i.e., $AM(BPP)=AM$[ZF87]. Theorems 5.2 and 5.4 yield the following generalization.

THEOREM 5.6. *For all $k, j\geqq 0$, the following hold*:

(a) $BP\cdot\Sigma_k^P(BP\cdot\Sigma_j^P)=BP\cdot\Sigma_{k+j}^P$.

(b) $BP\cdot\Sigma_k^P(BP\cdot\Delta_{j+1}^P)=BP\cdot\Sigma_{k+j}^P$.

(c) $BP\cdot\Delta_{k+1}^P(BP\cdot\Delta_{j+1}^P)=BP\cdot\Delta_{k+j+1}^P$.

Proof. First we prove (a). By Theorem 5.2(b), $BP\cdot\Sigma_j^P\subseteq\Sigma_j^P(T)$ for almost every T. Hence $BP\cdot\Sigma_k^P(BP\cdot\Sigma_j^P)\subseteq BP\cdot\Sigma_k^P(\Sigma_j^P(T))=BP\cdot\Sigma_{k+j}^P(T)$ for almost every T. By Theorem 5.4, $BP\cdot\Sigma_{k+j}^P(T)=\Sigma_{k+j}^P(T)$ for almost every T. Therefore, we have

$$BP\cdot\Sigma_k^P(BP\cdot\Sigma_j^P)\subseteq\Sigma_{k+j}^P(T) \quad \text{for almost every } T.$$

So $BP\cdot\Sigma_k^P(BP\cdot\Sigma_j^P)\subseteq BP\cdot\Sigma_{k+j}^P$ by Theorem 5.2(b). On the other hand,

$$BP\cdot\Sigma_{k+j}^P=BP\cdot\Sigma_k^P(\Sigma_j^P)\subseteq BP\cdot\Sigma_k^P(BP\cdot\Sigma_j^P).$$

This completes the proof of (a).

The proofs of (b) and (c) are similar. □

Schöning has proved that $\Sigma_2^P(BP\cdot\Sigma_k^P)=\Sigma_{k+2}^P$. Here we have something a little stronger. Theorem 5.6 yields $BP\cdot NP(BP\cdot\Sigma_k^P)=BP\cdot\Sigma_{k+1}^P$ so that $NP(BP\cdot\Sigma_k^P)\subseteq BP\cdot\Sigma_{k+1}^P$.

THEOREM 5.7. *For every $k\geqq 0$, the following hold*:

(a) $BP\cdot\Sigma_{k+1}^P\subseteq BP\cdot\Sigma_k^P$ *implies* $PH=BP\cdot\Sigma_k^P$.

(b) $BP\cdot\Sigma_{k+1}^P\subseteq BP\cdot\Delta_{k+1}^P$ *implies* $PH=BP\cdot\Delta_{k+1}^P$.

Proof. (a) $BP\cdot\Sigma_{k+2}^P=BP\cdot\Sigma_1^P(BP\cdot\Sigma_{k+1}^P)\subseteq BP\cdot\Sigma_1^P(BP\cdot\Sigma_k^P)=BP\cdot\Sigma_{k+1}^P\subseteq BP\cdot\Sigma_k^P$. By induction on i, we can prove $BP\cdot\Sigma_{k+1}^P\subseteq BP\cdot\Sigma_i^P$.

(b) the proof of (b) is similar to that of (a). □

In Theorem 5.7, Σ_k^P can be uniformly replaced by Π_k^P, and the resulting statements will still hold. Note that Theorem 5.7(a) is a generalization of Proposition 2.6(b): $NP\subseteq BPP$ implies $BP\cdot NP\subseteq BPP$, and taking $k=0$ in Theorem 5.7(a), we have $BP\cdot NP\subseteq BPP$ implies $PH=BPP$.

Schöning has proved that $\Sigma_2^P(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)=\Sigma_{k+1}^P$. We improve this in the following way.

THEOREM 5.8. *For $k\geqq 1$, $BP\cdot NP(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)=BP\cdot\Sigma_k^P$.*

Proof. From Theorem 5.2(b), we see that for almost every tally set T, $BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P\subseteq\Sigma_k^P(T)\cap\Pi_k^P(T)$. Since for every tally set T, $NP(\Sigma_k^P(T)\cap\Pi_k^P(T))=\Sigma_k^P(T)$, we have $NP(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)\subseteq NP(\Sigma_k^P(T)\cap\Pi_k^P(T))=\Sigma_k^P(T)$ for almost every tally set T. Thus, $NP(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)\subseteq BP\cdot\Sigma_k^P$. So $BP\cdot NP(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)\subseteq BP\cdot\Sigma_k^P$. On the other hand, $BP\cdot NP(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)\supseteq BP\cdot NP(\Sigma_k^P\cap\Pi_k^P)=BP\cdot\Sigma_k^P$. □

Using this theorem, we can strengthen Theorem 5.7 in the case where $k\geqq 1$.

THEOREM 5.9. *For every $k\geqq 1$,*

(a) *$BP\cdot\Pi_k^P\subseteq BP\cdot\Sigma_k^P$ implies $PH=BP\cdot\Sigma_k^P$;*

(b) *$BP\cdot\Sigma_k^P\subseteq BP\cdot\Pi_k^P$ implies $PH=BP\cdot\Pi_k^P$.*

Proof. Because $BP\cdot\Pi_k^P\subseteq BP\cdot\Sigma_k^P\Leftrightarrow BP\cdot\Pi_k^P=BP\cdot\Sigma_k^P\Leftrightarrow BP\cdot\Sigma_k^P\subseteq BP\cdot\Pi_k^P$, it is enough to prove (a).

By Theorem 5.6(a), we have $BP\cdot\Sigma_{k+1}^P=BP\cdot NP(BP\cdot\Sigma_k^P)$. By assumption and Theorem 5.8, $BP\cdot NP(BP\cdot\Sigma_k^P)=BP\cdot NP(BP\cdot\Sigma_k^P\cap BP\cdot\Pi_k^P)=BP\cdot\Sigma_k^P$. Hence, $BP\cdot\Sigma_{k+1}^P=BP\cdot\Sigma_k^P$. By Theorem 5.7(a), we have $PH=BP\cdot\Sigma_k^P$. □

Note that $BP\cdot\Pi_k^P\subseteq BP\cdot\Sigma_k^P\Leftrightarrow\Pi_k^P\subseteq BP\cdot\Sigma_k^P$. Theorem 5.9 is stronger than the corresponding result of Schöning, i.e., Proposition 2.5(c), since it forces PH to collapse to $BP\cdot\Sigma_k^P$ instead of Σ_{k+1}^P.

Note $BP\cdot\Pi_k^P\subseteq BP\cdot\Sigma_k^P$ if and only if for almost every T, $\Pi_k^P\subseteq\Sigma_k^P(T)$. Yap [Yap83] has proved that if there is a tally set T such that $\Pi_k^P\subseteq\Sigma_k^P(T)$, then $PH=\Sigma_{k+2}^P$. The above theorem says that if there are "lots of" such T's, then $PH=BP\cdot\Sigma_k^P$, i.e., PH collapses to a lower level.

All of the results above relativize to arbitrary oracle sets, that is, the results hold for classes such as $\Sigma_k^P(A)$, $\Pi_k^P(A)$, $\Delta_k^P(A)$, and $PH(A)$ for arbitrary A. All that is needed is that Lemma 5.1 remain valid for these classes.

Reviewing the consequences of Theorems 5.2 and 5.4, i.e., Theorems 5.6–5.9, we may note the similarity between the polynomial-time hierarchy and its probabilistic version, i.e., $\{BP\cdot\Sigma_k^P\}_{k\geqq 0}$. For example, since $BP\cdot PH=BP\cdot(\bigcup_{k\geqq 0}\Sigma_k^P)=\bigcup_{k\geqq 0}(BP\cdot\Sigma_k^P)=PH$, the following well-known facts [St77] are faithfully reflected in the corresponding statements of those theorems.

PROPOSITION 5.10. *For every $k,j\geqq 0$ and $l\geqq 1$, the following hold:*

(a) *$\Sigma_k^P(\Sigma_j^P)=\Sigma_{k+j}^P$.*

(b) *$\Sigma_{k+1}^P\subseteq\Sigma_k^P$ implies $PH=\Sigma_k^P$.*

(c) *$NP(\Sigma_l^P\cap\Pi_l^P)=\Sigma_l^P$.*

(d) *$\Pi_l^P\subseteq\Sigma_l^P$ implies $PH=\Sigma_l^P$.*

Our technique for proving the above theorems is also interesting. Note that Theorems 5.6–5.9 state the relationships in unrelativized settings. Nevertheless, the properties established for relativized complexity classes, i.e., Theorems 5.2 and 5.4, yield simple proofs for those theorems.

As a final remark of this section, we show the application of Theorem 4.2 to higher complexity classes.

THEOREM 5.11. *For every set A, the following hold:*

(a) *$A\in PH$ if and only if for almost every tally set T, $A\in PH(T)$ if and only if for all tally sets T, $A\in PH(T)$.*

(b) *For almost every pair (T_1,T_2) of tally sets, $PH=PH(T_1)\cap PH(T_2)$.*

(c) *For any countable class $\mathscr{D}$ such that $PH\subseteq\mathscr{D}$, $PH=\mathscr{D}\cap PH(T)$ for almost every tally set.*

Remark. The same results hold by replacing PH by PSPACE.

Acknowledgments. The authors are grateful to Professor Ronald Book for his support and encouragement. Comments from and discussions with Professors Book, Ker-I Ko, and Christopher Wilson have been very helpful. The ideas in this paper are deeply affected by the work of Ambos-Spies [Am86] and Bennett and Gill [BG81]. The authors are indebted to Ms. Leslie Wilson for her excellent preparation of the manuscript.

REFERENCES

[Am86] K. AMBOS-SPIES, *Randomness, relativizations and polynomial reducibilities*, in Proc. 1st Annual Conference on Structure in Complexity Theory, Lecture Notes in Computer Science 223, Springer-Verlag, 1986, Berlin, New York, pp. 23-34.

[Ba85] L. BABAI, *Trading group theory for randomness*, in Proc. 17th Annual ACM Symposium on Theory of Computing, 1985, pp. 421-429.

[Ba87] ———, *Random oracle separates* PSPACE *from the polynomial-time hierarchy*, Inform. Proc. Letts., 26 (1987/88), pp. 51-53.

[BBS86] J. BALCÁZAR, R. BOOK, AND U. SCHÖNING, *The polynomial-time hierarchy and sparse oracles*, Assoc. Comput. Mach., 33 (1986), pp. 603-617.

[BGS75] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativization of the* P = ? NP question, SIAM J. Comput., 4 (1975), pp. 431-442.

[BG81] C. H. BENNETT AND J. GILL, *Relative to a random oracle,* $P^A \neq NP^A \neq$ co-NP^A *with probability* 1, SIAM J. Comput., 10 (1981), pp. 96-113.

[BH77] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of* NP *and other complete sets*, SIAM J. Comput., 16 (1977), pp. 305-322.

[Ca86] J. CAI, *With probability one, a random oracle separates* PSPACE *from the polynomial-time hierarchy*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 21-29.

[Gi77] J. GILL, *Computational complexity of probabilistic Turing machines* , SIAM J. Comput., 6 (1977), pp. 675-695.

[GS86] S. GOLDWASSER AND M. SIPSER, *Private coins in interactive proof system*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 59-68.

[Ha50] P. HALMOS, *Measure Theory*, Van Nostrand, New York, 1950.

[Kä87] J. KÄMPER, *Non-uniform proof systems: a new framework to describe non-uniform and probabilistic complexity classes*, manuscript.

[KL80] R. M. KARP AND R. J. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th Annual ACM Symposium on Theory of Computing, 1980, pp. 302-309.

[Ko82] K. KO, *Some observations on probabilistic algorithms and* NP*-hard problems*, Inform. Proc. Lett., 14 (1982), pp. 39-43.

[Ku87] S. A. KURTZ, *A note on randomized polynomial time*, SIAM J. Comput., 16 (1987), pp. 852-853.

[LS86] T. LONG AND A. SELMAN, *Relativizing complexity classes with sparse oracles*, J. Assoc. Comput. Mach., 33 (1986), pp. 618-627.

[NW88] N. NISAN AND A. WIGDERSON, *Hardness vs. randomness*, manuscript.

[Ro67] H. ROGERS, JR., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967.

[Sc87] U. SCHÖNING, *Probabilistic complexity classes and lowness*, in Proc. 2nd Annual Conference on Structure in Complexity Theory, 1987, pp. 2-8.

[St77] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.

[Ta87] S. TANG, in preparation.

[TW88] S. TANG AND O. WATANABE, *On tally relativizations of* BP*-complexity classes*, in Proc. 3rd Annual Conference on Structure in Complexity Theory, 1988, to appear.

[Wr77] C. WRATHALL, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23-33.

[Ya85] A. YAO, *Separating the polynomial-time hierarchy by oracles*, in Proc. XXth Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 1-10.

[Yap83] C. K. YAP, *Some consequences of non-uniform conditions on uniform classes*, Theoret. Comput. Sci., 26 (1983), pp. 287-300.

[Za86] S. ZACHOS, *Probabilistic quantifiers, adversaries, and complexity classes*, in Proc. 1st Conference on Structure in Complexity Theory, Lecture Notes in Computer Science 223, Springer-Verlag, Berlin, New York, 1986, pp. 383-400.

[ZF87] S. ZACHOS AND M. FURER, *Probabilistic quantifiers vs. distrustful adversaries*, manuscript.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 463-472, June 1989

003

DYNAMIC PROGRAMMING BY EXCHANGEABILITY*

SHUO-YEN ROBERT LI†

Abstract. This article introduces the concept of *exchangeable stopping time* and a technique of dynamic programming based upon this concept for fast computation of the expected value of a payoff function upon stopping. One instance of an exchangeable stopping time is when a stopping time is defined by a threshold on the sequential sum of the process. Another instance is when a stopping time is defined by the occurrence of given patterns in observed values of the process. This new computation technique has applications in bin packing, casino blackjack, and random drawing for patterns.

Key words. bin packing, casino blackjack, dynamic programming, exchangeability, fast computation, permutation groups

AMS(MOS) subject classifications. 68Q20, 60G09, 90C39, 60G40

1. Introduction. Let $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ be a discrete stochastic process with a stopping time N. Let p be a function defined over all finite integer sequences, called the *payoff* function. We want to compute $E[p(\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_N)]$, i.e., the average payoff upon stopping. Assuming $N, \mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ are uniformly bounded, the computation can always be done by exhaustive search. This would require the computation of the probability of every sample path of the stopping process and the evaluation of the payoff function at the end of each path.

Computation of this type often can be simplified by techniques derived from fundamental structures in probability theory such as Markov chains, martingales, independently and identically distributed random variables, etc. When the stochastic process, the stopping time, and the payoff function do not meet requirements for invoking such fundamental techniques, we still hope to do the computation more efficiently than exhaustive search. This motivates the exploration of new concepts that give rise to efficient computation techniques but not necessarily close-form solutions. This article introduces one such concept called the *exchangeable stopping time* and the technique of *dynamic exchangeable programming.* Over a process of exchangeable random variables, the expected value of a symmetric payoff function upon an exchangeable stopping time can be computed by this new technique, which is exponentially faster than exhaustive search. Examples of applications of this fast computation technique are found in bin packing, casino blackjack, and random drawing for patterns.

2. Exchangeability and computation.

2.1. Taking advantage of symmetry in computation. Throughout the article, a process means a *finite or infinite* stochastic process and a stopping time is always *nonrandomized* (in the sense of Wald [8]). Let Σ denote a fixed finite set of nonnegative integers. Unless otherwise specified, a stochastic process always means a sequence of Σ-valued random variables. A finite sequence of elements from the set Σ will be called a *tuple.* In particular, the null tuple means the null sequence. Following DeFinetti [3], we define a process to be *exchangeable,* if rearrangements of random variables do not change the process. In other words, a stochastic process is exchangeable if the probability of any tuple is independent of the ordering among entries in the tuple.

* Received by the editors February 17, 1988; accepted for publication August 19, 1988.
† Bell Communications Research, 435 South Street, Morristown, New Jersey 07960.

Consider a process $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ with a stopping time N. A tuple $(x_1, x_2, \cdots, x_n)$ is said to be N-attainable if

$$P\{\mathbf{X}_1 = x_1, \mathbf{X}_2 = x_2, \cdots, \mathbf{X}_n = x_n, \text{ and } n \leqq N\} > 0. \tag{2.1}$$

Given a payoff function defined over tuples, we want to compute the expected payoff upon stopping but avoid repetitive computation on symmetric tuples. This, of course, would require the property that *the ordering of the random variables in the process is immaterial.* What does this property precisely mean and how does a computation algorithm take advantage of it? It should mean the "exchangeability" of the random variables in three regards. First, the random variables are exchangeable with regard to chain probabilities, i.e., they form an exchangeable process. Second, they are exchangeable with regard to the payoff function, i.e., the payoff function is symmetric. Finally, they are exchangeable with regard to the stopping time.

The meaning of this third exchangeability needs clarification. One possible interpretation could be that all symmetric tuples are either N-attainable together or non-N-attainable together. This interpretation is too strict to be useful because it essentially implies that the stopping time N is independent of the stochastic process. Another possible interpretation could be that, when two symmetric tuples are both known to be N-attainable, the stochastic process stops at both of them or at neither of them. Note that, for a given tuple, the prefix of a permutation of the tuple is not necessarily a permutation of a prefix. Therefore the latter interpretation of an exchangeable stopping time would be weak in linking the N-attainability among symmetric tuples. However, we shall use this interpretation of an exchangeable stopping time as the principal condition in the definition and compensate its weakness by side conditions.

2.2. Exchangeable stopping time. Let S_n denote the group of permutations on the numbers $1, 2, \cdots, n$. For $0 \leqq m < n$, we regard S_m as a subgroup of S_n. The product of two permutations a and b is denoted as $a \times b$. As a composite function over the numbers $1, 2, \cdots, n$, this product is interpreted as the function a followed by the function b. (It could be interpreted as b followed by a instead, as long as the interpretation is consistent throughout all definitions pertaining to products of permutations in this article.)

Consider a process $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ with a stopping time N. For every tuple $(x_1, x_2, \cdots, x_n)$, write

$$A(x_1, x_2, \cdots, x_n) = \{\sigma \in S_n \mid (x_{\sigma(1)}, x_{\sigma(2)}, \cdots, x_{\sigma(n)}) \text{ is } N\text{-attainable}\}. \tag{2.2}$$

We say that N is an *exchangeable* stopping time if the following two properties hold:

(A) When two symmetric tuples are both N-attainable, the stochastic process stops at both of them or at neither of them.

(B) For any nondecreasing tuple $(x_1, x_2, \cdots, x_n)$, if $A(x_1, x_2, \cdots, x_n)$ is nonempty, then so is $A(x_1, x_2, \cdots, x_{n-1})$.

The property (B) has been stated with respect to the natural ordering among integers in the set Σ. But, since elements of Σ can be relabeled, the property can actually be stated with respect to any other ordering as well. It has turned out that this technical condition is usually easy to meet.

The goal of introducing the concept of exchangeable stopping times is to derive a computation algorithm that avoids repetitive computation on symmetric tuples. Naturally, the algorithm will require some mechanism to quantify the symmetry among N-attainable tuples. Since our definition of the exchangeability of a stopping time is very weak in linking the N-attainability among symmetric tuples, the application of the algorithm would require more than just the exchangeability. The required extra

property of the stopping time can be a reasonably computable $|A(x_1, \cdots, x_n)|$ for certain tuples $(x_1, \cdots, x_n)$. It turns out that the reasonable computability of the following quantity would also suffice: Write

$$a(x_1, \cdots, x_n) = \frac{|A(x_1, \cdots, x_n)|}{|A(x_1, \cdots, x_{n-1})|}, \tag{2.3}$$

whenever the denominator is nonzero, where $|\ |$ is the usual notation for cardinality. Instances of exchangeable stopping times with easily computable $a(x_1, \cdots, x_n)$ are presented in § 3.

2.3. The computation algorithm. A fast computation algorithm based upon exchangeability will be given in § 2.3.2. First, § 2.3.1 describes the computation by exhaustive search so that the new algorithm can be described in parallel terms for easy contrast.

2.3.1. Dynamic programming. Let $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ be a process with a stopping time N and p a function defined over tuples. Define another function over tuples as follows:

$$D(x_1, \cdots, x_n) = E[p(\mathbf{X}_1, \cdots, \mathbf{X}_N) | \mathbf{X}_1 = x_1, \cdots, \mathbf{X}_n = x_n]. \tag{2.4}$$

In particular, the expected value of p upon stopping is $D(\ )$, i.e., the function D evaluated at the null tuple, which can be calculated from "backward induction" as follows. If $n = N$ for the tuple $(x_1, \cdots, x_n)$, then

$$D(x_1, \cdots, x_n) = p(x_1, \cdots, x_n). \tag{2.5}$$

If $N > n-1$ for the tuple $(x_1, \cdots, x_{n-1})$, then

$$D(x_1, \cdots, x_{n-1}) = \sum_{x_n} P\{\mathbf{X}_n = x_n | \mathbf{X}_1 = x_1, \cdots, \mathbf{X}_{n-1} = x_{n-1}\} D(x_1, \cdots, x_n). \tag{2.6}$$

The computation based upon this recursion is equivalent to the exhaustive search through a tree where each node is a tuple and an ancestor of a node means a prefix. This computation algorithm will be referred to as *dynamic programming.*

2.3.2. Dynamic exchangeable programming. Assume that $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ is an exchangeable process with an exchangeable stopping time N and that p is a symmetric function defined over tuples. Define the *stabilizer group* of a nondecreasing tuple $(x_1, x_2, \cdots, x_n)$ as follows:

$$G(x_1, \cdots, x_n) = \{\sigma \in S_n | (x_{\sigma(1)}, \cdots, x_{\sigma(n)}) = (x_1, \cdots, x_n)\}. \tag{2.7}$$

This group measures repetition of values in the tuple. Define the function

$$g(x_1, \cdots, x_n) = \frac{|G(x_1, \cdots, x_n)|}{|G(x_1, \cdots, x_{n-1})|} \tag{2.8}$$

over *nondecreasing* tuples of any nonzero length. Then g satisfies the following easy recursion:

$$g(x_1, \cdots, x_n) = 1 \qquad \text{if } n = 1 \text{ or } x_n > x_{n-1}, \tag{2.9}$$

$$g(x_1, \cdots, x_n) = g(x_1, \cdots, x_{n-1}) + 1 \quad \text{if } n \geqq 2 \text{ and } x_n = x_{n-1}. \tag{2.10}$$

Let a tuple be called a *stus-tuple* if it is symmetric to an upon-stopping tuple. Our goal is to compute the expected value of the function p upon stopping. Without loss of generality, hereafter throughout this section, we shall assume the following:

(C) p can have nonzero values only at stus-tuples.

We want to construct an algorithm that is similar to the exhaustive tree search in the above-described dynamic programming but avoids repetitive computation on symmetric tuples. Thus, consider all *nondecreasing* tuples that are symmetric to N-attainable tuples. We organize these tuples into a tree structure by linking each of them to its prefix with one less element, i.e., linking $(x_1, \cdots, x_n)$ to $(x_1, \cdots, x_{n-1})$. The requirement (B) of the exchangeability of N guarantees the connectivity of this tree structure.

All *external nodes* (also called *leaves*) of this tree and possibly some *internal nodes* are stus-tuples; we shall call them *stus-nodes*. To compute the expected value of p upon stopping, we need only to evaluate p at stus-nodes and enumerate the number of upon-stopping tuples symmetric to it. From the requirement (A) of the exchangeability of N, the number of upon-stopping tuples symmetric to a stus-node is the same as the number of N-attainable tuples symmetric to it. Note that $|A(x_1, \cdots, x_n)|/|G(x_1, \cdots, x_n)|$ is the number of *distinct* N-attainable tuples that are symmetric to $(x_1, \cdots, x_n)$. Therefore, by the assumption (C), we have

$$E[p(\mathbf{X}_1, \cdots, \mathbf{X}_N)] = \sum_{x_1 \leqq \cdots \leqq x_n} \frac{|A(x_1, \cdots, x_n)|}{|G(x_1, \cdots, x_n)|} P\{(\mathbf{X}_1, \cdots, \mathbf{X}_n) = (x_1, \cdots, x_n)\} p(x_1, \cdots, x_n). \tag{2.11}$$

This formula can be computed by tree search according to the following recursion. If $(x_1, \cdots, x_n)$ is an external node of the tree, let

$$F(x_1, \cdots, x_n) = p(x_1, \cdots, x_n)|A(x_1, \cdots, x_n)|. \tag{2.12}$$

If $(x_1, \cdots, x_{n-1})$ is an internal node of the tree, then let

$$\begin{aligned} F(x_1, \cdots, x_{n-1}) &= p(x_1, \cdots, x_{n-1})|A(x_1, \cdots, x_{n-1})| \\ &\quad + \sum_{x_n \geqq x_{n-1}} \frac{F(x_1, \cdots, x_n)}{g(x_1, \cdots, x_n)} P\{\mathbf{X}_n = x_n | \mathbf{X}_1 = x_1, \cdots, \mathbf{X}_{n-1} = x_{n-1}\}. \end{aligned} \tag{2.13}$$

We can then recursively compute $F(\ )$, which equals the expected value of p upon stopping. This algorithm involves the computation of $|A(x_1, \cdots, x_n)|$ on every stus-node $(x_1, \cdots, x_n)$.

Recall the definition (2.3) of $a(x_1, \cdots, x_n)$. If $|A(x_1, \cdots, x_n)|$ is computable, then, of course, $a(x_1, \cdots, x_n)$ can be computed with little extra effort. But the converse is not always true. Also, even when $|A(x_1, \cdots, x_n)|$ is computable, $a(x_1, \cdots, x_n)$ is often a much simpler number to compute. For these reasons, we prefer to compute the expected value of p upon stopping by the following recursion instead of (2.12) and (2.13): If $(x_1, \cdots, x_n)$ is an external node of the tree, let

$$M(x_1, \cdots, x_n) = p(x_1, \cdots, x_n). \tag{2.14}$$

If $(x_1, \cdots, x_{n-1})$ is an internal node of the tree, let

$$\begin{aligned} M(x_1, \cdots, x_{n-1}) &= p(x_1, \cdots, x_{n-1}) \\ &\quad + \sum_{x_n \geqq x_{n-1}} \frac{a(x_1, \cdots, x_n)}{g(x_1, \cdots, x_n)} P\{\mathbf{X}_n = x_n | \mathbf{X}_1 = x_1, \cdots, \mathbf{X}_{n-1} = \mathbf{X}_{n-1}\} M(x_1, \cdots, x_n). \end{aligned} \tag{2.15}$$

Then the expected value of p upon stopping equals $M(\ )$. This computation algorithm will be referred to as *dynamic exchangeable programming*.

2.3.3. Comparison of computational efficiency. Dynamic exchangeable programming applies only under the exchangeability assumptions, but, when applicable, its computation can be significantly faster than dynamic programming. Dynamic programming invokes its recursive formula (2.6) for every tuple before stopping, and evaluates the payoff at every tuple upon stopping. Dynamic exchangeable programming invokes its recursive formula (2.15) only for nondecreasing rearrangements of tuples before stopping, and evaluates the payoff only at nondecreasing rearrangements of tuples upon stopping. Except for the extra effort in the computation of $a(x_1, \cdots, x_n)$, the speedup factor of dynamic exchangeable programming over dynamic programming is roughly the ratio between the number of upon-stopping tuples and the number of nondecreasing upon-stopping tuples. Thus the speedup factor may be estimated as follows. For a fixed length n, there are $|\Sigma|^n$ tuples, while there are only $(n+|\Sigma|)!/n!|\Sigma|!$ nondecreasing tuples; in terms of the order in the parameter n, the ratio between these two numbers is $|\Sigma|^n$. In view of the convexity of the exponential function, we then estimate the speedup factor to be of the order of $|\Sigma|^{E[N]}$.

The computational complexity of $a(x_1, \cdots, x_n)$ is very much up to the individual instance of the exchangeable stopping time N. For example, if the exchangeability of N is due to Theorem 3.1 below, then the computational complexity depends solely upon the structure of the monotonic function f in the theorem. In particular, if N is as in Corollary 3.2, this computation would be a very minor effort incurred by the algorithm.

An alternative implementation of dynamic programming is to tabulate values of the payoff function at nondecreasing tuples and then look up the table whenever the function needs to be evaluated. This would save one aspect of the repetitive computation, i.e., the evaluation of the function at symmetric tuples. However, the implementation requires an algorithm for the translation of a tuple into the proper table address. A desirable algorithm should take only a reasonable amount of time and space. Conceivably, the number of tuples can be quite large in applications and there may not exist any natural way of storing nondecreasing tuples in a compact linear fashion. Hence, the desired translation algorithm is often unavailable in applications.

3. Quantification of symmetry. Dynamic exchangeable programming requires the computation of $a(x_1, \cdots, x_n)$ for nondecreasing tuples $(x_1, \cdots, x_n)$. In this section, we investigate structures of exchangeable stopping times that lead to natural ways of computing $a(x_1, \cdots, x_n)$. Then we present examples of applications of dynamic exchangeable programming.

3.1. Computation of $a(x_1, \cdots, x_n)$. For any two sets A and B of permutations, let $A \times B$ denote the set of permutations of the type $a \times b$, where a, b are elements of A, B, respectively. For $n \geqq 1$, let $\mathbf{F}_n$ be an n-element subset of S_n such that

$$S_n = S_{n-1} \times \mathbf{F}_n. \tag{3.1}$$

In other words, $\mathbf{F}_n$ is a complete set of representatives of right *cosets* of S_n over the subgroup S_{n-1}.

Our property of the stopping time that enhances the computability of $a(x_1, \cdots, x_n)$ is a natural expression of the set $A(x_1, \cdots, x_n)$ in the form of a "direct product" of the set $A(x_1, \cdots, x_{n-1})$ with another subset of S_n. A special form of this property is that, for every nondecreasing tuple $(x_1, \cdots, x_n)$, there exists a subset $H(x_1, \cdots, x_n)$ of $\mathbf{F}_n$ satisfying the equation

$$A(x_1, \cdots, x_n) = A(x_1, \cdots, x_{n-1}) \times H(x_1, \cdots, x_n), \tag{3.2}$$

because, then, $a(x_1, \cdots, x_n)$ is simply the cardinality of $H(x_1, \cdots, x_n)$. When a stopping time is exchangeable and possesses this property, we shall say that the stopping time is exchangeable with respect to the sequence $\mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \cdots$. The simplest example of the set $\mathbf{F}_n$ consists of *transpositions*:

$$\mathbf{F}_n = \{(1\ \ n), (2\ \ n), \cdots, (n\ \ n)\}. \tag{3.3}$$

When a stopping time is exchangeable with respect to the particular sequence $\mathbf{F}_n$ defined by (3.3), we say that it is *exchangeable with respect to transpositions.*

Let a function f over tuples be called *monotonic* if

$$f(x_1, \cdots, x_n) \leqq f(y_1, \cdots, y_n) \leqq f(y_1, \cdots, y_n, y_{n+1}) \tag{3.4}$$

whenever $x_1 \leqq y_1$, $x_2 \leqq y_2, \cdots, x_n \leqq y_n$, and $n \geqq 0$.

THEOREM 3.1. *Let f be a symmetric monotonic function on tuples and K a constant. Define the stopping time N on the process $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$, as the smallest number n such that*

$$f(\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_n) \geqq K. \tag{3.5}$$

Then N is exchangeable with respect to transpositions.

Proof. Given a nondecreasing tuple $(x_1, x_2, \cdots, x_n)$, we need to verify the condition (3.2) with a properly chosen subset $H(x_1, x_2, \cdots, x_n)$ of the set $\mathbf{F}_n$ in (3.3). If $f(x_1, x_2, \cdots, x_{n-1}) \geqq K$, then $A(x_1, x_2, \cdots, x_n)$ is null due to the monotonicity and the symmetry of f. We may therefore assume that $f(x_1, x_2, \cdots, x_{n-1}) < K$. Then the symmetry of f implies that $A(x_1, x_2, \cdots, x_{n-1}) = S_{n-1}$. Define m as the smallest index such that

$$f(x_1, x_2, \cdots, \hat{x}_m, \cdots, x_n) < K. \tag{3.6}$$

Here $\hat{}$ is the usual notation for deleting an item from an array. An element σ of S_n belongs to $A(x_1, x_2, \cdots, x_n)$ if and only if

$$f(x_{\sigma(1)}, x_{\sigma(2)}, \cdots, x_{\sigma(n-1)}) < K, \tag{3.7}$$

which is equivalent to that $\sigma(n) \geqq m$. This means that

$$A(x_1, x_2, \cdots, x_n) = S_{n-1} \times \{(m\ \ n), (m+1\ \ n), \cdots, (n\ \ n)\}. \tag{3.8}$$

The condition (3.2) is verified by choosing $H(x_1, x_2, \cdots, x_n)$ to be the set $\{(m\ \ n), (m+1\ \ n), \cdots, (n\ \ n)\}$, and the theorem is proved.

For the stopping time in the theorem, we have $a(x_1, \cdots, x_n) = n - m + 1$, where m is defined by (3.6). In other words, the computation of $a(x_1, \cdots, x_n)$ is equivalent to the computation of m, which directly relates to the structure of the symmetric monotonic function f. One simple and useful case of a symmetric monotonic function is as follows.

COROLLARY 3.2. *Given a constant K and a nonnegative nondecreasing function f on the set Σ, define the stopping time N as the smallest index n such that*

$$f(\mathbf{X}_1) + \cdots + f(\mathbf{X}_n) \geqq K. \tag{3.9}$$

Then N is exchangeable with respect to transpositions.

Proof. Expand the domain of the function f from Σ to tuples by

$$f(x_1, \cdots, x_m) = f(x_1) + \cdots + f(x_m). \tag{3.10}$$

Then f is a symmetric monotonic function on tuples.

3.2. Application to bin packing. A collection of items of various sizes are to be packed into bins. The number of items of each size is predetermined. All items are

mixed in a random order and revealed one at a time. A bin collects items one by one according to the order of arrival until the accumulative load reaches a predetermined threshold. Then a second bin starts to be loaded by subsequent items in the same manner, and so on. In other words, a new bin is started *after* overflow occurs. This differs from the usual bin packing, where a new bin is started *before* overflow occurs (see Coffman, Garey, and Johnson [2] for a comprehensive survey and Rhee [6] for a recent reference on bin packing).

To demonstrate the application of dynamic exchangeable programming, we shall compute the variance and the covariance of bin loads. When there are enough items to guarantee the filling of a fixed number, say k, of bins, then the loads of these k bins are identically distributed (and, in fact, form an exchangeable process). We want to compute the average load of the first bin, the average square of the load of the first bin, and the average product of the first two loads.

Thus, let $\mathbf{X}_i$ represent the size of the ith item. This defines an exchangeable process. Define the stopping time N as the smallest n such that $\mathbf{X}_1+\cdots+\mathbf{X}_n$ reaches the given threshold of bin load. Note that this is the special case of Corollary 3.2 with the function f being the identity function. Therefore the quantity $a(x_1, \cdots, x_n)$ is easily computable. To compute the average load of the first bin and the average square, the payoff functions are, respectively,

$$p_1(x_1, \cdots, x_n) = x_1+\cdots+x_n \tag{3.11}$$

and

$$p_2(x_1, \cdots, x_n) = (x_1+\cdots+x_n)^2. \tag{3.12}$$

To compute the average product of the first two loads, we use the same stopping time N and the payoff function:

$$p_3(x_1, \cdots, x_n) = E[\text{product of two loads} \mid \mathbf{X}_1 = x_1, \cdots, \mathbf{X}_n = x_n]. \tag{3.13}$$

At an upon-stopping tuple $(x_1, \cdots, x_n)$ the evaluation of $p_3(x_1, \cdots, x_n)$ can be done by exhaustive search through all possible values of $\mathbf{X}_{n+1}$, $\mathbf{X}_{n+2}, \cdots$ given that $\mathbf{X}_1 = x_1, \cdots, \mathbf{X}_n = x_n$. Alternatively, this evaluation can be done through a recursive application of dynamic exchangeable programming, since the conditional process of $\mathbf{X}_{n+1}$, $\mathbf{X}_{n+2}, \cdots$ given that $\mathbf{X}_1 = x_1, \cdots, \mathbf{X}_n = x_n$ can be regarded as the initial process in the bin-packing problem where those items corresponding to $x_1, \cdots, x_n$ are deducted from the initial collection.

3.3. Computation of casino blackjack. Since the pioneering work of Thorp [7], there have been numerous publications analyzing and simulating strategies for casino blackjack. A partial list of these studies is the bibliography of Humble and Cooper [4]. These studies, except for relatively simple ones, have all been based upon statistical simulations and estimations instead of full-confidence computation. One reason for avoiding computation is the immense computational complexity. In principle, for any version of the game rules, any shuffle practices of the house, and any strategy of the gambler, the exact expected return at any situation can be exhaustively computed. However, the complexity of blackjack computation often overwhelms the processing power of modern-day computers.

Consider the computation of the expected payoff of the *exhaustively optimal* strategy for playing *one* game *head-on* against the house. The net payoff depends on minor variations of game rules from house to house and can sometimes be extremely close to zero. There are many situations that one option for the gambler (such as *hit*,

stand, *split*, and *double-down*) may be better than another option by a margin so tiny that only the exact computation can make the right decision. This makes the simulation of the exhaustively optimal strategy difficult.

Dynamic exchangeable programming offers an alternative to exhaustive computation or simulations. The random variables in the exchangeable process are cards received by the gambler. Given the value of dealer's *up-card*, let K and KS be the minimum *hard* and *soft* points, respectively, that are not known to be sure hits. For example, when a game starts with a fresh deck, KS is 19 against an up-card 9 or 10 and is 18 against all others. The exchangeable process stops when either the hard total is at least K or the soft total is at least KS. (For the consideration of special stopping due to double-down, split, or *natural*, the readers are referred to Li [5].) The payoff function, at a standing hand, is defined as the average receipt to be credited to the hand, which is averaged over all possibilities of dealer's remaining cards. The payoff function, at any tuple upon stopping (not necessarily a standing hand yet), is the average receipt to be credited to the hand, assuming that the gambler plays optimally from that point on.

This would have set up the problem as a typical application of Corollary 3.2 except for the ambiguity of an ace being counted as either 1 or 11, and hence also the ambiguity in the meaning of a nondecreasing tuple. We classify tuples upon stopping into the following types:

(A) All cards are non-aces.

(B) There is exactly one ace and one non-ace.

(C) There is exactly one ace. The total points of non-aces are greater than or equal to K.

(D) There is exactly one ace and at least two non-aces. The total points of non-aces are less than K.

(E) There are at least two aces and at least one non-ace.

Assign point values to aces as follows. The ace in a type (B), (C), or (D) hand is counted as 11 points, and each ace in a type (E) hand is counted as one point. Figure 1 is a *finite-state diagram* that depicts the growth from the null tuple to the nondecreasing permutation of a tuple upon stopping. Note that $2 \leqq y_1 \leqq y_2 \leqq \cdots \leqq 10$ and $1 \leqq z_1 \leqq z_2 \leqq \cdots \leqq 10$ in Fig. 1. Application of dynamic exchangeable programming to this stopping process can be done in the same fashion as Corollary 3.2, because the stopping times for exiting both loops in Fig. 1 are defined by thresholds on the sequential sum.

3.4. Random drawing for patterns. Let the process $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ represent *Polya's urn scheme* (see, for example, Chung [1]), where the "urn" contains a collection of numbers with repetitions allowed. A specific instance of this process would be drawing cards from a deck (full or partial) without replacement. Consider the type of stopping times that are defined by the occurrence of certain "patterns of combinations." For example, define the stopping time N as when any number appears for a third time or when there are two repeated numbers. A stopping time of this type is clearly exchangeable. It is usually not difficult to compute the function $a(x_1, \cdots, x_n)$ for such a stopping time and thereby apply dynamic exchangeable programming to the averaging of any symmetric payoff function upon stopping. Below, we compute the function $a(x_1, \cdots, x_n)$ for the particular stopping time N.

Recall the function g over nondecreasing tuples in (2.9) and (2.10). Write

$$b(x_1, \cdots, x_n) = \prod_{i=1}^{n} g(x_1, \cdots, x_i). \tag{3.14}$$

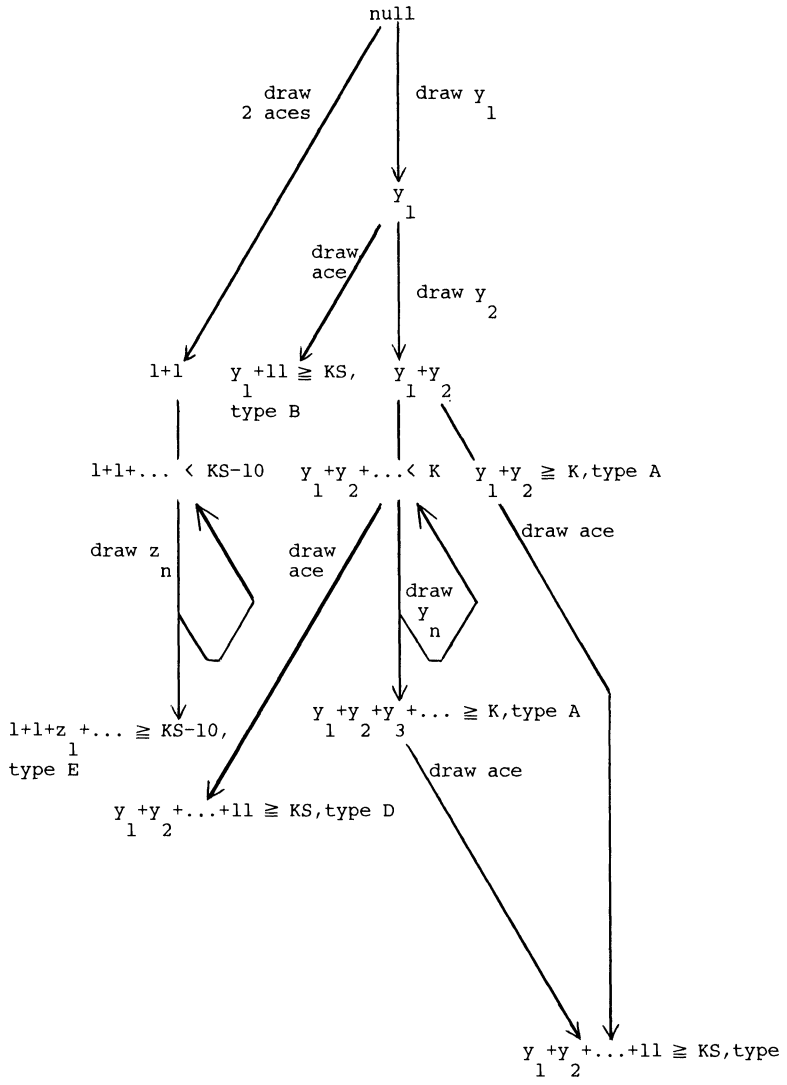


FIG. 1

Since the function g can be computed by an easy recursion, so can the function b. The function a relates to the function b as follows:

$$
\begin{aligned}
a(x_1, \cdots, x_n) = n \quad & \text{if } b(x_1, \cdots, x_n) \leqq 2, \\
4 \quad & \text{if } b(x_1, \cdots, x_{n-1}) = 2 \text{ and } b(x_1, \cdots, x_n) = 4 \\
3 \quad & \text{if } b(x_1, \cdots, x_{n-1}) = 2 \text{ and } b(x_1, \cdots, x_n) = 6, \\
n-1 \quad & \text{if } 4 \leqq b(x_1, \cdots, x_{n-1}) = b(x_1, \cdots, x_n) \leqq 6, \\
0 \quad & \text{if } b(x_1, \cdots, x_n) \geqq 8.
\end{aligned} \tag{3.15}
$$

The proof of (3.15) is relatively elementary, and hence is omitted.

REFERENCES

[1] K. L. CHUNG, *Elementary Probability Theory with Stochastic Processes*, Springer-Verlag, Berlin, New York, 1974.

[2] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Approximation algorithms for bin packing—an updated survey*, in Algorithm Design for Computer System Designs, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, Berlin, New York, 1984, pp. 49–106.

[3] B. DEFINETTI, *Theory of Probability: A Critical Introduction Treatment*, John Wiley, New York, 1974–75.

[4] L. HUMBLE AND C. COOPER, *The World's Greatest Blackjack Book*, Doubleday, New York, 1980.

[5] S.-Y. R. LI, *Factorizable stopping time and blackjack computation*, Tech. Memorandum, TM-ARH-002-170, Bell Communications Research, Morristown, NJ, August, 1986.

[6] W. T. RHEE, *Probabilistic analysis of the next fit decreasing algorithm for bin-packing*, Oper. Res. Lett., 6 (1987), pp. 189–191.

[7] E. O. THORP, *Beat the Dealer*, Blaisdell, New York, 1962.

[8] A. WALD, *Sequential Analysis*, John Wiley, New York, 1947.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 473-486, June 1989

004

OPTIMAL BIN PACKING WITH ITEMS OF RANDOM SIZES III *

WANSOO T. RHEE† AND MICHEL TALAGRAND‡

Abstract. Consider a probability measure μ on $[0, 1]$ and independent random variables $X_1, \cdots, X_n$ distributed according to μ. Let $Q_n = Q_n(X_1, \cdots, X_n)$ be the minimum number of unit-size bins needed to pack items of size $X_1, \cdots, X_n$. Let $c(\mu) = \lim_{n\to\infty} E(Q_n)/n$. In this paper it is proved that the random variable $(Q_n - nc(\mu))/\sqrt{n}$ converges in distribution. The limit is identified as a distribution of the supremum of a certain Gaussian process canonically attached to μ.

Key words. optimal stochastic bin packing, weak convergence

AMS(MOS) subject classifications. 90B99, 60K30

1. Introduction. The bin-packing problem requires finding the minimum number of unit-size bins needed to pack a given collection of items with sizes $X_1, \cdots, X_n$ in $[0, 1]$. This problem has many applications and has been shown to be NP-complete [Kar]. Consider a probability measure μ on $[0, 1]$. (No regularity assumption is made on μ.) Consider n items with sizes $X_1, \cdots, X_n$ which are independent identically distributed random variables distributed according to μ. (For simplicity, we denote by X_k both item names and item sizes.) We let $Q_n = Q_n(X_1, \cdots, X_n)$ denote the minimum number of unit-size bins needed to pack $X_1, \cdots, X_n$. It is well known that Q_n is a subadditive process (see [Kin]). So we have $\lim_n Q_n/n = c(\mu)$ almost surely for some constant $c(\mu)$ depending on μ, and $E(Q_n)/n \geqq c(\mu)$ for each n. In [Rhe3], the authors have shown that Q_n is very concentrated around its expectation. More precisely, for all $t > 0$,

$$P(|Q_n - E(Q_n)| \geqq t) \leqq 2 \exp(-t^2/2n).$$

This however gives no information on the value of $E(Q_n)$. In [Rhe2], the authors have shown that for some universal constant K, we have

$$0 \leqq E(Q_n) - nc(\mu) \leqq K(n(1+\log n))^{1/2}. \tag{1.1}$$

(Throughout the paper, a universal constant means a number independent of all the data of the problem. We will always denote by K a universal constant, not necessarily the same at each line.)

The present paper will go one step further in the description of Q_n. We will describe a random variable T_n, whose distribution is independent of n, and for which $E(|(Q_n - nc(\mu))/\sqrt{n} - T_n|) = o(1)$. This implies in particular that $(Q_n - nc(\mu))/\sqrt{n}$ converges in distribution. Some deterministic estimates were essential in the proof of (1.1); a sharpening of these results (Theorem A below) will be essential here. We now proceed to a precise description of our results. The ideas and concepts developed in our earlier paper [Rhe2] will be essential.

For $k \geqq 1$, let

$$R_k = \left\{ (x_1, \cdots, x_k) \in R^k : 0 \leqq x_1 \leqq \cdots \leqq x_k, \sum_{i \leqq k} x_i \leqq 1 \right\}.$$

* Received by the editors March 19, 1987; accepted for publication (in revised form) June 15, 1988. This research was supported in part by National Science Foundation grant DCR-8610255.

† Faculty of Management Sciences, Ohio State University, Columbus, Ohio 43210.

‡ Université de Paris VI, Equipe d'Analyse, Tour 46, 4 Place Jussieu, 75230 Parix Cedex 05, France.

For $x \in [0,1]$, we denote by δ_x the probability measure concentrated at x, that is for a Borel set G we have $\delta_x(G)=1$ if x belongs to G and $\delta_x(G)=0$ otherwise. For a compact metric space S, we denote by $M_1(S)$ the set of probability measures on S.

The following improves on [Rhe2], Theorem B.

THEOREM A. *Consider a nonnegative sequence* $(\alpha_k)_{k\geqq 0}$ *with* $\sum_{k\geqq 0}\alpha_k = 1$, *and a sequence* $\nu_k \in M_1(R_k)$. *Consider the measure*

$$\mu = \alpha_0\delta_0 + \sum_{k\geqq 1}(\alpha_k/k)\int_{R_k}\sum_{i\leqq k}\delta_{x_i}\,d\nu_k(x_1,\cdots,x_k).$$

For $l \geqq 1$, *let* $\beta_l = \sum_{k\geqq l}\alpha_k/k$. *Consider a sequence* $s_1,\cdots,s_n$ *of items. Let*

$$D = D(s_1,\cdots,s_n) = \operatorname*{Sup}_{0\leqq t\leqq 1}\{\operatorname{card}\{i\leqq n;\ s_i \geqq t\} - n\mu([t,1])\},$$

so $D \geqq 0$. *Then for all* $1 \leqq q \leqq n$, *we have*

$$Q_n(s_1,\cdots,s_n) \leqq D + n\sum_{k\geqq 1}(\alpha_k/k) + n/q^2 + 3q + 4 + K(\log n)^{1/2}\left(\sum_{1\leqq l\leqq q}\operatorname{Max}((n\beta_l/l)^{1/3},(n\beta_l)^{1/4})\right) \tag{1.2}$$

where K *is a universal constant* (i.e., *independent of* μ, n, *and* $s_1,\cdots,s_n$). *In particular,*

$$Q_n(s_1,\cdots,s_n) \leqq D + n\sum_{k\geqq 1}(\alpha_k/k) + Kn^{5/11}(1+\log n)^{1/2}. \tag{1.3}$$

Comments. It is sufficient (but essential) for our present purposes to have an error term in (1.3) that is of order $< n^{1/2}$. We do not know however what would be the best possible order of this error term. Since we see no reason why the present bound should be optimal, we have not conducted our computations in order to attempt to get a small value for K and we have always used crude but simple estimates.

PROPOSITION B. (a) *Let* $s_1,\cdots,s_n$ *be in* $[0,1]$. *Let* $\nu = (1/n)\sum_{i\leqq n}\delta_{s_i}$. *Then we have*

$$nc(\nu) \leqq Q_n(s_1,\cdots,s_n) \leqq nc(\nu) + Kn^{5/11}(1+\log n)^{1/2}. \tag{1.4}$$

(b) *For each distribution* μ,

$$nc(\mu) \leqq E(Q_n(X_1,\cdots,X_n)) \leqq nc(\mu) + Kn^{1/2}.$$

For a probability measure γ on $[0,1]$, we set $F_\gamma^-(t) = \gamma([0,t))$, $F_\gamma(t) = \gamma([0,t])$. We denote by $W_t = B_t - tB_1$ the Brownian bridge, where B_t is the standard Brownian motion (see [Bre]). For the simplicity of notation, we set

$$W_{\mu,t} = W_{\mu([0,t])},\quad W_{\mu,t}^- = W_{\mu([0,t))}.$$

Consider the set $\mathscr{C}$ of nondecreasing functions f from $[0,1]$ to $[0,1]$ that satisfy $f(0)=0$ and $f(a)+f(b) \leqq f(a+b)$ whenever $a, b \in [0,1]$. $a+b \leqq 1$.

To each f in $\mathscr{C}$, we associate f^- given by $f^-(0)=0$, and for $x>0$ by $f^-(x) = \lim_{y\to x, y<x} f(y)$, so $f^- \leqq f$, and it is easily seen that $f^- \in \mathscr{C}$. There is a unique positive measure γ_f on $[0,1]$ such that $\gamma_f(\{1\})=0$ and that $F_{\gamma_f}^-(t) = f^-(t)$ for each $0 \leqq t \leqq 1$, and γ_f has mass $f^-(1) \leqq 1$. We now fix μ in $M_1([0,1])$. We denote by $(z_i)_{i\in J}$ an enumeration of the points $z \in [0,1]$ such that $\mu(\{z\}) > 0$, where J is either empty, finite or countable infinite.

THEOREM C. *The set*

$$\mathscr{C}_\mu = \left\{ f \in \mathscr{C};\ \int f\,d\mu = c(\mu) \right\}$$

is nonempty. Consider the random variable

$$T = \operatorname*{Sup}_{f \in \mathscr{C}_\mu} \left[-\int_0^1 W_{\mu,t}\, d\gamma_f(t) + \sum_{i \in J} (f(z_i) - f^-(z_i))(W_{\mu,z_i} - W^-_{\mu,z_i}) \right]. \tag{1.5}$$

Then, for each n, one can find a random variable T_n distributed as T, and such that

$$\lim_{n\to\infty} E(|(Q_n(X_1, \cdots, X_n) - nc(\mu))/\sqrt{n} - T_n|) = 0.$$

Remarks. (1) The proof actually shows that for some $\alpha > 0$, we have

$$\lim_{n\to\infty} E(\exp(\alpha|(Q_n(X_1, \cdots, X_n) - nc(\mu))/\sqrt{n} - T_n|)) = 1.$$

(2) T is the supremum of a Gaussian process indexed by $\mathscr{C}_\mu$.

(3) The reason for the unintuitive formula (1.5) is explained in the heuristic proof of Theorem C given at the beginning of § 3.

COROLLARY D. *We have*

$$\lim_{n\to\infty} E((Q_n(X_1, \cdots, X_n) - nc(\mu))/\sqrt{n}) = 0$$

if and only if for any two functions f_1 and f_2 of $\mathscr{C}_\mu$, then $f_1 - f_2 = 0$ on the support of μ, except maybe on a countable set of μ-measure zero (that may depend on f_1 and f_2). (We recall that the support of a probability measure is the smallest closed set A for which $\mu(A) = 1$.)

Comments. While it is somehow surprising that results as accurate as Theorem C can be proved, it must be pointed out that the structure of the set $\mathscr{C}$ is very complicated and seems to contain a lot of the difficulty that is inherent to bin packing. In particular, we do not know how to describe the probability measures that satisfy the conditions of Corollary D.

The paper is organized as follows. Some simple facts are collected in § 2. Theorem C is deduced from Theorem A in § 3. In § 4, we prove the sharpened matching lemma that is basic for the proof of Theorem A, and Theorem A is proved in the final § 5.

2. Some basis facts. The *pointwise convergence topology* on the set of all functions on $[0, 1]$ is the coarsest topology such that all the maps $f \to f(t)$ are continuous for all $0 \leqq t \leqq 1$. *Helly's compact* H is the set of all nondecreasing functions from $[0, 1]$ to $[0, 1]$ provided with the pointwise convergence topology [Eng], and $\mathscr{C}$ is a closed subset of Helly's compact. It should be noted that $\mathscr{H}$ is not metrizable. For each probability measure γ on $[0, 1]$, the map $f \to \int f\,d\gamma$ is pointwise continuous on $\mathscr{H}$, and hence on $\mathscr{C}$ (see [Eng], 3.2.E). (This is an elementary exercise.) The following will be essential.

LEMMA 1. *For each probability measure γ on $[0, 1]$, we have*

$$c(\gamma) = \operatorname*{Sup}_{f \in \mathscr{C}} \int f\,d\gamma. \tag{2.1}$$

Proof. Let $\theta < c(\gamma)$. Then, according to Theorem E, b, of [Rhe2], there exists a nonnegative continuous function g on $[0, 1]$ such that $\int g\,d\gamma > \theta$, $g(0) = 0$, and that $\sum_{i \leqq k} g(x_i) \leqq 1$ whenever $\sum_{i \leqq k} x_i \leqq 1$, $0 \leqq x_i \leqq 1$. This shows that if we define

$$f(x) = \operatorname{Sup} \left\{ \sum_{i \leqq k} g(x_i);\ k \geqq 1,\ \sum_{i \leqq k} x_i \leqq x,\ 0 \leqq x_i \leqq 1 \right\}$$

then $f \leqq 1$. It is routine to check that $f \in \mathscr{C}$; and since $f \geqq g$, we have $\int f\,d\gamma \geqq \int g\,d\gamma$. So, we have shown that

$$c(\gamma) \leqq \operatorname*{Sup}_{f \in \mathscr{C}} \int f\,d\gamma.$$

To prove the reverse inequality, we note that if $f \in \mathscr{C}$, then if $0 \leqq x_i \leqq 1$ and $\sum_{i \leqq k} x_i \leqq 1$, we have $\sum_{i \leqq k} f(x_i) \leqq f(\sum_{i \leqq k} x_i) \leqq 1$, so the inequality $\int f\,d\gamma \leqq c(\gamma)$ follows from [Rhe2], Theorem E, c.

Since $f \to \int f\,d\mu$ is pointwise continuous, this Lemma implies that $\mathscr{C}_\mu$ is not empty, and that $c(\gamma) = \max_{f \in \mathscr{C}} \int f\,d\gamma$.

LEMMA 2. *For each n, we can find independent identically distributed random variables $X_1, \cdots, X_n$ distributed according to μ and processes $W_{n,\mu,t}$, $W^-_{n,\mu,t}$ jointly distributed as $W_{\mu,t}$, $W^-_{\mu,t}$ such that*

$$E(S_n) \leqq K(\log n)/\sqrt{n}, \qquad E(S_n^-) \leqq K(\log n)/\sqrt{n} \tag{2.2}$$

where

$$S_n = \operatorname*{Sup}_{0 \leqq t \leqq 1} |(\operatorname{card}\{i \leqq n;\ X_i \leqq t\} - nF_\mu(t))/\sqrt{n} - W_{n,\mu,t}| \tag{2.3}$$

$$S_n^- = \operatorname*{Sup}_{0 \leqq t \leqq 1} |(\operatorname{card}\{i \leqq n;\ X_i < t\} - nF_\mu^-(t))/\sqrt{n} - W^-_{n,\mu,t}|. \tag{2.4}$$

Proof. It is proved in [Kom] that one can find independent identically distributed random variables $Y_1, \cdots, Y_n$ uniformly distributed on $[0, 1]$ and a process $W_{n,t}$ distributed as W_t such that if

$$S_n' = \operatorname*{Sup}_{0 \leqq t \leqq 1} |(\operatorname{card}\{i \leqq n;\ Y_i \leqq t\} - nt)/\sqrt{n} - W_{n,t}|,$$

then $E(S_n') \leqq Kn^{-1/2} \log n$ (and actually $E(\exp(\alpha n^{1/2}(\log n)^{-1} S_n')) < K$ for some $\alpha > 0$). Consider the function H given by $H(u) = \inf\{t \geqq 0;\ \mu([0, t]) \geqq u\}$. Then (as well known), if $X_i = H(Y_i)$, the sequence $(X_i)_{i \leqq n}$ is independent identically distributed and distributed according to μ; and it is easy to see that the result follows by setting $W_{n,\mu,t} = W_{n,F_\mu(t)}$, $W^-_{n,\mu,t} = W^-_{n,F_\mu^-(t)}$ (note that $H(u) \leqq a \Leftrightarrow u \leqq F_\mu(a)$ and $H(u) < a \Leftrightarrow u < F_\mu^-(a)$).

LEMMA 3. (*Integration by parts*; *well known.*) *Let $f \in \mathscr{H}$ and ν be a probability measure on $[0, 1]$, and let $(u_i)_{i \in J}$ be an enumeration of the points t for which $\nu(\{t\}) > 0$. Then*

$$\int f\,d\nu = f^-(1) - \int F_\nu\,d\gamma_f + \sum_{i \in J} (f(u_i) - f^-(u_i))\nu(\{u_i\}).$$

Proof. We have $\int f\,d\nu = \int f^-\,d\nu + \sum_{i \in J} (f(u_i) - f^-(u_i))\nu(\{u_i\})$, since $f = f^-$ except on a set which is at most countable. It remains to show that $\int F^-_{\gamma_f}\,d\nu + \int F_\nu\,d\gamma_f = f^-(1)$. But the first integral is the measure of the set $\{(t, u);\ 0 \leqq t < u \leqq 1\}$ for the product measure $\gamma_f \times \nu$ while the second integral is the measure of $\{(t, u);\ 0 \leqq u \leqq t \leqq 1\}$.

LEMMA 4. *For a continuous function g on $[0, 1]$, and f in $\mathscr{H}$, consider*

$$\phi(f, g) = -\int g \circ F_\mu(t)\,d\gamma_f(t) + \sum_{i \in J} (f(z_i) - f^-(z_i))(g(F_\mu(z_i)) - g(F_\mu^-(z_i))).$$

Then for each g such that $g(0) = g(1) = 0$, the map $f \to \phi(f, g)$ is pointwise continuous on $\mathscr{H}$.

Comment. The purpose of the functional ϕ is to give a rigorous meaning to the expression $\int f\,d(g \circ F_\mu)$.

Proof. We consider first the case where g is piecewise affine. In that case, we can write $g = g_1 - g_2$, where g_1 and g_2 are nondecreasing, $g_1(0) = g_2(0) = 0$, $g_1(1) = g_2(1)$. There is no loss of generality to assume that $g_1(1) = g_2(1) = 1$. In that case, for $l = 1, 2$, $g_l \circ F_\mu$ is right continuous, so it is of the type F_{ν_l} for some probability measure ν_l on $[0, 1]$. Only the points $(z_i)_{i \in J}$ can be atoms of ν_l, and $\nu_l(\{z_i\}) = g_l(F_\mu(z_i)) - g_l(F_\mu^-(z_i))$. Lemma 3 then shows that

$$\phi(f, g_l) = -f^-(1) + \int f \, d\nu_l,$$

so that

$$\phi(f, g) = \int f \, d\nu_1 - \int f \, d\nu_2$$

is pointwise continuous as mentioned above.

We note that for each f in $\mathscr{H}$, we have

$$\sum f(t) - f^-(t) \leqq 1, \tag{2.5}$$

where the summation is taken over all the t for which $f(t) - f^-(t) > 0$. (This fact will be used repeatedly in the sequel.)

It follows easily that for each f in $\mathscr{H}$, we have

$$|\phi(f, g)| \leqq 3 \operatorname{Sup} |g|. \tag{2.6}$$

To conclude the proof, we note that any continuous function which is zero at 0 and 1 can be approximated arbitrarily well in the supremum norm by a piecewise affine function which is zero at 0 and 1.

Remark. If W denotes the function $t \to W_t$, we see from (1.5) that

$$T = \operatorname*{Sup}_{f \in \mathscr{C}_\mu} \phi(f, W).$$

3. Proof of Theorem C. The idea is very simple. Let $\nu = (1/n) \sum_{i \leqq n} \delta_{X_i}$. According to (1.4), and since $n^{-1/2}(n^{5/11}(\log n)^{1/2}) = o(1)$, it is enough to show that $E(|n^{1/2}(c(\nu) - c(\mu)) - T_n|) = o(1)$. The heuristic proof goes as follows. We have, by (2.1)

$$c(\nu) - c(\mu) = \operatorname*{Sup}_{f \in \mathscr{C}} \left(\int f \, d\nu - c(\mu) \right). \tag{3.1}$$

By (2.3), we have

$$F_\nu(t) = F_\mu(t) + n^{-1/2} W_{n, F_\mu(t)} + o(n^{-1/2}), \tag{3.2}$$

so we can hope that

$$d\nu = d\mu + n^{-1/2} \, d(W_{n, F_\mu}) + o(n^{-1/2})$$

and hence

$$\begin{aligned} \int f \, d\nu &= \int f \, d\mu + n^{-1/2} \int f \, d(W_{n, F_\mu}) + o(n^{-1/2}) \\ &= \int f \, d\mu + n^{-1/2} \phi(f, W_{n,\cdot}) + o(n^{-1/2}), \end{aligned}$$

(where $W_{n,\cdot}$ is the function $t \to W_{n,t}$). So we have

$$\begin{aligned} c(\nu) - c(\mu) &= \operatorname*{Sup}_{f \in \mathscr{C}} \left(\int f \, d\mu - c(\mu) + n^{-1/2} \phi(f, W_{n,\cdot}) \right) + o(n^{-1/2}) \\ &= n^{-1/2} \Delta + o(n^{-1/2}), \end{aligned} \tag{3.3}$$

where Δ is the right derivative at zero of the function $a \to \mathrm{Sup}_{f\in\mathscr{C}}(\int f\,d\mu - c(\mu) + a\phi(f, W_{n,\cdot}))$, that turns out to be $\mathrm{Sup}_{f\in\mathscr{C}_\mu}\phi(f, W_{n,\cdot}) = T_n$. This completes the heuristic proof.

We start the rigorous proof, and we first collect a few easy facts.

LEMMA 5. *With probability one, we have for all f in $\mathscr{H}$,*

$$\left|\int f\,d\nu - f^-(1) + \int F_\nu\,d\gamma_f - \sum_{i\in J}(f(z_i)-f^-(z_i))\nu(\{z_i\})\right| \leqq 1/n.$$

Proof. The points $X_1, \cdots, X_n$ are not distinct in general. However, with probability 1, if two points X_i, X_j are equal, they are equal to one z_i. Let $u_1, \cdots, u_m$ $(m \leqq n)$ be the points u for which $\nu(\{u\}) > 0$; that is $u_1, \cdots, u_m$ is the set of different values taken by $X_1, \cdots, X_n$. According to Lemma 3, we have

$$\int f\,d\nu = f^-(1) - \int F_\nu\,d\gamma_f + \sum_{i\leqq m}(f(u_i)-f^-(u_i))\nu(\{u_i\}). \tag{3.4}$$

Now (with probability 1) if u_j is not one of the points z_i, then $\nu(\{u_j\}) \leqq 1/n$. The result then follows from (2.5).

For f in $\mathscr{C}$, we denote by $T_{n,f}$ (respectively, T_f) the random variable given by

$$T_{n,f} = \phi(f, W_{n,\cdot}) = -\int W_{n,\mu,t}\,d\gamma_f(t) + \sum_{i\in J}(f(z_i)-f^-(z_i))(W_{n,\mu,z_i} - W^-_{n,\mu,z_i})$$

$$\left(\text{respectively,}\quad T_f = \phi(f, W_\cdot) = -\int W_{\mu,t}\,d\gamma_f(t) + \sum_{i\in J}(f(z_i)-f^-(z_i))(W_{\mu,z_i} - W^-_{\mu,z_i})\right)$$

so that $T = \mathrm{Sup}_{f\in\mathscr{C}_\mu} T_f$. We set $T_n = \mathrm{Sup}_{f\in\mathscr{C}_\mu} T_{n,f}$, so that T_n is distributed as T.

LEMMA 6. *With probability* 1, *for all f in $\mathscr{H}$, we have*

$$\left|\int f\,d\nu - \int f\,d\mu - n^{-1/2}T_{n,f}\right| \leqq 1/n + (2S_n + S_n^-)n^{-1/2}. \tag{3.5}$$

Proof. From Lemmas 3 and 5, with probability 1 we have for all f in $\mathscr{H}$,

$$\left|\int f\,d\nu - \int f\,d\mu + \int (F_\nu - F_\mu)\,d\gamma_f - \sum_{i\in J}(f(z_i)-f^-(z_i))(\nu(\{z_i\}) - \mu(\{z_i\}))\right| \leqq 1/n.$$

From (2.3), we have, for all $0 \leqq t \leqq 1$,

$$|F_\nu(t) - F_\mu(t) - n^{-1/2}W_{n,\mu,t}| \leqq S_n n^{-1/2}.$$

For all $i \in J$, we have

$$\nu(\{z_i\}) - \mu(\{z_i\}) = F_\nu(z_i) - F_\mu(z_i) - (F^-_\nu(z_i) - F^-_\mu(z_i)),$$

so from (2.3) and (2.4), we have

$$|\nu(\{z_i\}) - \mu(\{z_i\}) - n^{-1/2}(W_{n,\mu,z_i} - W^-_{n,\mu,z_i})| \leqq (S_n + S_n^-)n^{-1/2}.$$

Hence

$$\left|\int f\,d\nu - \int f\,d\mu - n^{-1/2}T_{n,f}\right| \leqq 1/n + S_n n^{-1/2} + \sum_{i\in J}(f(z_i)-f^-(z_i))(S_n + S_n^-)n^{-1/2}.$$

This concludes the proof.

The basic lemma is as follows.

LEMMA 7. *Consider the function*

$$\psi(a) = \operatorname{Sup}_{f \in \mathscr{C}} \left(\int f \, d\mu + a T_f - c(\mu) \right). \tag{3.6}$$

Then $\psi(0) = 0$, ψ is convex, and its right derivative at zero is equal to T.

Proof. It is obvious that $\psi(0) = 0$ (since $c(\mu) = \operatorname{Sup}_{f \in \mathscr{C}} \int f \, d\mu$) and that ψ is convex. Denote by Δ its right derivative at zero, i.e., $\Delta = \lim_{a \to 0, a > 0} \psi(a)/a$. Since ψ is convex, we have $\psi(a) \geqq a\Delta$ for $a > 0$. Obviously, we have $\psi(a) \geqq aT$, so $\Delta \geqq T$. Since $\psi(k^{-1}) \geqq k^{-1}\Delta$ for each k, we can find $f_k \in \mathscr{C}$ such that

$$\int f_k \, d\mu + k^{-1} T_{f_k} - c(\mu) \geqq k^{-1}\Delta - k^{-2}.$$

Since $T_{f_k} = \phi(f_k, W_\cdot)$ satisfies $|T_{f_k}| \leqq 3 \operatorname{Sup}_t |W_t|$ for all k by (2.6), we see that $\liminf_{k \to \infty} \int f_k \, d\mu \geqq c(\mu)$. Hence all cluster points of the sequence (f_k) (for the pointwise topology) belong to $\mathscr{C}_\mu$. Let f be such a cluster point. We have

$$k^{-1}\Delta - k^{-2} \leqq \int f_k \, d\mu + k^{-1} T_{f_k} - c(\mu) \leqq k^{-1} T_{f_k},$$

so $\Delta - k^{-1} \leqq T_{f_k}$ for each k. Since, by Lemma 4, the map $h \to T_h$ is pointwise continuous on $\mathscr{H}$, we see that $\Delta \leqq T_f$, and $T_f \leqq T$ since $f \in \mathscr{C}_\mu$. So $\Delta \leqq T$, and this concludes the proof.

We now prove Theorem C. From Lemma 1, we have $c(\nu) = \operatorname{Sup}_{f \in \mathscr{C}} \int f \, d\nu$, so $c(\nu) - c(\mu) = \operatorname{Sup}_{f \in \mathscr{C}} \int f \, d\nu - c(\mu)$. From (3.5), we have

$$\left| c(\nu) - c(\mu) - \operatorname{Sup}_{f \in \mathscr{C}} \left(\int f \, d\mu + n^{-1/2} T_{n,f} - c(\mu) \right) \right| \leqq (2S_n + S_n^-) n^{-1/2} + 1/n,$$

so if ψ_n denotes the function defined as ψ but using $T_{n,f}$ instead of T_f, we have

$$|n^{1/2}(c(\nu) - c(\mu)) - n^{1/2}\psi_n(n^{-1/2})| \leqq 2S_n + S_n^- + n^{-1/2}.$$

So we have

$$E(|n^{1/2}(c(\nu) - c(\mu)) - n^{1/2}\psi_n(n^{-1/2})|) \leqq K n^{-1/2} \log n$$

for some universal constant K. Now the random variable $n^{1/2}\psi_n(n^{-1/2}) - T_n$ is distributed as $n^{1/2}\psi(n^{-1/2}) - T$. By Lemma 7, we have $T \leqq n^{1/2}\psi(n^{-1/2}) \leqq \psi(1)$, and $\lim_{n \to \infty} n^{1/2}\psi(n^{-1/2}) = T$. Since $E(|\psi(1)|) < \infty$, by dominated convergence we see that $\lim_{n \to \infty} E(|n^{1/2}\psi(n^{-1/2}) - T|) = 0$, so $\lim_{n \to \infty} E(|n^{1/2}\psi_n(n^{-1/2}) - T_n|) = 0$. This concludes the proof.

Proof of Corollary D. From Theorem C, we see that

$$\lim_{n \to \infty} E((Q_n(X_1, \cdots, X_n) - nc(\mu))/\sqrt{n}) = 0$$

if and only if $E(T) = 0$. Now T is the supremum of a collection of centered Gaussian random variables, so it can have zero expectation only if whenever $f_1, f_2 \in \mathscr{C}_\mu$, we have $T_{f_1} = T_{f_2}$ almost surely, i.e. $T_{f_1} - T_{f_2} = 0$ almost surely. It is a well-known fact that for a continuous linear functional Z on $C([0, 1])$, the random variable $Z(W_t)$ is almost surely zero if and only if $Z(g) = 0$ for all g in $C([0, 1])$ for which $g(0) = g(1) = 0$. We have (with the notation of Lemma 4) $T_{f_1} - T_{f_2} = \phi(f_1, W_t) - \phi(f_2, W_t)$. So, the necessary and sufficient condition is that $\phi(f_1, g) = \phi(f_2, g)$ for all g in $C([0, 1])$ such that $g(0) = g(1) = 0$.

FACT. *We have $f_1 = f_2$ on the support A of μ, except on a countable set of μ measure zero if and only if $\int f_1\, d\nu = \int f_2\, d\nu$ for each probability measure ν such that $F_\nu = h \circ F_\mu$ for a continuous nondecreasing function h.*

Proof. It is routine to check that ν has the property that $F_\nu = h \circ F_\mu$ for a continuous nondecreasing function h if and only if $\nu(A) = 1$ and $\nu(\{t\}) = 0$ whenever $\mu(\{t\}) = 0$. This makes the necessity obvious. Conversely, if $\int f_1\, d\nu = \int f_2\, d\nu$ for each such probability ν, then $f_1 = f_2$ at each point t for which $\mu(\{t\}) > 0$, as well as at each point t of A at which f_1 and f_2 are continuous. Since f_1 and f_2 each have at most a countable number of discontinuities, this concludes the proof.

We go back to the proof of Corollary D.

Proof of Necessity. We define the function ι by $\iota(t) = t$. Whenever h is continuous nondecreasing, $h(0) = 0$ and $h(1) = 1$, we must have $\phi(f_1, h - \iota) = \phi(f_2, h - \iota)$. Using Lemma 3, we see that for $i = 1, 2$, we have $\phi(f_i, \iota) = \int f_i\, d\mu - f_i^-(1) = c(\mu) - f_i^-(1)$. Let ν be the probability measure such that $F_\nu = h \circ F_\mu$. Then Lemma 3 shows that $\phi(f_i, h) = \int f_i\, d\nu - f_i^-(1)$ for $i = 1, 2$, so $\phi(f_i, h - \iota) = \int f_i\, d\nu - c(\mu)$, so we must have $\int (f_1 - f_2)\, d\nu = 0$ whenever $F_\nu = h \circ F_\mu$ for a continuous function h such that $h(0) = 0$, $h(1) = 1$. The conclusion then follows from the Fact.

Proof of Sufficiency. The fact shows that $\int f_1\, d\nu = \int f_2\, d\nu$ whenever $F_\nu = h \circ F_\mu$ for some continuous nondecreasing function h with $h(0) = 0$, $h(1) = 1$, and hence that $\phi(f_1, h - \iota) = \phi(f_2, h - \iota)$ by the same computation as above. It follows that for each continuous nondecreasing function h which is zero at the origin, we have $\phi(f_1, h - h(1)\iota) = \phi(f_2, h - h(1)\iota)$. Any g in $C([0, 1])$ with $g(0) = g(1) = 0$ can be approximated in the uniform norm by a difference $h_1 - h_2$, where h_1 and h_2 are nondecreasing and $h_1(0) = h_2(0) = 0$, $h_1(1) = h_2(1)$. Since $h_1 - h_2 = h_1 - h_1(1)\iota - (h_2 - h_2(1)\iota)$, we see that $\phi(f_1, g) = \phi(f_2, g)$ and this completes the proof.

4. Tighter bounds for packing. The proof closely follows that of Theorem B of [Rhe2]. The crucial point of that proof was a matching Lemma (Lemma 4 of [Rhe2]). Our gain in accuracy will be obtained by an improvement of that lemma. While the proof of Lemma 4 of [Rhe2] was purely deterministic, the method we will use here is partly probabilistic. It will make use of the following lemma. Lemma 8 could be sharpened and generalized, but we will give only the simplest version that is suitable for our needs.

LEMMA 8. *Let $m \geqq 16$. Consider independent random variable $(Z_i)_{i \leqq m}$, numbers a, b, and numbers $(a_i)_{i \leqq m}$, $(b_i)_{i \leqq m}$ such that $a \leqq a_i \leqq Z_i \leqq b_i \leqq b$ for each i. Assume that for some $r \geqq \sqrt{\log m}$ and for any t, we have*

$$\operatorname{card}\{i \leqq m;\ a_i \leqq t \leqq b_i,\ a_i \neq b_i\} \leqq r.$$

Then we have

$$P\left(\sup_{a \leqq t \leqq b} \left| \operatorname{card}\{i \leqq m;\ Z_i \geqq t\} - \sum_{i \leqq m} P(Z_i \geqq t) \right| \leqq 1 + 3\sqrt{r \log m} \right) \geqq \frac{3}{4}.$$

Proof. Let $F(t) = \sum_{i \leqq m} P(Z_i \geqq t)$, so F is left continuous and nonincreasing. Let $t_0 = b$ and by induction over $l \geqq 1$, define

$$t_l = \operatorname{Sup}\{t;\ a \leqq t \leqq b,\ F(t) > F(t_{l-1}) + 1\},$$

so that $F(t_l) \geqq F(t_{l-1}) + 1$, and hence $F(t_l) \geqq l$. The construction continues until we construct t_q, for which $F(a) = m \leqq F(t_q) + 1$. Set $t_{q+1} = a$. Since $F(t_q) \leqq m$, we have $q \leqq m$. For $0 \leqq l \leqq q + 1$, $i \leqq m$, define

$$X_{i,l} = P(Z_i \geqq t_l) - 1_{\{Z_i \geqq t_l\}}, \qquad Y_{i,l} = P(Z_i > t_l) - 1_{\{Z_i > t_l\}},$$

and set

$$X_l = \sum_{1 \leqq i \leqq m} X_{i,l}, \qquad Y_l = \sum_{1 \leqq i \leqq m} Y_{i,l}.$$

We note that $E(X_{i,l}) = E(Y_{i,l}) = 0$.

Fix t in $[a, b[$ and let l be the smallest integer $\leqq q+1$ such that $t_l \leqq t$, so $t < t_{l-1}$. Assume that $t_l < t$. By definition of t_l, we have $F(u) \leqq F(t_{l-1}) + 1$ for $u > t_l$. Also $F(t_{l-1}) \leqq F(t)$. We have

$$\sum_{i \leqq m} P(Z_i > t_l) = \lim_{u \to t_l, u > t_l} F(u) \leqq F(t_{l-1}) + 1 \leqq F(t) + 1,$$

and hence

$$-F(t) + \operatorname{card}\{i \leqq m;\ Z_i \geqq t\} \leqq -\sum_{i \leqq m} P(Z_i > t_l) + \operatorname{card}\{i \leqq m;\ Z_i > t_l\} + 1 = 1 - Y_l.$$

We also have

$$F(t) - \operatorname{card}\{i \leqq m;\ Z_i \geqq t\} \leqq F(t_{l-1}) - \operatorname{card}\{i \leqq m;\ Z_i \geqq t_{l-1}\} + 1 = 1 + X_{l-1}.$$

This shows that

$$\operatorname*{Sup}_{t \in [a,b]} |F(t) - \operatorname{card}\{i \leqq m;\ Z_i \geqq t\}| \leqq 1 + \operatorname*{Sup}_{0 \leqq l \leqq q+1} (|X_l|, |Y_l|).$$

For any given l, by hypothesis at most r of the variables $X_{l,i}$ are not constant. Also $-1 \leqq X_{l,i} \leqq 1$ and $\operatorname{Var}(X_{l,i}) \leqq 1$. It then follows from Bernstein's inequality, as in [Hoe], that for $u > 0$,

$$P\left(\left|\sum_{i \leqq m} X_{l,i}\right| > u\right) \leqq 2 \exp(-u^2/(2r + 2u/3)).$$

Taking $u = 3\sqrt{r \log m}$, we have $u \leqq 3r$, so

$$P\left(\left|\sum_{i \leqq m} X_{l,i}\right| > 3\sqrt{r \log m}\right) \leqq 2 \exp(-(\tfrac{9}{4}) \log m) = 2m^{-9/4}$$

and a similar inequality holds for $Y_{l,i}$. So for $m \geqq 16$,

$$P(\text{for each } l, |X_l| \leqq 3\sqrt{r \log m}, |Y_l| \leqq 3\sqrt{r \log m}) \geqq 1 - 2(m+2)m^{-9/4} \geqq 1 - 4/m \geqq \tfrac{3}{4}.$$

The following lemma is purely technical. Its proof is entirely similar to that of Lemma 3 of [Rhe2].

LEMMA 9. *Let γ be a positive measure on a compact metric space S. Consider $p \geqq 1$. Consider a sequence $(b_i)_{i \leqq p}$ such that $\sum_{1 \leqq i \leqq p} b_i = \|\gamma\|$, and a continuous function f on S. Then there is $u_p \leqq u_{p-1} \leqq \cdots \leqq u_0$ and for $1 \leqq i \leqq p$ there is a positive measure γ_i on S such that the following hold*

$$\gamma = \sum_{i \leqq p} \gamma_i \tag{4.1}$$

$$\gamma_i \text{ is supported by } f^{-1}([u_i, u_{i-1}]) \tag{4.2}$$

$$\|\gamma_i\| = b_i. \tag{4.3}$$

LEMMA 10. *Let $16 \leqq m \leqq n$. Let γ be a positive measure on $[0,1]^2$. Assume that $(m/n) \leqq \|\gamma\| < (m+1)/n$. Consider the three positive measures given by, for each Borel set U,*

$$\gamma'(U) = \gamma(U \times [0,1])$$
$$\gamma''(U) = \gamma([0,1] \times U)$$
$$\eta(U) = \gamma(\{(x,y);\ x + y \in U\}).$$

(In other words, γ' and γ'' are the marginals of γ, and η is the distribution of $x+y$, so η is supported by $[0,2]$.)

Consider two sequences $(l_i)_{i\le m}$, $(y_i)_{i\le m}$ of numbers, $0\le l_i,\ y_i\le 1$. Assume that

$$\text{for each } t\in[0,1],\quad \text{card}\{i\le m;\ l_i\ge t\}\le n\gamma'([t,1]), \tag{4.4}$$

$$\text{for each } t\in[0,1],\quad \text{card}\{i\le m;\ y_i\ge t\}\le n\gamma''([t,1]). \tag{4.5}$$

Let $1\le k\le m^{1/4}$. There exist two subsets A and B of $\{1,\cdots,m\}$ such that $B\subset A$, and that

$$\text{card } A\ge m-K\sqrt{\log m}\,(m/k)^{1/3}, \tag{4.6}$$

$$\text{card } B\ge m-K\sqrt{\log m}\,(mk^2)^{1/3}, \tag{4.7}$$

and there exists a one-to-one map ϕ from B to $\{1,\cdots,m\}$ such that if we set $z_i=l_i$ for $i\in A\setminus B$ and $z_i=l_i+y_{\phi(i)}$ for $i\in B$, then

$$\text{for each } t\in[0,2],\quad \text{card}\{i\in A;\ z_i\ge t\}\le n\eta([t,2]). \tag{4.8}$$

Remark. In the case $k=\lfloor m^{1/4}\rfloor$, this means that there is a subset B of $\{1,\cdots,m\}$ with card $B\ge m-Km^{1/4}\sqrt{\log m}$ and a one-to-one map ϕ from B to $\{1,\cdots,m\}$ such that

$$\text{for each } t\in[0,2],\ \text{card}\{i\in B;\ l_i+y_{\phi(i)}\ge t\}\le n\eta([t,2]). \tag{4.9}$$

Proof of Lemma 10. Let $q=\lfloor m^{1/3}k^{-4/3}\rfloor$, $p=\lfloor\sqrt{m/q}\rfloor$. We note that $q\le m^{1/3}$, so $p\ge\lfloor m^{1/3}\rfloor\ge q$ and $p\ge 2$. We note that $(p+1)^2q\ge m$, so $(p+3)pq\ge m+1$. Also $p^2q\le m\le n\|\gamma\|$. Hence, if $p'=\lceil n\|\gamma\|/pq\rceil$, we have $q\le p\le p'\le p+3$.

Step 1. Using Lemma 9, we can find numbers $u_{p'}\le\cdots\le u_0$ and for $1\le i\le p'$ a positive measure γ_i on $[0,1]^2$ that is supported by the set

$$\{(x,y)\in[0,1]^2;\ u_i\le x+y\le u_{i-1}\},$$

such that $\gamma=\sum_{1\le i\le p'}\gamma_i$, and that

$$\|\gamma_i\|=pq/n\quad\text{for } 1\le i<p',$$

$$\|\gamma_{p'}\|=(n\|\gamma\|-(p'-1)pq)/n\le pq/n.$$

Step 2. For each $1\le i\le p'-1$, let $q'(i)=q$. Define $q'(p')$ as the smallest integer $\ge\|\gamma_{p'}\|n/p$, so $q'(p')\le q$. Using Lemma 9 for each $1\le i\le p'$, we can find numbers $u_{i,q'(i)}\le\cdots\le u_{i,0}$, and for $1\le i\le q'(i)$ a positive measure $\gamma_{i,j}$ on $[0,1]^2$, that is supported by the set

$$\{(x,y)\in[0,1]^2;\ u_i\le x+y\le u_{i-1},\ u_{i,j}\le y\le u_{i,j-1}\},$$

such that $\gamma_i=\sum_{1\le j\le q(i)}\gamma_{i,j}$ and that

$$\|\gamma_{i,j}\|=p/n\quad\text{for } i<p'\ \text{ or }\ j<q'(p)$$

and

$$\|\gamma_{p',q'(p')}\|\le p/n.$$

Step 3. For each $1\le i\le p'$, and each $1\le j\le q'(i)$, we denote by $r(i,j)$ the smallest integer $\ge n\|\gamma_{i,j}\|$, so $r(i,j)=p$ unless $i=p'$, $j=q'(p')$, in which case $r(p',q'(p'))\le p$. Using Lemma 9 for each $1\le i\le p'$, $1\le j\le q'(i)$, we can find numbers $u_{i,j,r(i,j)}\le\cdots\le u_{i,j,0}$ and for $1\le l\le r(i,j)$ a positive measure $\gamma_{i,j,l}$ on $[0,1]^2$, that is supported by the set

$$A_{i,j,l}=\{(x,y)\in[0,1]^2;\ u_i\le x+y\le u_{i-1},\ u_{i,j}\le y\le u_{i,j-1}$$
$$u_{i,j,l}\le x\le u_{i,j,l-1}\}$$

and such that $\gamma_{i,j} = \sum_{1 \leqq l \leqq r(i,j)} \gamma_{i,j,l}$ and $\|\gamma_{i,j,l}\| = 1/n$ except for $i = p'$, $j = q'(p')$, $l = r(p', q'(p'))$ in which case $\|\gamma_{i,j,l}\| \leqq 1/n$.

For the sake of simplicity, we set

$$S = \{(i, j, l);\ 1 \leqq i \leqq p';\ 1 \leqq j \leqq q'(i);$$

$$1 \leqq l \leqq r(i, j);\ (i, j, l) \neq (p', q'(p'), r(p', q'(p')))\}$$

so that card $S = m$ and $\|\gamma_{i,j,l}\| = 1/n$ for (i, j, l) in S.

Step 4. We prove the following assertions: for all t,

$$\text{card}\{s \in S;\ \text{there exists } (x, y) \in A_s,\ x + y = t,\ \text{there exists } (x', y') \in A_s,\ x' + y' \neq t\} \leqq 2pq, \tag{4.10}$$

$$\text{card}\{s \in S;\ \text{there exists } (x, y) \in A_s,\ y = t,\ \text{there exists } (x', y') \in A_s,\ y' \neq t\} \leqq 2p'p, \tag{4.11}$$

$$\text{card}\{s \in S;\ \text{there exists } (x, y) \in A_s,\ x = t,\ \text{there exists } (x', y') \in A_s,\ x' \neq t\} \leqq 2p'q. \tag{4.12}$$

To prove (4.10), we fix t and we denote by i' the smallest index for which $u_{i'} < t$, and by i'' the largest index for which $t < u_{i''-1}$. (The case where one of these indices cannot be defined is simpler and is left to the reader.) If, for some $s = (i, j, l) \in S$, there exists $(x, y) \in A_s$ with $x + y = t$, then necessarily $i'' \leqq i \leqq i'$. If we have $i'' < i < i'$, then, by definition of i' and i'', we have $u_i \geqq t$, $u_{i-1} \leqq t$. Since $u_{i-1} \geqq u_i$, we have $t = u_i = u_{i-1}$, and there exists no (x', y') in A_s with $x' + y' \neq t$. So, we have shown that if there is (x, y) in A_s with $x + y = t$, and there is (x', y') in A_s with $x' + y' \neq t$, then $i = i'$ or $i = i''$. So there are only two possibilities for i and at most p possibilities for j, q possibilities for l, which proves (4.10).

To prove (4.11) (respectively, (4.12)), we fix t and we prove in a similar way, that for any $1 \leqq i \leqq p'$ (respectively $1 \leqq i \leqq p'$, $1 \leqq j \leqq q'(i)$) there are at most two possibilities for j (respectively, l).

Step 5. For each s in S, we consider a random variable (U_s, V_s) distributed according to $n\gamma_s$. We set

$$F'(t) = \sum_{s \in S} P(U_s \geqq t), \quad F''(t) = \sum_{s \in S} P(V_s \geqq t), \quad F(t) = \sum_{s \in S} P(U_s + V_s \geqq t).$$

Since $\|\gamma - \sum_{s \in S} \gamma_s\| \leqq 1/n$, we have

$$\sup_{0 \leqq t \leqq 1} |F'(t) - n\gamma'([t, 1])| \leqq 1, \qquad \sup_{0 \leqq t \leqq 1} |F''(t) - n\gamma''([t, 1])| \leqq 1,$$

$$\sup_{0 \leqq t \leqq 2} |F(t) - n\gamma([t, 2])| \leqq 1.$$

It is easily seen that, since $m \geqq 16$, we have $pq \geqq \sqrt{\log m}$, so that $p'p \geqq p^2 \geqq pq \geqq \sqrt{\log m}$. We can then use Step 4 and Lemma 8 to see that

$$P\left(\sup_{0 \leqq t \leqq 1} |\text{card}\{s \in S;\ U_s \geqq t\} - n\gamma'([t, 1])| \leqq 2 + 3\sqrt{2p'q \log m}\right) \geqq \frac{3}{4},$$

$$P\left(\sup_{0 \leqq t \leqq 1} |\text{card}\{s \in S;\ V_s \geqq t\} - n\gamma''([t, 1])| \leqq 2 + 3\sqrt{2p'p \log m}\right) \geqq \frac{3}{4},$$

$$P\left(\sup_{0 \leqq t \leqq 2} |\text{card}\{s \in S;\ U_s + V_s \geqq t\} - n\eta([t, 2])| \leqq 2 + 3\sqrt{2pq \log m}\right) \geqq \frac{3}{4}.$$

So, we can find points $(u_s, v_s)_{s\in S}$ such that for $0\leqq t\leqq 1$,

$$\text{card}\,\{s\in S;\ u_s\geqq t\}\geqq n\gamma'([t,1])-2-3\sqrt{2p'q\log m}\,,$$

$$\text{card}\,\{s\in S;\ v_s\geqq t\}\geqq n\gamma''([t,1])-2-3\sqrt{2p'p\log m}\,,$$

and for $0\leqq t\leqq 2$,

$$\text{card}\,\{s\in S;\ u_s+v_s\geqq t\}\leqq n\eta([t,2])+2+3\sqrt{2pq\log m}\,.$$

Step 6. It follows from [Rhe2], Lemma 2 (in which we take $p=1$) that there is $S'\subset S$ with $\text{card}\,(S\backslash S')\leqq 3+3\sqrt{2pq\log m}$, and such that for $0\leqq t\leqq 2$,

$$\text{card}\,\{s\in S';\ u_s+v_s\geqq t\}\leqq n\eta([t,2]). \tag{4.13}$$

So we have, for $0\leqq t\leqq 1$,

$$\text{card}\,\{s\in S';\ u_s\geqq t\}\geqq n\gamma'([t,1])-5-6\sqrt{2p'q\log m}\,,$$

$$\text{card}\,\{s\in S';\ v_s\geqq t\}\geqq n\gamma''([t,1])-5-6\sqrt{2p'p\log m}\,.$$

In the last inequality, we have used the fact that $q\leqq p\leqq p'$. It then follows from (4.4) and (4.5) that for $0\leqq t\leqq 1$,

$$\text{card}\,\{i\leqq m;\ l_i\geqq t\}\leqq \text{card}\,\{s\in S';\ u_s\geqq t\}+5+6\sqrt{2p'q\log m}\,,$$

$$\text{card}\,\{i\leqq m;\ y_i\geqq t\}\leqq \text{card}\,\{s\in S';\ v_s\geqq t\}+5+6\sqrt{2p'p\log m}\,.$$

It follows from [Rhe2] Lemma 1, that there is a subset A of $\{1,\cdots,m\}$ such that

$$\text{card}\,A\geqq m-5-6\sqrt{2p'q\log m}\,, \tag{4.14}$$

and a one-to-one map ϕ_1 from A to S' such that $l_i\leqq u_{\phi_1(i)}$ for i in A. The same lemma implies that there is a subset C of $\{1,\cdots,m\}$ such that

$$\text{card}\,C\geqq m-5-6\sqrt{2p'p\log m}\,,$$

and a one-to-one map ϕ_2 from C to S' such that $y_i\leqq v_{\phi_2(i)}$ if $i\in C$. We consider the subset B of A where $\phi=\phi_2^{-1}\circ\phi_1$ is defined; so

$$\text{card}\,B\geqq m-10-12\sqrt{2p'p\log m}\,. \tag{4.15}$$

For i in A, we have $l_i\leqq u_{\phi_1(i)}\leqq u_{\phi_1(i)}+v_{\phi_1(i)}$. For i in B, we have

$$l_i\leqq u_{\phi_1(i)},\ y_{\phi(i)}\leqq v_{\phi_2\phi(i)}=v_{\phi_1(i)},\quad \text{so } l_i+y_{\phi(i)}\leqq u_{\phi_1(i)}+v_{\phi_1(i)}.$$

This and (4.13) easily imply (4.8) while (4.14) and (4.15) imply (4.6) and (4.7).

5. Proof of Theorem A. For $l\geqq 1$, consider the positive measure λ_l on $[0,1]$ given by

$$\lambda_l=\sum_{k\geqq l}(\alpha_k/k)\int_{R_k}\delta_{x_{k-l+1}}\,d\nu_k(x_1,\cdots,x_k),$$

so that $\|\lambda_l\|=\sum_{k\geqq l}\alpha_k/k=\beta_l$. After normalization, λ_l is the distribution of the size of the lth largest item in bins that contain at least l items.

We now fix q and we prove Theorem A. Consider the positive measure

$$\mu'=\sum_{k\geqq q+1}(\alpha_k/k)\int_{R_k}\sum_{1\leqq i\leqq k-q}\delta_{x_i}\,d\nu_k(x_1,\cdots,x_k), \tag{5.1}$$

so, since $x_i\leqq x_{k-q}\leqq 1/(q+1)$, μ' is supported by $[0,1/(q+1)]$. After normalization, μ' is the distribution of the sizes of the items which are not among the q largest in

the bin that contains them. We note that $\mu = \sum_{l \leqq q} \lambda_l + \mu'$. For $1 \leqq l \leqq q$, let $f_l(t) = \lambda_l([t, 1])$ for $0 \leqq t \leqq 1$. Let

$$f_{q+1}(t) = \mu'([t, 1]), \quad \text{so} \sum_{l \leqq q+1} f_l(t) = \mu([t, 1]).$$

Consider the sequence $s_1, \cdots, s_n$ of items. For each t,

$$\text{card}\,\{i \leqq n;\ s_i \geqq t\} \leqq n \sum_{l \leqq q+1} f_l(t) + D.$$

We now use Lemma 2 of [Rhe2] to find disjoint subsets $(C_l)_{l \leqq q+1}$ of $\{1, 2, \cdots, n\}$ with $\sum_{i \leqq q+1} \text{card}\, C_i \geqq n - q - D - 1$ such that for each $l \leqq q+1$, we have

$$\text{card}\,\{i \in C_l;\ s_i \geqq t\} \leqq n f_l(t)$$

for all $0 \leqq t \leqq 1$.

Let $C' = \{1, \cdots, n\} \backslash \cup_{l \leqq q+1} C_l$, so card $C' \leqq q + D + 1$, and hence C' can be packed in $q + D + 1$ bins.

We now pack the families $(C_l)_{l \leqq q}$. This is done exactly as in [Rhe2], except that now at step l, we can use the improved matching Lemma 10 (with $m = \lfloor n\beta_l \rfloor$, $k = \text{Min}\,(\lfloor n\beta_l \rfloor^{1/4}, l)$) instead of the Lemma 4 of [Rhe2]. This replaces terms $(n/\beta_l)^{1/2}$ by $K(\log m)^{1/2}\,\text{Max}\,((n\beta_l/l)^{1/3}, (n\beta_l)^{1/4})$. Denote by ξ_q the positive measure given by

$$\xi_q = \sum_{k \geqq q+1} (\alpha_k/k) \int_{R_k} \delta_{u_k(x)}\, d\nu_k(x_1, \cdots, x_k), \tag{5.2}$$

where $u_k(x) = \sum_{k-q+1 \leqq i \leqq k} x_i$. After normalization, ξ_q is the distribution of the sum of the sizes of the q largest item in bins that contain at least $q+1$ items. We have $\|\xi_q\| = \beta_{q+1}$. Let $n_{n+1} = \lfloor n\beta_{q+1} \rfloor$. The proof of [Rhe2] shows that the families $C_1, \cdots, C_q$ can be packed in two sets of bins the first of which ("dead bins") have at most

$$n \sum_{l \leqq q} \alpha_l/l + 2q + K(\log n)^{1/2} \sum_{l \leqq q} \text{Max}\,((n\beta_l/l)^{1/3}, (n\beta_l)^{1/4})$$

elements. The second set of bins ("live bins") contains n_{q+1} bins $B_1, \cdots, B_{n_{q+1}}$. If we denote by L_i the sum of the sizes of the items packed in bin B_i, we have the following relation:

$$\text{for all } t \text{ in } [0, 1],\ \text{card}\,\{i \leqq n_{q+1};\ L_i \geqq t\} \leqq n\xi_q([t, 1]).$$

We have

$$\begin{aligned} \sum_{i \leqq n_{q+1}} L_i &= \int_0^1 \text{card}\,\{i \leqq n_{q+1};\ L_i \geqq t\}\, dt \\ &\leqq n \int_0^1 \xi_q([t, 1])\, dt = n \int x\, d\xi_q(x) \\ &= n \sum_{k \geqq q+1} (\alpha_k/k) \int_{R_k} \sum_{k-q+1 \leqq i \leqq k} x_i\, d\nu_k(x_1, \cdots, x_k), \end{aligned} \tag{5.3}$$

where the last equality follows from (5.2). Since $n_{q+1} \geqq n\beta_{q+1} - 1$, we see that the available empty space S in the bins B_i, $i \leqq n_{q+1}$, satisfies

$$\begin{aligned} S &= n_{q+1} - \sum_{i \leqq n_{q+1}} L_i \\ &\geqq -1 + n \sum_{k \geqq q+1} (\alpha_k/k) \int_{R_k} \left(1 - \sum_{k-q+1 \leqq i \leqq k} x_i\right) d\nu_k(x_1, \cdots, x_k) \\ &\geqq -1 + n \sum_{k \geqq q+1} (\alpha_k/k) \int_{R_k} \sum_{1 \leqq i < k-q} x_i\, d\nu_k(x_1, \cdots, x_k) \end{aligned}$$

since $\sum_{1\le i\le k} x_i \le 1$ for $(x_1, \cdots, x_k) \in R_k$. The sum $\sum$ of the elements of C_{q+1} is equal to

$$\int_0^1 \text{card}\,\{i \le C_{q+1};\ s_i \ge t\}\,dt \le \int_0^1 n\mu'([t,1])\,dt = n\int_0^1 x\,d\mu'(x)$$

$$= n\sum_{k\ge q+1}(\alpha_k/k)\int_{R_k}\sum_{1\le i<k-q} x_i\,d\nu_k(x_1,\cdots,x_k).$$

This shows that $\sum \le S+1$. We now pack as many as possible of the elements of C_{q+1} in the available space in the bins $(B_i)_{i\le n_{q+1}}$. Since each element of C_{q+1} has size $\le 1/(q+1)$, if we cannot pack all of C_{q+1}, there is at most $1/(q+1)$ wasted space in each bin B_i, so a total amount of wasted space at most $n_{q+1}/(q+1) \le n\beta_{q+1}/(q+1) \le n/(q+1)^2$. Since $\sum \le S+1$, the sum of the sizes of the items of C_{q+1}, which we have not packed, is at most $n/(q+1)^2+1$, and then, according to [Rhe2], Lemma 5, they can be packed in at most $1+(1+n/(q+1)^2)/(1-1/(q+1)) \le 3+n/q^2$ bins. Thus, we have succeeded in packing the families $(C_l)_{l\le q+1}$ in at most

$$3+n/q^2+2q+n\sum_{l\ge 1}\alpha_l/l+K(\log n)^{1/2}\sum_{l\le q}\text{Max}\,((n\beta_l/l)^{1/3},(n\beta_l)^{1/4})$$

for some universal constant K. This proves (1.2). To prove (1.3), we note that by Hölder's inequality,

$$\sum_{l\le q}(n\beta_l)^{1/4} \le n^{1/4}q^{3/4}\left(\sum_{l\le q}\beta_l\right)^{1/4} \le n^{1/4}q^{3/4}$$

and

$$\sum_{l\le q}(n\beta_l/l)^{1/3} \le n^{1/3}\left(\sum_{l\le q}\beta_l\right)^{1/3}\left(\sum_{1\le l\le q}1/l^{1/2}\right)^{2/3} \le Kn^{1/3}q^{1/3}.$$

Taking $q = \lfloor n^{3/11}\rfloor$ gives (1.3). This completes the proof.

REFERENCES

[Bre] L. BREIMAN (1968), *Probability*, Addison-Wesley Publishing Co.

[Cof1] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON (1984), *Approximation algorithms for bin-packing—An updated survey*, in Algorithm Design for Computing System Design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, New York, pp. 49-106.

[Cof2] E. G. COFFMAN, JR., G. S. LUEKER, AND A. H. G. RINNOOY KAN (1988), *Asymptotic methods in the probabilistic analysis of sequencing and packing heuristics*, Management Sci., 34, pp. 266-290.

[Coh] D. L. COHN (1980), *Measure Theory*, Birkhäuser-Verlag, Boston.

[Eng] R. ENGELKING (1977), *General Topology*, Polish Scientific Publishers Monografie Matematyczne, Warsaw.

[Hoe] W. HOEFFDING (1963), *Probability inequalities for sums of bounded random variables*, J. Amer. Statist. Assoc., 58, pp. 13-30.

[Kar] R. M. KARP (1972), *Reducibility among Combinatorial Problems*, in Complexity of Computers Computations, R. E. Miller and J. W. Thatcher eds., Plenum Press, New York, pp. 85-103.

[Kin] J. F. C. KINGMAN (1976), *Sub-additive processes*, Lecture Notes in Mathematics, Springer-Verlag, Berlin, New York, pp. 168-222.

[Kom] J. KOMLÓS, P. MAJOR, AND G. TUSNÁDY (1975), *An approximation of partial sums of independent random variables and the sample* DFI, Z. Wahrsch. Verw. Gebiete, 32, pp. 111-131.

[Lue1] G. S. LUEKER (1982), *An average-case analysis of bin packing with uniformly distributed item sizes*, Report No. 181, Department of Information and Computer Science, University of California, Irvine, CA.

[Lue2] G. S. LUEKER (1983), *Bin packing with items uniformly distributed over intervals* [a, b], Proc. 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, pp. 289-297.

[Rhe1] W. T. RHEE (1988), *Optimal bin packing with items of random sizes*, Math. Oper. Res., 13, pp. 140-151.

[Rhe2] W. T. RHEE AND M. TALAGRAND (1989), *Optimal Bin Packing with Items of Random Sizes*—II, SIAM J. Comput., 18, pp. 139-151.

[Rhe3] W. T. RHEE AND M. TALAGRAND (1987), *Martingale inequalities and* NP-*complete problems*, Math. Oper. Res., 12, pp. 177-181.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 487-498, June 1989

005

OPTIMAL BIN COVERING WITH ITEMS OF RANDOM SIZE*

WANSOO T. RHEE† AND MICHEL TALAGRAND‡

Abstract. Consider a probability measure μ on $[0, 1]$ and independent random variables $X_1, \cdots, X_n$ distributed according to μ. Let $Q_n = Q_n(X_1, \cdots, X_n)$ be the largest number of unit-size bins that can be covered by items of size $X_1, \cdots, X_n$. The properties of Q_n are investigated in the spirit of our work on optimal bin packing with items of random sizes. While the covering problem has many similarities with the packing problem, it turns out to be significantly harder.

Key words. stochastic bin packing, size distribution, compactness, weak convergence

AMS(MOS) subject classifications. 90B99, 60K30

1. Introduction and results. The bin-packing problem requires finding the minimum number of unit-size bins needed to pack a given collection of items with sizes $x_1, \cdots, x_n$, subject to the constraint that the sum of the sizes of the items placed in any given bin must not exceed one. This problem has many applications and has been shown to be *NP*-complete. The *bin-covering problem* requires finding the maximum number $Q_n(x_1, \cdots, x_n)$ of unit-size bins that can be covered by a given collection of items with sizes $x_1, \cdots, x_n$, subject to the constraint that the sum of the sizes of the items attributed to any given bin must be greater than or equal to one. This paper is mostly concerned with the bin-covering problem. This problem is closely related to the bin-packing problem.

The bin-covering problem is *NP*-complete. The purpose of this paper is to make a stochastic analysis of the problem. In our model, we consider a probability measure μ on $[0, 1]$. (No regularity assumption is made on μ.) Consider n items with sizes $X_1, \cdots, X_n$, which are independent identically distributed (i.i.d.) according to μ. (For simplicity, we denote by X_k both item names and item sizes.) It is transparent that $Q_n = Q_n(X_1, \cdots, X_n)$ is a superadditive process, and hence $\lim_{n\to\infty} Q_n/n = d(\mu)$ almost surely (a.s.) for a constant $d(\mu)$, depending only on μ. Also $E(Q_n)/n \leqq d(\mu)$. Among the main results of this paper are the description of $d(\mu)$ in function of the structure of μ and (under mild conditions on μ) the convergence in distribution of the process $n^{-1/2}(Q_n - nd(\mu))$. It should also be noted that one of our most difficult results is purely deterministic (Theorem 2). Our study is in the spirit of our previous work on stochastic bin packing ([3], [4], [5]). Actually, at first glance, our present results look like rather automatic modifications of our bin-packing results (by just "inversing inequalities"). It is true indeed that the bin-packing and bin-covering problems have many similarities, and we do not dwell in great detail on the arguments that are mere transpositions of arguments used for the bin-packing problem (and that are contained in § 2). The whole point of the present paper is that a crucial deterministic estimate (3) is not as sharp as for the bin-packing problem (the reason for this is explained in the comment near the end of the paper). This makes the proof of the convergence in distribution of $n^{-1/2}(Q_n - nd(\mu))$ considerably harder than in the bin-packing case (§ 3). We now explain our basic notations and describe our results.

* Received by the editors October 19, 1987; accepted for publication June 22, 1988. This research is supported in part by National Science Foundation grant DCR-861025.

† Ohio State University, Faculty of Management Science, 1775 College Road, Columbus, Ohio 43210.

‡ Université de Paris VI, Equipe d'Analyse—Tour 46, 4 Place Jussieu, 75230 Paris Cedex 05, France.

For $k \geqq 1$, we set

$$S_k = \left\{ (x_1, \cdots, x_k) \in \mathbf{R}^k; 0 \leqq x_k \leqq \cdots \leqq x_1 \leqq 1, 1 \leqq \sum_{i \leqq k} x_i \leqq 3 \right\}$$

so S_k is compact metric. For a compact space S, we denote by $M_1(S)$ the set of probability measures on S. For $x \in S$, we denote by δ_x the probability measure concentrated at x, that is, for a Borel set G, we have $\delta_x(G) = 1$ if x belongs to G, and $\delta_x(G) = 0$ otherwise. To each $\nu \in M_1(S_k)$, we associate $\mathscr{P}_k(\nu) \in M_1([0, 1])$, such that for each Borel set G of $[0, 1]$, we have

$$\mathscr{P}_k(\nu)(G) = \frac{1}{k} \int_{S_k} \sum_{i \leqq k} \delta_{x_i}(G)\, d\nu(x_1, \cdots, x_k).$$

In short, we will write this formula as

$$\mathscr{P}_k(\nu) = \frac{1}{k} \int_{S_k} \sum_{i \leqq k} \delta_{x_i}\, d\nu(x_1, \cdots, x_k)$$

and we will use a similar convention for similar expressions.

THEOREM 1. *For $\mu \in M_1([0, 1])$, there exists a nonnegative sequence $(\alpha_k)_{k \geqq 0}$ with $\sum_{k \geqq 0} \alpha_k = 1$ and for each $k \geqq 1$, a $\nu_k \in M_1(S_k)$ such that*

$$\mu = \alpha_0 \delta_0 + \sum_{k \geqq 1} \alpha_k \mathscr{P}_k(\nu_k), \tag{1}$$

and that $d(\mu) = \sum_{k \geqq 1} \alpha_k / k$.

In the converse direction, we have the following theorem.

THEOREM 2. *Consider a nonnegative sequence $(\alpha_k)_{k \geqq 0}$ such that $\sum_{k \geqq 0} \alpha_k = 1$, and a sequence ν_k in $M_1(R_k)$. Consider the measure μ given by (1). For $k \geqq 1$, let $\beta_k = \sum_{l \geqq k} \alpha_l / l$. Consider a sequence $s_1, \cdots, s_n$ of items. Let*

$$D = D(s_1, \cdots, s_n) = \sup_{0 \leqq t \leqq 1} n\mu([t, 1]) - \operatorname{card}\{i \leqq n; s_i \geqq t\}.$$

Then, for each $q \geqq 1$, we have

$$Q_n(s_1, \cdots, s_n) \geqq n \sum_{k \geqq 1} \alpha_k / k - (D + 3q + n\beta_q + Kn^{1/3} \sum_{1 \leqq i < q} (\beta_i)^{1/3} (\log n)^{1/2}), \tag{2}$$

where K is a universal constant (independent of μ, n, $s_1, \cdots, s_n$). In particular, we have

$$Q_n(s_1, \cdots, s_n) \geqq n \sum_{k \geqq 1} \alpha_k / k - Kn^{3/5} (\log n)^{3/10} - D. \tag{3}$$

It is to be noted that the error term $n^{3/5}(\log n)^{3/10}$ that we obtain here is of a larger order than the error term $n^{1/3}(\log n)^{1/2}$ that we obtained for the corresponding result in the bin-packing probem ([4]). While we do not know what the best possible error term in (3) is, either for the bin-packing or the bin-covering problems, there are reasons to believe that the bin-covering problem does require a bigger error term. These reasons will be explained in the course of the proof of Theorem 2.

COROLLARY 3. *Under the assumptions of Theorem 2, assume moreover that $\beta_k = o(k^{-3/2}(\log k)^{-3/4})$. Then we have*

$$Q_n(s_1, \cdots, s_n) \geqq n \sum_{k \geqq 1} \alpha_k / k - D - n^{1/2} \varepsilon_n, \tag{4}$$

where ε_n is a quantity depending only on μ and n and that satisfies $\lim_{n \to \infty} \varepsilon_n = 0$.

COROLLARY 4. *Let $s_1, \cdots, s_n \in [0, 1]$. Let $\nu = 1/n \sum_{i \leqq n} \delta_{s_i}$. Then*

$$nd(\nu) \geqq Q_n(s_1, \cdots, s_n) \geqq nd(\nu) - Kn^{3/5} (\log n)^{3/10}. \tag{5}$$

COROLLARY 5. *For each distribution* μ, *we have*

$$nd(\mu) \geqq EQ_n(X_1, \cdots, X_n) \geqq nd(\mu) - Kn^{3/5}(\log n)^{3/10}.$$

COROLLARY 6. *If* μ *is as in* (1), *then* $d(\mu) \geqq \sum_{k \geqq 1} \alpha_k / k$.

Another corollary deserves special notice. It is clear that we always have $d(\mu) \leqq \int x\, d\mu$ for μ in $M_1([0, 1])$. It is said that a distribution μ allows *perfect packing* when the asymptotic level of bin occupancy in optimal bin-packing of a random sequence of items is 1. (See [6] for a more formal definition.) Let us say that μ allows *perfect covering* if this occurs for optimal *bin-covering* packing, or equivalently when $d(\mu) = \int x\, d\mu$. Then we have the following theorem.

THEOREM 7. *A distribution* μ *on* $[0, 1]$ *allows perfect covering if and only if it allows perfect packing.*

This statement surely supports the fact that the bin-packing problem and the bin-covering problem are closely related. (It should however be noted that, for $\varepsilon > 0$, when packing items of constant size $1 - \varepsilon$ (respectively, $\frac{1}{2} + \varepsilon$), there is a proportion of wasted bin space of ε (respectively, $\frac{1}{2} - \varepsilon$) in bin packing, and of $1 - 2\varepsilon$ (respectively, 2ε) in bin covering.

We denote by $\mathscr{C}$ the class of nondecreasing functions f from $[0, 1]$ to $[0, 1]$ that satisfy $f(1) = 1$ and $f(x + y) \leqq f(x) + f(y)$ whenever $x, y, x + y \in [0, 1]$.

THEOREM 8. *For any distribution* μ *on* $[0, 1]$, *we have*

$$d(\mu) = \inf_{f \in \mathscr{C}} \int f\, d\mu.$$

For a positive measure γ on $[0, 1]$, we set $F_\gamma^-(t) = \gamma([0, t))$, $F_\gamma(t) = \gamma([0, t])$. We denote by B_t the standard Brownian motion, and by $W_t = B_t - tB_1$ the Brownian bridge (see [1]). For the simplicity of notations, we set $W_{\mu,t} = W_{\mu([0,t])}$, $W_{\mu,t}^- = W_{\mu([0,t))}$. To each f in $\mathscr{C}$, we associate f^- given by $f^-(x) = \lim_{y \to x, y < x} f(y)$, so that $f^- \leqq f$. Since f^- is left continuous, there is a unique positive measure γ_f on $[0, 1]$ such that $F_{\gamma_f}^-(t) = f^-(t)$ for each $0 \leqq t \leqq 1$ and γ_f has mass $f^-(1) \leqq 1$. Fixing μ in $M([0, 1])$, we denote by $(z_i)_{i \in J}$ an enumeration of the points z in $[0, 1]$ such that $\mu(\{z\}) > 0$, where J is either empty, finite, or countably infinite.

THEOREM 9. *Assume that we can write* $\mu = \alpha_0 \delta_0 + \sum_{k \geqq 1} \alpha_k \mathscr{P}_k(\nu_k)$, *where* $\alpha_k \geqq 0$, $\sum_{k \geqq 0} a_k = 1$, $\sum_{k \geqq 1} \alpha_k / k = d(\mu)$, $\nu_k \in M_1(S_k)$, *and* $\beta_k = o(k^{-3/2}(\log k)^{-3/4})$, *where* $\beta_k = \sum_{l \geqq k} \alpha_l / l$. *Then the set*

$$\mathscr{C}_\mu = \left\{ f \in \mathscr{C};\ \int f\, d\mu = d(\mu) \right\}$$

is nonempty. Consider the random variable

$$T = \inf_{f \in \mathscr{C}_\mu} \left\{ -\int_0^1 W_{\mu,t}\, d\gamma_f(t) + \sum_{i \in J} (f(z_i) - f^-(z_i))(W_{\mu,z_i} - W_{\mu,z_i}^-) \right\}.$$

Then for each n, one can find a random variable T_n *distributed like T and such that*

$$\lim_{n \to \infty} E|(Q_n(X_1, \cdots, X_n) - nd(\mu))/\sqrt{n} - T_n| = 0.$$

The paper is organized as follows. All the proofs are contained in § 2, with the exception of the proof of Theorems 2 and 9. Theorem 9 is deduced from Theorem 2 in § 3, and Theorem 2 itself is proved in the final § 4.

2. Easiest proofs. We recall that for a compact metric space S, the weak topology on $M_1(S)$ is the topology such that a sequence (μ_n) converges weakly to μ whenever

$\int f d\mu_n$ converges to $\int f d\mu$ for all f in $C(S)$, the space of continuous functions of S. For that topology, $M_1(S)$ is compact metric.

Proof of Theorem 1. The proof is very similar to the proof of Theorem A of [4]. Let $D_0 = \{\delta_0\}$, and for $k \geqq 1$, consider the set D_k of probability measures on $[0, 1]$ that are of the type $\mathcal{P}_k(\nu)$ for $\nu \in M_1(S_k)$. Since the map $\nu \to \mathcal{P}_k(\nu)$ is weakly continuous, D_k is convex and weakly compact. We now observe that, if $\mu \in D_k$, $k \geqq 1$, we have $\int x\, d\mu(x) \leqq 3/k$. Indeed, if $\mu = \mathcal{P}_k(\nu)$, $\nu \in M_1(S_k)$, we have

$$\int x\, d\mu(x) = \frac{1}{k} \int_{S_k} \sum_{i \leqq k} x_i\, d\nu(x_1, \cdots, x_k) \leqq 3/k$$

since $\sum_{i \leqq k} x_i \leqq 3$ for $(x_1, \cdots, x_k) \in S_k$. It follows that if we have a sequence (μ_n) in $D_{i(n)}$, where $i(n) \geqq n$, that converge weakly to μ, we must have $\mu = \delta_0$, since $\int x\, d\mu = \lim_{n \to \infty} \int x\, d\mu_n = 0$. It follows that the sequence $(D_i)_{i \geqq 0}$ satisfies the hypothesis of Proposition 2.2 of [4].

It is a well-known consequence of the law of large numbers that $(1/n) \sum_{i \leqq n} \delta_{X_i} \to \mu$ almost surely in the weak topology. Also, $(1/n) Q_n(X_1, \cdots, X_n) \to d(\mu)$ almost surely, so we can find a sequence (x_i) such that $\mu_n = (1/n) \sum_{i \leqq n} \delta_{x_i}$ converges weakly to μ, and that k_n/n converges to $d(\mu)$, where $k_n = Q_n(x_1, \cdots, x_n)$. (We denote item sizes by lowercase letters to emphasize the fact that they are not random.) For each n, we can cover k_n bins with items of size $x_1, \cdots, x_n$. The jth bin, $j \leqq k_n$, is covered by p_j items of size $y_{j,1}, \cdots, y_{j,p_j}$, $y_{j,1} \geqq \cdots \geqq y_{j,p_j}$. Obviously, we have $1 \leqq \sum_{1 \leqq i \leqq p_j} y_{j,i} \leqq 3$ (otherwise one more bin could be covered) and if we set $\vec{y}_j = (y_{j,1}, \cdots, y_{j,p_j})$, we have $\vec{y}_j \in S_{p_j}$. Let $\nu_j = \delta_{\vec{y}_j} \in M_1(S_{p_j})$. If we set $\eta_j = \mathcal{P}_{p_j}(\nu_j)$, we have $\eta_j = (1/p_j)(\sum_{1 \leqq i \leqq p_j} \delta_{y_{j,i}})$. We have $\mu_n = \sum_{j \leqq k_n} (p_j/n) \eta_j$, where $\sum_{j \leqq k_n} p_j/n = 1$, $\sum_{j \leqq k_n} (p_j/n)/p_j = k_n/n$, and $\eta_j \in D_{p_j}$. Theorem 1 then follows from Proposition 2.2 of [4].

Proof of Corollary 3. If we denote by $\lfloor t \rfloor$ the integer part of $t \in \mathbf{R}$, equation (4) is a consequence of (2), by taking $q = \lfloor n^{1/3} (\log n)^{-1/2} \rfloor$ and by elementary computations.

Proof of Corollary 4. According to Theorem 1, ν has a representation as in (1) with $d(\nu) = \sum_{k \geqq 1} \alpha_k/k$. If we apply (3) with $\mu = \nu$, we get the right-hand side of (5), since $D = 0$. To prove the left-hand side inequality, let us draw a random number N of items independent and distributed according to ν, and for $i \leqq n$, let us denote by N_i the number of items of size s_i. Let $a_N = \inf_{i \leqq n} N_i$. Among our items, we can find a_N collections that contain exactly one item of each size. Each of these collections allows us to cover $Q_n(s_1, \cdots, s_n)$ bins, so we can cover at least $a_N Q_n(s_1, \cdots, s_n)$ bins. The law of large numbers shows that $\lim_{N \to \infty} E(a_N)/N = 1/n$. Since

$$\begin{aligned} d(\nu) = \lim_{N \to \infty} E(Q_N(X_1, \cdots, X_n))/N &\geqq \lim_{N \to \infty} E(a_N Q_n(s_1, \cdots, s_n))/N \\ &= Q_n(s_1, \cdots, s_n)/n, \end{aligned}$$

this implies the result.

Proof of Corollary 5. This is an immediate consequence of (3), and of the fact that $ED = O(n^{1/2})$, as follows from the well-known properties of the Kolmogorov–Smirnov statistics.

Proof of Corollary 6. Obvious from Corollary 5.

Proof of Theorem 7. Suppose first that μ allows perfect covering. Then, from Theorem 1, we can find a sequence $\alpha_k \geqq 0$ with $\sum_{k \leqq 0} \alpha_k = 1$, $\sum_{k \geqq 1} \alpha_k/k = d(\mu) = \int x\, d\mu(x)$, and ν_k in $M_1(S_k)$, such that $\mu = \alpha_0 \delta_0 + \sum_{k \geqq 1} \alpha_k \mathcal{P}_k(\nu_k)$. We have

$$\int x\, d\mathcal{P}_k(\nu_k)(x) = \frac{1}{k} \int_{S_k} \sum_{i \leqq k} x_i\, d\nu_k(x_1, \cdots, x_k) \geqq 1/k \tag{6}$$

since $\sum_{i\leqq k} x_i \geqq 1$ for $(x_1, \cdots, x_k)$ in S_k. So we have

$$\int x\, d\mu(x) = \sum_{k\geqq 1} \alpha_k \int x\, d\mathscr{P}_k(\nu_k)(x)$$

$$\geqq \sum_{k\geqq 1} \alpha_k/k = d(\mu) = \int x\, d\mu(x).$$

This shows that we have $\int x\, d\mathscr{P}_k(\nu_k)(x) = 1/k$ whenever $\alpha_k \neq 0$. But (6) shows that this implies $\sum_{i\leqq k} x_i = 1\, \nu_k$ almost surely. If we set

$$R_k = \left\{(x_1, \cdots, x_k) \in \mathbf{R}^k;\ x_1, \cdots, x_k \geqq 0,\ \sum_{i\leqq k} x_i = 1\right\},$$

we then have $\nu_k(R_k) = 1$ whenever $\alpha_k \neq 0$. Since when $\alpha_k = 0$, the value of ν_k is irrelevent, we can assume that $\nu_k(R_k) = 1$ for $k \geqq 1$, so that $\nu_k \in M_1(R_k)$. It then follows from [3] that μ allows perfect packing.

Conversely, assume that μ allows perfect packing. Then, by the results of [3], we can write $\mu = \alpha_0\delta_0 + \sum_{k\geqq 1} \alpha_k \mathscr{P}_k(\nu_k)$, where $\sum_{k\geqq 0} \alpha_k = 1$ and $\nu_k \in M_1(R_k)$. Since $\nu_k \in M_1(R_k)$, we have, for $k \geqq 1$,

$$\int x\, d\mathscr{P}_k(\nu_k)(x) = \frac{1}{k} \int_{S_k} \sum_{i\leqq k} x_i\, d\nu_k\,(x_1, \cdots, x_k) = 1/k$$

so we have $\int x\, d\mu = \sum_{k\geqq 1} \alpha_k/k$. By Corollary 6, we have $d(\mu) \geqq \sum_{k\geqq 1} \alpha_k/k = \int x\, d\mu$. Since the reverse inequality is obvious, we have shown that $d(\mu) = \int x\, d\mu$, so μ allows perfect covering. □

Proof of Theorem 8. Let us fix f in $\mathscr{C}$. We first note that (as easily seen) for $k \geqq 1$, $x_1, \cdots, x_k$ in $[0, 1]$ with $\sum_{i\leqq k} x_i \geqq 1$, we have $\sum f(x_i) \geqq 1$. For ν in $M_1(S_k)$, we then have

$$\int f\, d\mathscr{P}_k(\nu) = \frac{1}{k} \int_{S_k} \sum_{i\leqq k} f(x_i)\, d\nu(x_1, \cdots, x_k) \geqq 1/k$$

since $\sum_{i\leqq k} x_i \geqq 1$ for $(x_1, \cdots, x_k) \in S_k$. So, when we have a representation of μ as in (1), we have $\int f\, d\mu \geqq \sum_{k\geqq 1} \alpha_k/k$. Since, by Theorem 1, we can find such a representation with $d(\mu) = \sum_{k\geqq 1} \alpha_k/k$, we have shown that $d(\mu) \leqq \inf_{f\in\mathscr{C}} \int f\, d\mu$.

To prove the converse inequality, let $\theta > d(\mu)$. Consider the set C_θ of probability measures η on $[0, 1]$ that can be written as $\eta = \sum_{k\geqq 0} \alpha_k \eta_k$, where $\alpha_k \geqq 0$, $\sum_{k\geqq 0} \alpha_k = 1$, $\sum_{k\geqq 1} \alpha_k/k \geqq \theta$, and $\mu_k \in D_k$ (where D_k is defined in the proof of Theorem 1). Then, by Corollary 8, we have $\mu \notin C_\theta$. It then follows from the Proposition 2.5 of [4] that there is a continuous function g on $[0, 1]$, such that $\int g\, d\mu < \theta$ and $\int g\, d\eta \geqq 1/k$ whenever $\eta \in D_k$, $k \geqq 1$. So we have $g(0) \geqq 0$, and since for $k \geqq 1$, we have $1/k \sum_{i\leqq k} \delta_{x_i} \in D_k$ whenever $0 \leqq x_i \leqq 1$, $1 \leqq \sum_{i\leqq k} x_i \leqq 3$, we also have $\sum_{i\leqq k} g(x_i) \geqq 1$. For $0 < x \leqq 1$, if m is the smallest integer for which $mx \geqq 1$, we have $mx \leqq 2$, so we have $mg(x) \geqq 1$, and $g(x) \geqq 0$. It follows that $g \geqq 0$. Define

$$h(t) = \inf\left\{\sum_{i\leqq k} g(x_i);\ k \geqq 1, \forall i,\ 1 \leqq i \leqq k,\ 0 \leqq x_i \leqq 1,\ t \leqq \sum_{i\leqq k} x_i \leqq 3\right\}.$$

Since $g \geqq 0$, it is easily seen that

$$h(t) = \inf\left\{\sum_{i\leqq k} g(x_i);\ k \geqq 1, \forall i,\ 1 \leqq i \leqq k,\ 0 \leqq x_i \leqq 1,\ t \leqq \sum_{i\leqq k} x_i \leqq 1 + t\right\}.$$

This implies that when $t, u \leqq 1$, and $t + u \leqq 1$, we have $h(t+u) \leqq h(t) + h(u)$. Also, we have $h \leqq g$, so we have $\int h\, d\mu < \theta$. Clearly, we have $h(1) \geqq 1$. Let now $f = \min(h, 1)$,

so we have $\int f\,d\mu < \theta$, and it is clear that $f \in \mathscr{C}$. So we have shown that $\inf_{f\in\mathscr{C}} \int f\,d\mu < \theta$. Since θ is arbitrary, the proof is complete.

3. Proof of Theorem 9. The proof will use many of the ideas of the proof of Theorem C of [5]. It is, however, made harder by the fact that in (5) the error term is of order $> n^{1/2}$; hence, it is not possible proving Theorem 9 to replace $Q_n(X_1, \cdots, X_n)$ by $d(\nu)$, where $\nu = 1/n \sum_{i\leqq n} \delta_{X_i}$ (as is done in [5] for bin packing). As in [5], we see that $\mathscr{C}_\mu$ is not empty (the point being that the map $f \to \int f\,d\mu$ is continuous for a certain compact topology on $\mathscr{C}$). We first recall some lemmas.

LEMMA 1 ([5]). *For some universal constant K, for each n, we can find independently and identically distributed random variables $X_1, \cdots, X_n$ distributed according to μ, and processes $W_{n,\mu,t}$, $W^-_{n,\mu,t}$ jointly distributed like $W_{\mu,t}$, $W^-_{\mu,t}$, such that*

$$ES_n \leqq K(\log n)^2 n^{-1/2};\ ES_n^- \leqq K(\log n)^2 n^{-1/2} \tag{7}$$

where

$$S_n = \sup_{0\leqq t\leqq 1} |(\operatorname{card}\{i \leqq n;\ X_i \leqq t\} - nF_\mu(t))/\sqrt{n} - W_{n,\mu,t}| \tag{8}$$

$$S_n^- = \sup_{0\leqq t\leqq 1} |(\operatorname{card}\{i \leqq n;\ X_i < t\} - nF_\mu^-(t))/\sqrt{n} - W^-_{n,\mu,t}|. \tag{9}$$

For f in $\mathscr{C}$, we consider the random variables defined by

$$T_{n,f} = -\int W_{n,\mu,t}\,d\gamma_f(t) + \sum_{i\in J} (f(z_i) - f(z_i)^-)(W_{n,\mu,z_i} - W^-_{n,\mu,z_i})$$

$$T_f = -\int W_{\mu,t}\,d\gamma_f(t) + \sum_{i\in J} (f(z_i) - f(z_i)^-)(W_{\mu,z_i} - W^-_{\mu,z_i})$$

so that

$$T = \inf_{f\in\mathscr{C}_\mu} T_f.$$

We set

$$T_n = \inf_{f\in\mathscr{C}_\mu} T_{n,f}.$$

LEMMA 2 ([5]). *For f in $\mathscr{C}$, and a probability measure η on $[0, 1]$ such that $(u_i)_{i\in J}$ is an enumeration of the points t for which $\eta(\{t\}) > 0$, we have*

$$\int f\,d\eta = f^-(1) - \int F_\eta\,d\gamma_f + \sum_{i\in J} (f(u_i) - f^-(u_i))\eta(\{u_i\}).$$

LEMMA 3 ([5]). *With probability one, we have for all f in $\mathscr{C}$ that*

$$\left| \int f\,d\nu - \int f\,d\mu - n^{-1/2} T_{n,f} \right| \leqq 1/n + 2S_n + S_n^-$$

where $\nu = 1/n \sum_{i\leqq n} \delta_{X_i}$.

The following is proved exactly as Lemma 6 of [5].

LEMMA 4. *Consider the function*

$$\psi(x) = \inf_{f\in\mathscr{C}} \left(\int f\,d\mu + xT_f - d(\mu) \right).$$

Then $\psi(0) = 0$, ψ is concave, and its right derivative at zero is equal to T.

The next lemma is similar to Lemma 2 of [4].

LEMMA 5. *Let $p, q, n > 0$. Consider a family $(a_i)_{i \leqq q}$, $0 \leqq a_i \leqq 1$, and for $1 \leqq j \leqq p$, consider left continuous nonincreasing functions $(f_j)_{j \leqq p}$ on $[0, 1]$. Set*

$$h = \sup_{0 \leqq t \leqq 1} \left\{ n \sum_{j \leqq p} f_j(t) - \operatorname{card} \{1 \leqq i \leqq q;\ a_i \geqq t\} \right\}.$$

Then there are disjoint subsets $(H_j)_{j \leqq p}$ of $\{1, \cdots, q\}$ with $\operatorname{card} H_j \leqq nf_j(0)$ and integers $(h_j)_{j \leqq p}$ such that $\sum_{j \leqq p} h_j \leqq h + p$ and that for each $j \leqq p$, each t in $[0, 1]$, we have

$$\operatorname{card} \{i \in H_j;\ a_i \geqq t\} \geqq nf_j(t) - h_j. \tag{10}$$

Proof. For $1 \leqq j \leqq p$, let $n_j = \lfloor nf(0) \rfloor$. *For* $l \leqq n_j$, define

$$b_{l,j} = \sup \{t;\ 0 \leqq t \leqq 1,\ nf_j(t) \geqq l\}.$$

Since f_j is left continuous, we have $nf_j(b_{l,j}) \geqq l$. Note also that $(b_{l,j})$ is nonincreasing in l, i.e., $b_{l+1,p} \leqq b_{l,j}$ for $1 \leqq l, l+1 \leqq n_j$.

FACT. For $0 \leqq t \leqq 1$, we have

$$nf_j(t) - 1 < \operatorname{card} \{l \leqq n_j;\ b_{l,j} \geqq t\} \leqq nf_j(t). \tag{11}$$

Proof of Fact. Let r be the largest integer with $r \leqq nf_j(t)$, so $r > nf_j(t) - 1$. By definition of $b_{l,j}$, we have $b_{l,j} \geqq t$ if and only if $1 \leqq l \leqq r$. □

We return to the proof of Lemma 2. Consider the index set

$$U = \{(l, j);\ 1 \leqq j \leqq p,\ 1 \leqq l \leqq n_j\}.$$

For $u = (l, j) \in U$, let $b_u = b_{l,j}$. By summation of the inequalities (11), we get for each t,

$$n \sum_{j \leqq p} f_j(t) \geqq \operatorname{card} \{u \in U;\ b_u \geqq t\}$$

so, by the definition of h,

$$\sup_{0 \leqq t \leqq 1} \{\operatorname{card} \{u \in U;\ b_u \geqq t\} - \operatorname{card} \{i \leqq q;\ a_i \geqq t\}\} \leqq h.$$

Using [4], Lemma 1, we get a subset H of U, with $\operatorname{card} H \geqq \operatorname{card} U - h$, and a one-to-one map ϕ from H to $\{1, \cdots, q\}$ such that $b_u \leqq a_{\phi(u)}$ for u in H. Consider the set H_j of those elements of $\{1, \cdots, q\}$ of the form $\phi((l, j))$, $l \leqq n_j$, $(l, j) \in H$, and let $m_j = n_j - \operatorname{card} H_j$. for each t, we have

$$\begin{aligned} nf_j(t) &< 1 + \operatorname{card} \{l \leqq n_j;\ b_{l,j} \geqq t\} \\ &< 1 + \operatorname{card} \{l \leqq n_j;\ (l, j) \in H;\ b_{l,j} \geqq t\} + m_j \\ &\leqq 1 + m_j + \operatorname{card} \{i \in H_j;\ a_j \geqq t\} \end{aligned}$$

so (10) holds with $h_j = 1 + m_j$. We have $\sum_{j \leqq p} h_j = p + \operatorname{card} U - \operatorname{card} H \leqq p + h$. This concludes the proof.

We now start to prove Theorem 9. By Corollary 4 and Theorem 8, we have, for all $f \in \mathscr{C}$,

$$Q_n(X_1, \cdots, X_n) \leqq n\, d(\nu) \leqq n \int f\, d\nu.$$

By Lemma 3, we have

$$Q_n(X_1, \cdots, X_n) \leqq n \int f\, d\mu + n^{1/2} T_{n,f} + n^{-1/2} + n^{1/2}(2S_n + S_n^-).$$

So, if $f \in \mathscr{C}_\mu$, we have

$$Q_n(X_1, \cdots, X_n) - n\, d(\mu) \leqq n^{1/2} T_{n,f} + n^{-1/2} + n^{1/2}(2S_n + S_n^-)$$

and hence

$$n^{-1/2}(Q_n(X_1, \cdots, X_n) - n\, d(\mu)) - T_n \leqq 2S_n + S_n^- + n^{-1}. \tag{12}$$

It will be much harder to get an inequality in the reverse direction. We set $m = \lfloor n^{2/3} \rfloor$, $\varepsilon = n^{1/2}/m$. Consider, for $0 \leqq t \leqq 1$,

$$G_n(t) = \max\,(0, \min_{u \leqq t}\,(\mu([u, 1]) + \varepsilon W^-_{n,\mu,u})). \tag{13}$$

This is a nonnegative nonincreasing function, and it is easily checked to be left-continuous. Also $G_n(0) = 1$. So there exists a probability measure θ on $[0, 1]$ such that for all t, we have $\theta([t, 1]) = G_n(t)$. We have $\theta(\{t\}) > 0$ only if $\mu(\{t\}) > 0$. It follows from (9) that

$$\begin{aligned}
\nu([t, 1]) &= 1 - F_\nu^-(t) \geqq 1 - F_\mu^-(t) + n^{-1/2} W^-_{n,\mu,t} - n^{-1/2} S_n^- \\
&= \mu([t, 1]) + n^{-1/2} W^-_{n,\mu,t} - n^{-1/2} S_n^- \\
&= (1 - (m/n))\mu([t, 1]) + (m/n)(\mu([t, 1]) + \varepsilon W_{n,\mu,t}) - n^{-1/2} S_n^-.
\end{aligned}$$

This means that, for $0 \leqq t \leqq 1$,

$$\text{card}\,\{i \leqq n;\, X_i \geqq t\} \geqq (n - m)\mu([t, 1]) + m\theta([t, 1]) - n^{1/2} S_n^-.$$

We apply Lemma 5 with $q = n$, $p = 2$, and we find two disjoint sets $H_1, H_2 \subset \{1, \cdots, n\}$ and with card $H_1 \leqq n - m$, card $H_2 \leqq m$, and h_1, h_2 with $h_1 + h_2 \leqq n^{1/2} S_n^{-1} + 2$ such that for each t in $[0, 1]$,

$$\text{card}\,\{i \in H_1;\, X_1 \geqq t\} \geqq (n - m)\mu([t, 1]) - h_1, \tag{14}$$

$$\text{card}\,\{i \in H_2;\, X_2 \geqq t\} \geqq m\theta([t, 1]) - h_2. \tag{15}$$

Let $q_1 = \text{card}\, H_1$, $q_2 = \text{card}\, H_2$, so $q_1 \leqq n - m$, $q_2 \leqq m$, and making $t = 0$ in (14) and (15), we have $q_1 \geqq n - m - h_1$, $q_2 \geqq m - h_2$. We have, for $0 \leqq t \leqq 1$,

$$\text{card}\,\{i \in H_1;\, X_i \geqq t\} \geqq q_1 \mu([t, 1]) - h_1, \tag{16}$$

$$\text{card}\,\{i \in H_2;\, X_i \geqq t\} \geqq q_2 \theta([t, 1]) - h_2. \tag{17}$$

We now use (4) with $n = q_1$ to see that the number Q_1 of bins that can be covered by the items X_i, $i \in H_1$ is such that

$$Q_1 \geqq q_1\, d(\mu) - h_1 - n^{1/2}\varepsilon_{q_1} \geqq (n - m)\, d(\mu) - 2h_1 - n^{1/2}\varepsilon_n'',$$

where ε_n'' is a nonrandom quantity with $\lim_{n \to \infty} \varepsilon_n'' = 0$. We now use (3) with $n = q_2$ and $\mu = \theta$ to see that the number Q_2 of bins that can be covered by the items X_i, $i \in H_2$ satisfies

$$\begin{aligned}
Q_2 &\geqq q_2\, d(\theta) - h_2 - Km^{3/5}(\log m)^{3/10} \\
&\geqq m\, d(\theta) - 2h_2 - Km^{3/5}(\log m)^{3/2}.
\end{aligned}$$

We have shown that

$$Q_n(X_1, \cdots, X_n) \geqq (n - m)\, d(\mu) + m\, d(\theta) - 2n^{1/2} S_n - n^{1/2}\varepsilon_n'$$

where ε_n' is a nonrandom quantity with $\lim_{n \to \infty} \varepsilon_n' = 0$. So we have

$$Q_n(X_1, \cdots, X_n) \geqq n\, d(\mu) + m(d(\theta) - d(\mu)) - 2n^{1/2} S_n - n^{1/2}\varepsilon_n'. \tag{18}$$

We will now estimate $d(\theta)$ using Theorem 8. For this we need to estimate F_θ. This is the purpose of the next lemma.

LEMMA 6. *Let* $S'_n = \sup_{0\leqq t\leqq 1} |F_\theta(t) - F_\mu(t) - \varepsilon W_{n,\mu,t}|$. *Then* $E(S'_n) = o(\varepsilon) = o(n^{-1/6})$.

Proof. Let $G'_n(t) = \mu([t,1]) + \varepsilon W^-_{n,\mu,t}$. We note that

$$F_\theta(t) = 1 - \lim_{u>t, u\to t} \theta([u,1])$$

$$F_\mu(t) - \varepsilon W_{n,\mu,t} = 1 - \lim_{u>t, u\to t} (\mu([u,1]) + \varepsilon W^-_{n,\mu,u})$$

so that

$$S'_n = \sup_{0\leqq t\leqq 1} |G'_n(t) - G_n(t)|.$$

Let $M_n = \sup_{0\leqq t\leqq 1} |W_{n,t}|$. We first note the obvious fact that $S'_n \leqq 2\varepsilon M_n$. Since $E(M_n)$ is finite and the distribution of M_n is independent of n, it suffices to show that given $\gamma > 0$, for n large enough, we have $S'_n < \gamma\varepsilon$ with probability $> 1-\gamma$. There exists a real $\delta > 0$ and $M > 0$ independent of n such that with probability $\geqq 1-\gamma$, we have $M_n \leqq M$ and $|W_{n,t} - W_{n,u}| \leqq \gamma$ whenever $|t-u| \leqq \delta$. Let n be large enough that $\varepsilon \leqq \delta/2M$. If for $u < t$, we have $\mu([u,1]) \geqq \mu([t,1]) + \delta$, since $\varepsilon|W^-_{n,\mu,u}| + \varepsilon|W_{n,\mu,t}| \leqq \delta$, we have

$$\mu([u,1]) + \varepsilon W^-_{n,\mu,u} \geqq \mu([t,1]) + \varepsilon W^-_{n,\mu,t}.$$

If $\mu([u,1]) \leqq \mu([t,1]) + \delta$, then $|F^-_\mu(u) - F^-_\mu(t)| \leqq \delta$, so we have $|W^-_{n,\mu,u} - W^-_{n,\mu,t}| \leqq \gamma$, and hence

$$\mu([u,1]) + \varepsilon W^-_{n,\mu,u} \geqq \mu([t,1]) + \varepsilon W^-_{n,\mu,t} - \varepsilon\gamma.$$

This shows that, with probability $\geqq 1-\gamma$, for all t, we have $G_n(t) \geqq G'_n(t) - \varepsilon\gamma$. The same argument with $t = 1$ shows that $G'_n(t) \geqq -\varepsilon\gamma$ for all t, so that $G_n(t) \leqq \max(0, G'_n(t)) \leqq G'_n(t) + \varepsilon\gamma$. This concludes the proof.

For f in $\mathscr{C}$, we have, by Lemma 2, that

$$\int f\,d\theta = f^-(1) - \int F_\theta\,d\gamma_f + \sum_{i\in J} (f(z_i) - f(z_i)^-)\theta(\{z_i\}).$$

Since $\theta(\{z_i\}) = F_\theta(z_i) - F^-_\theta(z_i)$, and since $\sum_{i\in J} |f(z_i) - f(z_i)^-| \leqq 1$ for f in $\mathscr{C}$, we see from the definition of S'_n that

$$\left| \int f\,d\theta - (f^-(1) - \int F_\mu\,d\gamma_f - \varepsilon \int W_{n,\mu,t}\,d\gamma_f \right.$$

$$\left. + \sum_{i\in J} (f(z_i) - f(z_i)^-)(\mu(\{z_i\}) + \varepsilon W_{n,\mu,z_i} - \varepsilon W^-_{n,\mu,z_i})) \right| \leqq 3S'_n.$$

Using Lemma 2 again, we see that

$$\left| \int f\,d\theta - \int f\,d\mu - \varepsilon T_{n,f} \right| \leqq 3S'_n.$$

So, since $d(\theta) = \inf_{f\in\mathscr{C}} \int f\,d\theta$, we have

$$\left| d(\theta) - d(\mu) - \inf_{f\in\mathscr{C}} \left(\int f\,d\mu + \varepsilon T_{n,f} - d(\mu) \right) \right| \leqq 3S'_n$$

i.e., $|d(\theta) - d(\mu) - \psi_n(\varepsilon)| \leqq 3S'_n$, where ψ_n is defined as ψ but using $T_{n,f}$ instead of T_n. From (18), we then have

$$Q_n(X_1, \cdots, X_n) \geqq n\,d(\mu) + m\psi_n(\varepsilon) - 2n^{1/2}S_n - n^{1/2}\varepsilon'_n - 3mS'_n$$

so, since $\varepsilon = n^{1/2}/m$,

$$(19)\quad (Q_n(X_1, \cdots, X_n) - nd(\mu))n^{-1/2} - T_n \geqq (\psi_n(\varepsilon)/\varepsilon - T_n) - 2S_n - \varepsilon'_n - 3n^{1/6}S'_n.$$

The random variable $U_n = \psi_n(\varepsilon)/\varepsilon - T_n$ is distributed like $\psi(\varepsilon)/\varepsilon - T$. By Lemma 5 and dominated convergence, we have that $\lim_{n\to\infty} E|U_n| \to \infty$. By Lemma 6, $\lim_{n\to\infty} n^{1/6} E(S_n') = 0$. It then follows that the expectation of the absolute value of the right-hand side of (19) goes to zero. Together with (12), this concludes the proof.

4. Proof of Theorem 2. The proof will be similar, but somewhat simpler than the proof of theorem of [5]. The proof is simpler because the strategy is less sophisticated (and, of course, the error term obtained is of a larger order). We have not been able to get an error term as small as in [4], and with our approach, bin covering seems genuinely harder than bin packing. Our proof will make essential use of the following lemma, which proof is entirely similar to that of Lemma 10 of [4].

LEMMA 7. *Let γ be a positive measure on $[0,3]^2$. Let $m \leqq n$. Assume that $m/n \leqq \|\gamma\| \leqq (m+1)/n$. Consider the three positive measures given by, for a Borel set U,*

$$\gamma'(U) = \gamma(U \times [0,3]), \qquad \gamma''(U) = \gamma([0,3] \times U)$$

$$\eta(U) = \gamma(\{(x,y);\ x+y \in U\}).$$

In other words, γ' and γ'' are the marginals of γ, and η is the distribution of $x+y$.

Consider two sequences $(l_i)_{i\leqq m}$, $(y_i)_{i\leqq m}$ of numbers, $0 \leqq l_i, y_i \leqq 1$ and $a, b > 0$. Assume that for each t in $[0,3]$,

$$\text{card}\,\{i \leqq m;\ l_i \geqq t\} \geqq n\gamma'([t,3]) - a, \tag{20}$$

$$\text{card}\,\{i \leqq m;\ y_i \geqq t\} \geqq n\gamma''([t,3]) - b. \tag{21}$$

Then there exists a one to one and onto map ϕ from $\{1, \cdots, m\}$ to $\{1, \cdots, m\}$ such that for each $t \geqq 0$,

$$\text{card}\,\{i \leqq m;\ l_i + y_{\phi(i)} \geqq t\} \geqq n\eta([t,\infty)) - a - b - Km^{1/3}(\log n)^{1/2}, \tag{22}$$

where K is a universal constant.

Remark. It is likely that it is possible with some extra effort to remove the factor $(\log n)^{1/2}$ in (22), which would remove the term $(\log n)^{3/10}$ in (3). There is, however, little motivation to do that, since we see no reason why the exponent $\frac{1}{3}$ in (22) should be sharp.

We will need some distributions associated with μ. For $l \geqq 1$, consider the positive measure λ_l on $[0,1]$ such that

$$\lambda_l = \sum_{k\geqq l} (\alpha_k/k) \int_{S_k} \delta_{x_l}\, d\nu_k(x_1, \cdots, x_k)$$

so $\|\lambda_l\| = \sum_{k\geqq l} \alpha_k/k = \beta_l$ and $\mu = \alpha_0\delta_0 + \sum_{l\geqq 1} \lambda_l$. For x in R_k, $k \geqq l$, let $u_{k,l}(x) = \sum_{1\leqq i\leqq l} x_i$. Consider the following three positive measures on $[0,3]$,

$$\theta_l = (\alpha_l/l) \int_{S_l} \delta_{u_{l,l}(x)}\, d\nu_l(x_1, \cdots, x_l)$$

$$n_l = \sum_{k\geqq l} (\alpha_k/k) \int_{S_k} \delta_{u_{k,l}(x)}\, d\nu_k(x_1, \cdots, x_k)$$

$$\xi_l = \sum_{k\geqq l+1} (\alpha_k/k) \int_{S_k} \delta_{u_{k,l}(x)}\, d\nu_k(x_1, \cdots, x_k)$$

$$= \eta_l - \theta_l.$$

We note that $\eta_1 = \lambda_1$, $\|\eta_l\| = \beta_l$, and $\|\xi_l\| = \beta_{l+1}$. After normalization, θ_l, η_l, ξ_l can be interpreted as follows. θ_l is the distributions of the level of bins that are covered by l

elements. η_l (respectively, ξ_l) is the distribution of the sum of the l largest items in bins that are covered by at least l (respectively, at least $l+1$) elements. We also need the distribution γ_l on $[0,3]^2$ given by

$$\gamma_l = \sum_{k \geqq l+1} (\alpha_k / k) \int_{S_k} \delta_{v(x)}\, d\nu_k(x_1, \cdots, x_k),$$

where $v(x) = (u_{k,l}(x), x_{l+1})$. Note that $\|\gamma_l\| = \beta_{l+1}$. Using the fact that $u_{k,l}(x) + x_{l+1} = u_{k,l+1}(x)$, we see that the first marginal of γ_l is ξ_l, the second is λ_{l+1}, and the distribution of $x+y$ is η_{l+1}.

For each i, let $f_i(t) = \lambda_i([t,3])$, so $\sum_{i \geqq 1} f_i(t) = \mu([t,1])$. We set $n_i = \lfloor n f_i(0) \rfloor = \lfloor n\beta_i \rfloor$. We now prove Theorem 2. We have for each $0 \leqq t \leqq 3$,

$$\sum_{i \leqq q} n f_i(t) - \text{card}\,\{i \leqq n;\ s_i \geqq t\} \leqq D. \tag{23}$$

By Lemma 5, (used with $q = n$, and q instead of p), we can find disjoint subsets $(H_j)_{j \leqq q}$ of $\{1, \cdots, n\}$ with card $H_j \leqq \lfloor n f_j(0) \rfloor = n_j$ and integers $(h_j)_{j \leqq q}$ such that $\sum_{j \leqq q} h_j \leqq D + q$ and that for $j \leqq q$, for each t in $[0,3]$,

$$\text{card}\,\{i \in H_j;\ s_i \geqq t\} \geqq n f_j(t) - h_j. \tag{24}$$

By adding some zero size elements, we can actually assume that card $H_j = n_j$. The core of the proof is to show, by induction over $r \leqq q$, that the items s_i for $i \in \bigcup_{j \leqq r} H_j$ can be packed in a collection of bins whose occupancy levels $(L_u)_{u \in U}$ satisfy for each t in $[0,3]$,

$$\text{card}\,\{u \in U;\ L_u \geqq t\} \geqq n\left(\sum_{1 \leqq j \leqq r} \theta_j + \xi_r\right)([t,3]) - \sum_{1 \leqq j \leqq r} h_j - \sum_{1 \leqq j < r} K n_j^{1/3} (\log n)^{1/2} - 2(r-1). \tag{25}$$

(We index the bins by an arbitrary set U since their number is irrelevant at that stage.) We note that $\lambda_1 = \theta_1 + \xi_1$, so for $r = 1$, there is nothing to prove, since (25) follows from (24) when $j = 1$.

We now perform the induction step from r to $r+1$. For simplicity, we let $a = \sum_{1 \leqq j \leqq r} h_j - \sum_{1 \leqq j < r} K n_j^{1/3} (\log n)^{1/2} - 2(r-1)$ and $\theta = \sum_{1 \leqq j \leqq r} \theta_j$, so (25) becomes for all t in $[0,3]$,

$$n(\theta + \xi_r)([t,3]) - \text{card}\,\{u \in U;\ L_\mu \geqq t\} \leqq a.$$

It follows from Lemma 2 (used on $[0,3)$ instead of $[0,1)$), that we can find two disjoint subsets U_1, U_2 of U with card $U_1 \leqq n\|\xi_r\|$, card $U_2 \leqq n\|\theta\|$, and a_1, a_2 with $a_1 + a_2 \leqq a + 2$ such that for $0 \leqq t \leqq 3$, we have

$$\text{card}\,\{u \in U_1;\ L_u \geqq t\} \geqq n\xi_r([t,3]) - a_1 \tag{26}$$

$$\text{card}\,\{u \in U_2;\ L_u \geqq t\} \geqq n\theta([t,3]) - a_2. \tag{27}$$

By adding some new empty bins, we can assume that card $U = \lfloor n\|\xi_r\| \rfloor$. We are now in position to use Lemma 7, with $m = \lfloor n\|\xi_r\| \rfloor = n_{r+1}$, $\gamma = \gamma_r$, $\gamma' = \xi_r$, $\gamma'' = \lambda_{r+1}$, $\eta = \eta_{r+1}$, to find a one-to-one and onto map ϕ from H_{r+1} to U such that for any $t \geqq 0$, we have

$$\text{card}\,\{i \in H_{r+1};\ L_{\phi(i)} + s_i \geqq t\} \geqq n\eta_{r+1}([t,3]) - a_1 - h_{r+1} - K n_{r+1}^{1/3} (\log n)^{1/2}. \tag{28}$$

We now attribute the element s_i to bin $\phi(i)$. Combining (28) with (27) finishes the induction hypothesis, since $\eta_{r+1} = \theta_{r+1} + \xi_{r+1}$.

Comment. When considering the same problem for bin packing ([3], [4]) in pairing the bin occupancy level L_i with the elements s_i controlled by λ_{r+1}, we took advantage of the fact that these elements s_i are $\leqq 1/(r+1)$, and hence that it is possible to disregard a rather large number of them in the pairing procedure since these elements can be efficiently packed in any case. We do not see how to take advantage of a similar feature in the case of bin covering.

If we use (25) for $r=q$, $t=1$, we see that

$$\operatorname{card}\{u\in U;\ L_u\geqq 1\}\geqq n\left(\sum_{1\leqq j<q}\alpha_j/j\right)-D-3q-\sum_{1\leqq j<q}Kn_j^{1/3}(\log n)^{1/2}$$

and this proves (2).

To prove (3), we note that $\sum_{k\geqq 1}\beta_k=\sum_{k\geqq 1}\alpha_k\leqq 1$, so by Holder's inequality, we have $\sum_{j\leqq q}\beta_j^{1/3}\leqq q^{2/3}$. Also $\beta_q\leqq 1/q$. Then (3) follows from (2) by taking $q=\lfloor n^{2/5}(\log n)^{-3/10}\rfloor$.

REFERENCES

[1] L. BREIMAN (1968), *Probability*, Addison-Wesley, Reading, MA.

[2] J. CSIRIK, J. B. G. FRENK, G. GALAMBOS, AND A. H. G. RINNOOY KAN (1986), *A probabilistic analysis of the dual bin packing problem*, preprint.

[3] W. RHEE (1988), *Optimal bin packing with items of random sizes*, Math. Oper. Res., 13, pp. 140-151.

[4] W. RHEE AND M. TALAGRAND (1989), *Optimal bin packing with items of random sizes* II, SIAM J. Comput., pp. 139-151.

[5] W. RHEE AND M. TALAGRAND (1987), *Optimal bin packing with items of random sizes* III, preprint.

[6] W. RHEE AND M. TALAGRAND (1988), *Some distributions that allow perfect packing*, J. Assoc. Comput. Mach., 35, pp. 564-578.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 499–532, June 1989

006

CASCADING DIVIDE-AND-CONQUER: A TECHNIQUE FOR DESIGNING PARALLEL ALGORITHMS*

MIKHAIL J. ATALLAH†, RICHARD COLE‡, AND MICHAEL T. GOODRICH§

Abstract. Techniques for parallel divide-and-conquer are presented, resulting in improved parallel algorithms for a number of problems. The problems for which improved algorithms are given include segment intersection detection, trapezoidal decomposition, and planar point location. Efficient parallel algorithms are algo given for fractional cascading, three-dimensional maxima, two-set dominance counting, and visibility from a point. All of the algorithms presented run in $O(\log n)$ time with either a linear or a sublinear number of processors in the CREW PRAM model.

Key words. parallel algorithms, parallel data structures, divide-and-conquer, computational geometry, fractional cascading, visibility, planar point location, trapezoidal decomposition, dominance, intersection detection

AMS(MOS) subject classifications. 68E05, 68C05, 68C25

1. Introduction.

This paper presents a number of general techniques for parallel divide-and-conquer. These techniques are based on nontrivial generalizations of Cole's recent parallel merge sort result [13] and enable us to achieve improved complexity bounds for a large number of problems. In particular, our techniques can be applied to any problem solvable by a divide-and-conquer method such that the subproblem merging step can be implemented using a restricted, but powerful, set of operations, which include (i) merging sorted lists, (ii) computing the values of labeling functions on elements stored in sorted lists, and (iii) changing the identity of elements in a sorted list monotonically. The elements stored in such sorted lists need not belong to a total order, so long as the computation can be specified so that we will never try to compare two incomparable elements. We demonstrate the power of these techniques by using them to design efficient parallel algorithms for solving a number of fundamental problems from computational geometry.

The general framework is one in which we want to design efficient parallel algorithms for the CREW PRAM or EREW PRAM models. Recall that the CREW PRAM model is the synchronous shared memory model in which processors may simultaneously read from any memory location but simultaneous writes are not allowed. The EREW PRAM model does not allow for any simultaneous access to a memory cell. Our goal is to find algorithms that run as fast as possible and are efficient in the following sense: if $p(n)$ is the processor complexity, $t(n)$ the parallel time complexity, and $seq(n)$ the time complexity of the best-known sequential algorithm for the problem

* Received by the editors September 14, 1987; accepted for publication (in revised form) August 12, 1988. This paper appeared in preliminary form as [3] and as portions of [17].

† Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by the Office of Naval Research under grants N00014-84-K-0502 and N00014-86-K-0689, and by the National Science Foundation under grant DCR-84-51393, with matching funds from AT&T.

‡ Courant Institute, New York University, New York, New York 10012. The research of this author was supported in part by National Science Foundation grants DCR-84-01633 and CCR-8702271, and by an Office of Naval Research grant N00014-85-K-0046.

§ Department of Computer Science, Johns Hopkins University, Baltimore, Maryland 21218. The research of this author was supported by Office of Naval Research grants N00014-84-K-0502 and N000-86-K-0689, National Science Foundation grants DCR-84-51393, with matching funds from AT&T, and CCR-88-10568, and a David Ross grant from the Purdue Research Foundation.

under consideration, then $t(n) * p(n) = O(seq(n))$. If the product $t(n) * p(n)$ achieves the sequential lower bound for the problem, then we say the algorithm is *optimal.* When specifying the processor complexity, we omit the "big oh," e.g., we say "n processors" rather than "$O(n)$ processors"; this is justified because we can always save a constant factor in the number of processors at a cost of the same constant factor in the running time. In all of the problems listed below, we achieve $t(n) = O(\log n)$ and, simultaneously (except for planar point location), an optimal $t(n) * p(n)$.

Previous work on parallel divide-and-conquer has produced relatively few algorithms that are optimal in the above sense. Exceptions to this include some of the previous algorithms for the convex hull problem [1], [4], [6], [18], [27] and the problem of circumscribing a convex polygon with a minimum-area triangle [1]. Unfortunately, each of these approaches was very problem-specific. Thus, there is a need for techniques of wider scope.

This is in fact the motivation for our work, for we give a number of general techniques for efficiently solving problems in parallel by divide-and-conquer. We model the divide-and-conquer paradigm as a binary tree whose nodes contain sorted lists of some kind. The computation involves computing on this tree in a recursively defined bottom-up fashion using lists of items and labeling functions defined for each node in the tree. In Cole's scheme [13], the list at a node was defined to be the sorted merge of the two lists stored at its children. In our scheme, however, the lists at a node of the tree can depend on the lists of its children in more complex ways. For example, in our solution to the *segment intersection detection* problem, the lists at a node depend on computing, in addition to merges, set difference operations that are not directly solvable by the "cascading" method used by Cole [13]. Such operations arise here because the lists at a node contain segments ordered by their intersections with a vertical line (the so-called "above" relationship), which is obviously not a total order. One may be tempted to try to solve this problem by delaying the performance of these set difference operations until the end of the computation. Unfortunately, this is not feasible for many reasons, not the least of which is that this approach could lead to a situation in which a processor tries to compare two incomparable items. Nor does it seem possible to explicitly perform the set difference operations on-line without sacrificing the time-efficiency of the cascading method. Our solution avoids both of these problems by using an on-line "identity-changing" technique.

Another significant contribution of this paper is an optimal parallel construction of the "fractional cascading" data structure of Chazelle and Guibas [12]. This too is based on a generalization of Cole's method [13] in the sense that instead of having the computation proceeding up and down a tree, it now moves around a directed graph (possibly with cycles). Our solution to fractional cascading is quite different from the sequential method of Chazelle and Guibas (their method relies on an amortization scheme to achieve a linear running time).

The following is a list of the problems for which our techniques result in improved complexity bounds. Unless otherwise specified, each performance bound is expressed as a pair $(t(n), p(n))$, where $t(n)$ and $p(n)$ are the time and processor complexities, respectively, in the CREW PRAM model.

Fractional cascading. Given a directed graph $G = (V, E)$, such that every node v contains a sorted list $C(v)$, construct a data structure that, given a walk $(v_1, v_2, \cdots, v_m)$ in G and an arbitrary element x, enables a single processor to locate x quickly in each $C(v_i)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. In [12] Chazelle and Guibas gave an elegant $O(n)$ time, $O(n)$ space, sequential construction, where $n = \sum_{v \in V} |C(v)|$. We give a $(\log n, n/\log n)$ construction.

Trapezoidal decomposition. Given a set S of n line segments in the plane, determine for each segment endpoint p the first segment "stabbed" by the vertical ray emanating upward (and downward) from p. A $(\log^2 n, n)$ solution to this problem was given by Aggarwal et al. in [1], later improved to $(\log n \log\log n, n)$ by Atallah and Goodrich in [5]. We improve this to $(\log n, n)$.

Planar point location. Given a subdivision of the plane into (possibly unbounded) polygons, construct, in parallel, a data structure that, once built, enables one processor to determine for any query point p the polygonal face containing p. Let $Q(n)$ denote the time for performing such a query, where n is the number of edge segments in the subdivision. A $(\log^2 n, n)$, $Q(n) = O(\log^2 n)$ solution was given by Aggarwal et al. in [1], later improved to $(\log n \log\log n, n)$, $Q(n) = O(\log n)$ by Atallah and Goodrich in [5]. In [14] Dadoun and Kirkpatrick further improved this to $(\log n \log^* n, n)$, $Q(n) = O(\log n)$. We give a $(\log n, n)$, $Q(n) = O(\log n)$ solution.

Segment intersection detection. Given a set S of n line segments in the plane, determine if any two segments in S intersect. A $(\log^2 n, n)$ solution was given in [1], later improved to $(\log n \log\log n, n)$ in [5]. We improve this to $(\log n, n)$.

Three-dimensional maxima. Given a set S of n points in three-dimensional space, determine which points are maxima. A *maximum* in S is any point p such that no other point of S has x, y, and z coordinates that simultaneously exceed the corresponding coordinates of p. A $(\log n \log\log n, n)$ solution was given in [5]. We improve this to $(\log n, n)$.

Two-set dominance counting. Given a set $A = \{q_1, q_2, \cdots, q_l\}$ and a set $B = \{r_1, r_2, \cdots, r_m\}$ of points in the plane, determine for each point r_i in B the number of points in A whose x and y coordinates are both less than the corresponding coordinates of r_i. The problem size is $n = l + m$. A $(\log n \log\log n, n)$ solution was given in [5]. We improve this to $(\log n, n)$.

Visibility from a point. Given n line segments such that no two intersect (except possibly at endpoints) and a point p, determine that part of the plane visible from p if all the segments are opaque. A $(\log n \log\log n, n)$ solution was given in [5]. We improve this to $(\log n, n)$.

We recently learned that Reif and Sen [24] solved planar point location, trapezoidal decomposition, segment intersection and visibility in randomized $O(\log n)$ time using $O(n)$ processors in the CREW PRAM model. All of our algorithms are deterministic.

This paper is organized as follows. In § 2 we present a generalized version of the cascading merge procedure and in § 3 we give our method for doing fractional cascading in parallel. In § 4 we show how to apply the fractional cascading technique to a data structure we call the *plane sweep tree*, showing how to solve the trapezoidal decomposition and point location problems. In § 5 we show how to extend the cascading merge technique to allow for cascading in the "above" partial order of line segments, giving solutions to the problems of building the plane sweep tree and solving the intersection detection problem. In § 6 we use the cascading divide-and-conquer technique to compute labeling functions and show how to use this approach to solve three-dimensional maxima, two-set dominance counting, and visibility from a point. Finally, in § 7, we briefly describe how most of our algorithms can be implemented in the EREW PRAM model with the same time and processor bounds as our CREW PRAM algorithms, and we conclude in § 8.

2. A generalized cascading merge procedure. In this section we present a technique for a generalized version of the merge sorting problem. Suppose we are given a binary tree T (not necessarily complete) with items, taken from some total order, placed at

the leaves of T, so that each leaf contains at most one item. For simplicity, we assume that the items are distinct. We wish to compute for each internal node $v \in T$ the sorted list $U(v)$ that consists of all the items stored in descendant nodes of v. (See Fig. 1.) In this section we show how to constrict $U(v)$ for every node in the tree in $O(height\ (T))$ time using $|T|$ processors, where $|T|$ denotes the number of nodes in T. This is a generalization of the problem studied by Cole [13], because in his version the tree T is complete. Without loss of generality, we assume that every internal node v of T has two children. For if v has only one child then we can add a child to v (a leaf node) that does not store any items from the total order. Such an augmentation will at most double the size of T and does not change its height.

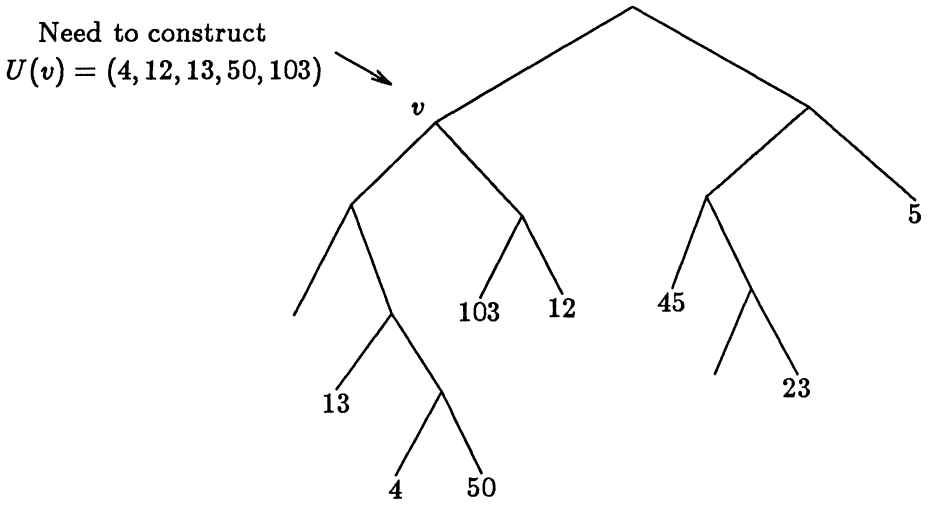


FIG. 1. *An example of the generalized merge problem.*

Let a, b, and c be three items, with $a \leqq b$. We say c is *between* a and b if $a < c \leqq b$. Let two sorted (nondecreasing) lists $A = (a_1, a_2, \cdots, a_n)$ and $B = (b_1, b_2, \cdots, b_m)$ be given. Given an element a, we define the *predecessor* of a in B to be the greatest element in B that is less than or equal to a. If $a < b_1$, then we say that the predecessor of a is $-\infty$. We define the *rank* of a in B to be the rank of the predecessor of a in B ($-\infty$ has rank zero). We say that A is *ranked* in B if for every element in A we know its rank in B. We say that A and B are *cross-ranked* if A is ranked in B and B is ranked in A. We define two operations on sorted lists. We define $A \cup B$ to be the sorted *merged* list of all elements in A or B. If B is a subset of A, then we define $A - B$ to be the sorted list of the elemented in A that are not in B.

Let T be a binary tree. For any node v in T we let $parent(v)$, $sibling(v)$, $lchild\ (v)$, $rchild\ (v)$, and $depth(v)$ denote the parent of v, the sibling of v, the left child of v, the right child of v, and the depth of v (the root is at depth zero), respectively. We also let $root(T)$ and $height(T)$ denote the root node of T and the height of T, respectively. The *altitude*, denoted $alt(v)$, is defined $alt(v) = height(T) - depth(v)$. $Desc(v)$ denotes the set of descendant nodes of v (including v itself).

Let a sorted list L and a sorted list J be given. Using the terminology of Cole [13], we say that L is a *c-cover* of J if between each two adjacent items in $(-\infty, L, \infty)$ there are at most c items from J (where $(-\infty, L, \infty)$ denotes the list consisting of $-\infty$, followed by the elements of L, followed by ∞). We let $SAMP_c(L)$ denote the sorted

list consisting of every cth element of L, and call this list the *c-sample* of L. That is, $SAMP_c(L)$ consists of the cth element of L followed by the $(2c)$th element of L, and so on.

The algorithm for constructing $U(v)$ for each $v \in T$ proceeds in stages. Intuitively, in each stage we will be performing a portion of the merge of $U(lchild\ (v))$ and $U(rchild\ (v))$ to give the list $U(v)$. After performing a portion of this merge we will gain some insight into how to perform the merge at v's parent. Consequently, we will pass some of the elements formed in the merge at v to v's parent, so we can begin performing the merge at v's parent.

Specifically, we denote the list stored at a node v in T at stage s by $U_s(v)$. Initially, $U_0(v)$ is empty for every node except the leaf nodes of T, in which case $U_0(v)$ contains the item stored at the leaf node v (if there is such an item). We say that an internal node v is *active at stage s* if $\lfloor s/3 \rfloor \leqq alt(v) \leqq s$, and we say v is *full at stage s* if $alt(v) = \lfloor s/3 \rfloor$. As will become apparent below, if a node v is full, then $U_s(v) = U(v)$. For each active node $v \in T$ we define the list $U'_{s+1}(v)$ as follows:

$$U'_{s+1}(v) = \begin{cases} SAMP_4(U_s(v)) & \text{if } alt(v) \geqq s/3, \\ SAMP_2(U_s(v)) & \text{if } alt(v) = (s-1)/3, \\ SAMP_1(U_s(v)) & \text{if } alt(v) = (s-2)/3. \end{cases}$$

At stage $s+1$ we perform the following computation at each internal node v that is currently active.

Per-stage computation $(v, s+1)$. Form the two lists $U'_{s+1}(lchild\ (v))$ and $U'_{s+1}(rchild\ (v))$, and compute the new list

$$U_{s+1}(v) := U'_{s+1}(lchild\ (v)) \cup U'_{s+1}(rchild\ (v)).$$

This formalizes the notion that we pass information from the merges performed at the children of v in stage s to the merge being performed at v in stage $s+1$. Note that until v becomes full, $U'_{s+1}(v)$ will be the list consisting of every fourth element of $U_s(v)$. This continues to be true about $U'_{s+1}(v)$ up to the point that v becomes full. If s_v is the stage at which v becomes full (and $U_s(v) = U(v)$), then at stage s_v+1, $U'_{s+1}(v)$ is the two-sample of $U_s(v)$, and, at stage s_v+2, $U'_{s+1}(v) = U_s(v)$ $(= U(v))$. Thus, at stage s_v+3, $parent(v)$ is full. Therefore, after $3 * height(T)$ stages every node has become full and the algorithm terminates. We have yet to show how to perform each stage in $O(1)$ time using n processors.

We begin by showing that the number of items in $U_{s+1}(v)$ can be only a little more than twice the number of items in $U_s(v)$, a property that is essential to the construction.

LEMMA 2.1. *For any stage* $s \geqq 0$ *and any node* $v \in T$, $|U_{s+1}(v)| \leqq 2|U_s(v)| + 4$.

Proof. The proof is by induction on s.

Basis $(s = 0)$. The claim is clearly true for $s = 0$.

Induction step $(s > 0)$. Assume the claim is true for stage $s-1$. If v is full (i.e., $alt(v) = \lfloor s/3 \rfloor$), then the claim is obviously true, since $U_{s+1}(v) = U_s(v) = U(v)$. Consider the case where either the children of v were not full at stage s or had just become full at stage s. We know that $U_{s+1}(v) = U'_{s+1}(x) \cup U'_{s+1}(y)$, where $x = lchild(v)$ and

$y = rchild\ (v)$. In addition, we have the following:

$$
\begin{aligned}
|U_{s+1}(v)| &= \left\lfloor \frac{|U_s(x)|}{4} \right\rfloor + \left\lfloor \frac{|U_s(y)|}{4} \right\rfloor \quad \text{(from definitions)} \\
&\leqq \left\lfloor \frac{2|U_{s-1}(x)|+4}{4} \right\rfloor + \left\lfloor \frac{2|U_{s-1}(y)|+4}{4} \right\rfloor \quad \text{(by induction hypothesis)} \\
&\leqq 2\left(\left\lfloor \frac{|U_{s-1}(x)|}{4} \right\rfloor + \left\lfloor \frac{|U_{s-1}(y)|}{4} \right\rfloor \right) + 4 \\
&= 2|U_s(v)| + 4.
\end{aligned}
$$

The case when the children of v are full at stage $s-1$ is similar (except that one divides by 2 or 1 instead of 4). Actually, it is simpler, since in this case the children of v were full in stage $s-1$; hence, the step using the induction hypothesis can be replaced by a simple algebraic substitution step. □

In the next lemma we show that the way in which the $U_s(v)$'s grow is "well behaved."

LEMMA 2.2. *Let* $[a, b]$ *be an interval with* $a, b \in (-\infty, U'_s(v), \infty)$. *If* $[a, b]$ *intersects* $k+1$ *items in* $(-\infty, U'_s(v), \infty)$, *then it intersects at most* $8k+8$ *items in* $U_s(v)$ *for all* $k \geqq 1$ *and* $s \geqq 1$.

Proof. The proof is by induction on s. The claim is initially true (for $s = 1$). Actually, for any stage s, if $U'_s(v)$ is empty, then $U_{s-1}(v)$ contains at most three items, hence, $U_s(v)$ contains at most ten elements, by the previous lemma. Also, if $U'_s(v)$ contains one item, then $U_{s-1}(v)$ contains at most seven items, hence, $U_s(v)$ contains at most 18 items, by the previous lemma. At most 15 of these items can be between any two adjacent items in $(-\infty, U'_s(v), \infty)$, since the item in $U'_s(v)$ was the fourth item in $U_{s-1}(v)$ by definition.

Inductive step (assume true for stage s). Let $[a, b]$ be an interval with a, b both in the list $(-\infty, U'_{s+1}(v), \infty)$, and suppose $[a, b]$ intersects $k+1$ items in $(-\infty, U'_{s+1}(v), \infty)$. The lemma is immediately true if v was full stage s, since the smallest sample we take is a four-sample. So, next, suppose that either the children of v are not full or have just become full in stage s. Let g be the number of items in $(-\infty, U_s(v), \infty)$ intersected by $[a, b]$. Recall that $U_s(v) = U'_s(lchild\ (v)) \cup U'_s(rchild\ (v))$. Let $[a_1, b_1]$ (respectively, $[a_2, b_2]$) be the smallest interval containing $[a, b]$ such that $a_1, b_1 \in (-\infty, U'_s(lchild\ (v)), \infty)$ (respectively, $a_2, b_2 \in (-\infty, U'_s(rchild\ (v)), \infty)$). Suppose the interval $[a_1, b_1]$ intersects $h+1$ items in the list $(-\infty, U'_s(lchild\ (v)), \infty)$ and $[a_2, b_2]$ intersects $j+1$ items in $(-\infty, U'_s(rchild\ (v)), \infty)$. Note that $h+j = g$. By the induction hypothesis, $[a_1, b_1]$ intersects at most $8h+8$ items in $U_s(lchild\ (v))$, and hence at most $(8h+8)/4 = 2h+2$ items in $U'_{s+1}(lchild\ (v))$. Likewise, $[a_2, b_2]$ intersects at most $2j+2$ items in $U'_{s+1}(rchild\ (v))$. The definition of $U'_{s+1}(v)$ implies that $g \leqq 4k+1$. Therefore, since $U_{s+1}(v) = U'_{s+1}(lchild\ (v)) \cup U'_{s+1}(rchild\ (v))$, $[a, b]$ intersects at most $(2h+2)+(2j+2)$ items in $U_{s+1}(v)$, where $(2h+2)+(2j+2) \leqq (2h+2)+(2(4k-h+1)+2) \leqq 8k+8$.

The proof for the case when the children of v were full in stage $s-1$ is similar. Actually, it is simpler, since the induction steps can be replaced by algebraic substitution steps in this case. □

COROLLARY 2.3. *The list* $(-\infty, U'_s(v), \infty)$ *is a four-cover for* $U'_{s+1}(v)$, *for all* $s \geqq 0$. □

This corollary is used in showing that we can perform each stage of the merge procedure in $O(1)$ time. In addition to this corollary, we also need to maintain the following rank information at the start of each stage s:

(1) For each item in $U'_s(v)$: its rank in $U'_s(sibling(v))$.

(2) For each item in $U'_s(v)$: its rank in $U_s(v)$ (and hence, implicitly, its rank in $U'_{s+1}(v)$).

The lemma that follows shows that the above information is sufficient to allow us to merge $U'_{s+1}(lchild\ (v))$ and $U'_{s+1}(rchild\ (v))$ into the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors.

LEMMA 2.4 (THE MERGE LEMMA) [13]. *Suppose we are given sorted lists A_s, A'_{s+1}, B'_s, B'_{s+1}, C'_s, and C'_{s+1}, where the following (input) conditions are true*:

(1) $A_s = B'_s \cup C'_s$;

(2) A'_{s+1} *is a subset of* A_s;

(3) B'_s *is a* c_1*-cover for* B'_{s+1};

(4) C'_s *is a* c_2*-cover for* C'_{s+1};

(5) B'_s *is ranked in* B'_{s+1};

(6) C'_s *is ranked in* C'_{s+1};

(7) B'_s *and* C'_s *are cross-ranked.*

Then in $O(1)$ time using $|B'_{s+1}|+|C'_{s+1}|$ processors in the CREW PRAM *model, we can compute the following (output computations)*:

(1) *the sorted list* $A_{s+1} = B'_{s+1} \cup C'_{s+1}$;

(2) *the ranking of* A'_{s+1} *in* A_{s+1};

(3) *the cross-ranking of* B'_{s+1} *and* C'_{s+1}. □

We apply this lemma by setting $A_s = U_s(v)$, $A'_{s+1} = U'_{s+1}(v)$, $A_{s+1} = U_{s+1}(v)$, $B'_s = U'_s(x)$, $B'_{s+1} = U'_{s+1}(x)$, $C'_s = U'_s(y)$, and $C'_{s+1} = U'_{s+1}(y)$, where $x = lchild\ (v)$ and $y = rchild\ (v)$. Note that assigning the lists of Lemma 2.4 in this way satisfies input conditions (1)–(4) from the definitions. The ranking information we maintain from stage to stage satisfies input conditions (5)–(7). Thus, in each stage s, we can construct the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors. Also, the new ranking information (of output computations (2) and (3)) gives us the input conditions (5)–(7) for the next stage. By Corollary 2.3 we have that the constants c_1 and c_2 (of input conditions (3) and (4)) are both equal to four. Note that in stage s it is only necessary to store the lists for $s-1$; we can discard any lists for stages previous to that.

The method for performing all these merges with a total of $|T|$ processors is basically to start out with $O(1)$ virtual processors assigned to each leaf node, and each time we pass k elements from a node v to the parent of v (to perform the merge at the parent), we also pass $O(k)$ virtual processors to perform the merge. When v's parent becomes full, then we no longer "store" any processors at v. (See [17] for details.) There can be at most $O(n)$ elements present in active nodes of T for any stage s (where n is the number of leaves of T), since there are n elements present on the full level, at most $n/2$ on the level above that, $n/8$ on the level above that, and so on. Thus, we can perform the entire generalized cascading procedure using $O(n)$ virtual processors, or n actual processors (by a simple simulation argument). This also implies that we need only $O(n)$ storage for this computation, in addition to that used for the output, since once a node v becomes full we can consider the space used for $U(v)$ to be part of the output. Equivalently, if we are using the generalized merging procedure in an algorithm that does not need a $U(v)$ list once v's parent becomes full, then we can implement that algorithm in $O(n)$ space by deallocating the space for a $U(v)$ list once it is no longer needed (this is in fact what we will be doing in § 6).

It will often be more convenient to relax the condition that there be at most one item stored at each leaf. So, suppose there is an unsorted set $A(v)$ (which may be empty) stored at each leaf. In this case we can construct a tree T' from T by replacing each leaf v of T with a complete binary tree with $|A(v)|$ leaves, and associating each item in $A(v)$ with one of these leaves. T' would now satisfy the conditions of the method outlined above. We incorporate this observation in the following theorem, which summarizes the discussion of this section.

THEOREM 2.5. *Suppose we are given a binary tree T such that there is an unsorted set $A(v)$ (which may be empty) stored at each leaf. Then we can compute, for each node $v \in T$, the list $U(v)$, which is the union of all items stored at descendents of v, sorted in an array. This computation can be implemented in $O(\text{height}(T) + \log(\max_v |A(v)|))$ time using a total of $n + N$ processors in the* CREW PRAM *computational model, where n is the number of leaves of T and N is the total number of items stored in T.*

Proof. The complexity bounds follow from the fact that the tree T' described above would have height at most $O(\text{height}(T) + \log(\max_v |A(v)|))$ and $|T'|$ is $O(|T| + N)$. □

The above method comprises one of the main building blocks of the algorithms presented in this paper. We present another important building block in the following section.

3. Fractional cascading in parallel. Given a directed graph $G = (V, E)$, such that every node v contains a sorted list $C(v)$, the fractional cascading problem is to construct an $O(n)$ space data structure that, given a walk $(v_1, v_2, \cdots, v_m)$ in G and an arbitrary element x, enables a single processor to locate x quickly in each $C(v_i)$, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. Fractional cascading problems arise naturally from a number of computational geometry problems. As a simple example of a fractional cascading problem, suppose we have five different English dictionaries and would like to build a data structure that would allow us to look up a word w in all the dictionaries. Chazelle and Guibas [12] give an elegant $O(n)$ time sequential method for constructing a fractional cascading data structure from any graph G, as described above, achieving a search time of $O(\log n + m \log d(G))$, where $d(G)$ is the maximum degree of any node in G. However, their approach does not appear to be "parallelizable."

In this section we show how to construct a data structure achieving the same performance as that of Chazelle and Guibas in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors. Our method begins with a preprocessing step similar to one used by Chazelle and Guibas where we "expand" each node of G into two binary trees—one for its in-edges and one for its out-edges—so that each node in our graph has in-degree and out-degree at most 2. We then perform a cascading merge procedure in stages on this graph. Each catalogue $C(v)$ is "fed into" the node v in samples that double in size with each stage and these lists are in turn sampled and merged along the edges of G. Lists continue to be sampled and "pushed" across the edges of G (even in cycles) for a logarithmic number of stages, at which time we stop the computation and add some links between elements in adjacent lists. We conclude this section by showing that this gives us a fractional cascading data structure, and that the computation can be implemented in $O(\log n)$ time and $O(n)$ space using $\lceil n/\log n \rceil$ processors.

We show below how to perform the computations in $O(\log n)$ time and $O(n)$ space using n processors. We will show later how to get the number of processors down to $\lceil n/\log n \rceil$ by a careful application of Brent's theorem [11].

Define $In(v, G)$ (respectively, $Out(v, G)$) to be the set of all nodes w in V such that $(w, v) \in E$ (respectively, $(v, w) \in E$). The *degree* of a vertex v, denoted $d(v)$, is

defined as $d(v) = \max\{|In(v, G)|, |Out(v, G)|\}$. The *degree* of G, denoted $d(G)$, is defined as $d(G) = \max_{v \in V}\{d(v)\}$. A sequence $(v_1, v_2, \cdots, v_m)$ of vertices is a *walk* if $(v_i, v_{i+1}) \in E$ for all $i \in \{1, 2, \cdots, m-1\}$.

As mentioned above, we begin the construction by preprocessing the directed graph G to convert it into a directed graph $\hat{G} = (\hat{V}, \hat{E})$ such that $d(\hat{G}) \leqq 2$ and such that an edge (v, w) in G corresponds to a path in $\hat{G}$ of length at most $O(\log d(G))$. Specifically, for each node $v \in V$ we construct two complete binary trees T_v^{in} and T_v^{out}. Each leaf in T_v^{in} (respectively, T_v^{out}) corresponds to an edge coming into v (respectively, going out of v). So there are $|In(v, G)|$ leaves in T_v^{in} and $|Out(v, G)|$ leaves in T_v^{out}. (See Fig. 2.) We call T_v^{in} the *fan-in tree* for v and T_v^{out} the *fan-out tree* for v. An edge $e = (v, w)$ in G corresponds to a node e in $\hat{G}$ such that e is a leaf of the fan-out tree for v and e is also a leaf of the fan-in tree for w. The edges in T_v^{in} are all directed up towards the root of T_v^{in}, and the edges in T_v^{out} are all directed down towards the leaves of T_v^{out}. For each $v \in V$ we create a new node v' and add a directed edge from v' to v, a directed edge from the root of T_v^{in} to v', and an edge from v' to the root of T_v^{out}. We call v' the *gateway* for v. (See Fig. 2). Note that $d(\hat{G}) = 2$. We assume that for each node v we have access to the nodes in $In(v, \hat{G})$ as well as those in $Out(v, \hat{G})$. We structure fan-out trees so that a processor needing to go from v to w in $\hat{G}$, with $(v, w) \in E$, can correctly determine the path down T_v^{out} to the leaf corresponding to (v, w). More specifically, the leaves of each fan-out tree are ordered so that they are listed from left to right by increasing destination name, i.e., if the leaf in T_v^{out} for $e = (v, u)$ is to the left of the leaf for $f = (v, w)$, then $u < w$. (The leaves of T_v^{in} need not be sorted, since all edges are directed towards the root of that tree.) If we are not given the $Out(v, G)$ sets in sorted order, then we must perform a sort as a part of the T_v^{out} construction, which can be done in $O(\log d(G))$ time using n processors using Cole's merge sorting algorithm [13]. We also store in each internal node z of T_v^{out} the leaf node u that has the smallest name of all the descendants of z.

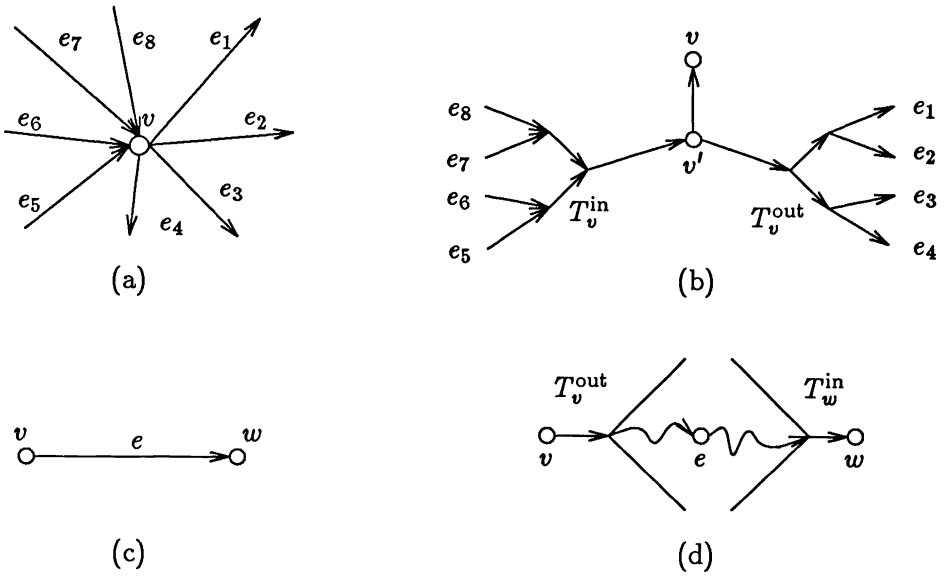


FIG. 2. *Converting G into a bounded degree graph $\hat{G}$. A node v in G* (a) *corresponds into a node v adjacent to its gateway v', which is connected to the fan-in tree and the fan-out tree for v* (b). *An edge e in G* (c) *is converted into a node in $\hat{G}$ which corresponds to a leaf node of two trees* (d).

The above preprocessing step is similar to a preprocessing step used in the sequential fractional cascading algorithm of Chazelle and Guibas [12]. This is where the resemblance to the sequential algorithm ends, however.

The goal for the rest of the computation is to construct a special sorted list $B(v)$, which we call the *bridge list*, for every node $v \in \hat{V}$. We shall define these bridge lists so that $B(v) = C(v)$ if v is in V; if v is in $\hat{V}$ but not in V, then for every $(v, w) \in \hat{E}$, if a single processor knows the position of a search item x in $B(v)$, it can find the position of x in $B(w)$ in $O(1)$ time.

The construction of the $B(v)$'s proceeds in stages. Let $B_s(v)$ denote the bridge list stored at node $v \in \hat{V}$ at the end of stage s. Initially, $B_0(v) = \varnothing$ for all v in $\hat{V}$. Intuitively, the per-stage computation is designed so that if v came from the original graph G (i.e., $v \in V$), then v will be "feeding" $B_s(v)$ with samples of the catalogue $C(v)$ that double in size with each stage. These samples are then cascaded back into the gateway v' for v and from there back through the fan-in tree for v. We will also be merging any samples "passed back" from the fan-out tree for v with $B_s(v')$, and cascading these values back through the fan-in tree for v as well. We iterate the per-stage computation for $\lceil \log N \rceil$ stages, where N is the size of the largest catalogue in G. We will show that after we have completed the last stage, and updated some ranking pointers, $\hat{G}$ will be a fractional cascading data structure for G. The details follow.

Recall that $B_0(v) = \varnothing$ for all $v \in \hat{V}$. For stage $s \geqq 0$, we define $B'_{s+1}(v)$ and $B_{s+1}(v)$ as follows:

$$B'_{s+1}(v) = \begin{cases} SAMP_4(B_s(v)) & \text{if } v \in \hat{V} - V, \\ SAMP_{c(s)}(C(v)) & \text{if } v \in V, \end{cases}$$

$$B_{s+1}(v) = \begin{cases} B'_{s+1}(w_1) \cup B'_{s+1}(w_2) & \text{if } Out(v, \hat{G}) = \{w_1, w_2\}, \\ B'_{s+1}(w) & \text{if } Out(v, \hat{G}) = \{w\}, \\ \varnothing & \text{if } Out(v, \hat{G}) = \varnothing, \end{cases}$$

where $c(s) = 2^{\lceil \log N \rceil - s}$ and N is the size of the largest catalogue. The per-stage computation, then, is as follows.

Per-stage computation $(v, s+1)$. Using the above definitions, construct $B_{s+1}(v)$ for all $v \in \hat{V}$ in parallel (using $|B_{s+1}(v)|$ processors for each v).

The function $c(s)$ is defined so that if $v \in V$, then as the computation proceeds the list $B'_{s+1}(v)$ will be empty for a while. Then at some stage $s+1$, it will consist of a single element of $C(v)$ (the $(2^{\lceil \log N \rceil - s})$th element), in stage $s+2$ at most three elements (evenly sampled), in stage $s+3$ at most five elements, in stage $s+4$ at most nine elements, and so on. This continues until the final stage (stage $\lceil \log N \rceil$), when $B'_{s+1}(v) = C(v)$. Intuitively, the $c(s)$ function is a mechanism for synchronizing the processes of "feeding" the $C(v)$ lists into the nodes of $\hat{G}$ so that all the processes complete at the same time. We show below that each stage can be performed in $O(1)$ time, resulting in a running time of the cascading computations that is $O(\log N)$ (plus the time it takes time to compute the value of N, namely, $O(\log n)$). The following important lemma is similar to Lemma 2.1 in that it guarantees that the bridge lists do not grow "too much" from one stage to another.

LEMMA 3.1. *For any stage* $s \geqq 0$ *and any node* $v \in T$, $|B_{s+1}(v)| \leqq 2|B_s(v)| + 4$.

Proof. The proof is by induction on s.

Basis $(s = 0)$. The claim is clearly true for $s = 0$.

Induction step $(s > 0)$. Assume the claim is true for stage $s-1$. If $v \in V$, then the claim follows immediately from the definition of $c(s)$, since in this case $B_{s+1}(v)$ and

$B_s(v)$ are both samples of $C(v)$ with $B_{s+1}(v)$ being twice as fine as $B_s(v)$, i.e., $|B_{s+1}(v)| \leqq 2|B_s(v)|+1$.

Consider the case when $v \in \hat{V} - V$, and $Out(v, \hat{G}) = \{w_1, w_2\}$. We know in this case $B_{s+1}(v) = B'_{s+1}(w_1) \cup B'_{s+1}(w_2)$. Thus, we have the following:

$$
\begin{aligned}
|B_{s+1}(v)| &= \left\lfloor \frac{|B_s(w_1)|}{4} \right\rfloor + \left\lfloor \frac{|B_s(w_2)|}{4} \right\rfloor \quad \text{(from definitions)} \\
&\leqq \left\lfloor \frac{2|B_{s-1}(w_1)|+4}{4} \right\rfloor + \left\lfloor \frac{2|B_{s-1}(w_2)|+4}{4} \right\rfloor \quad \text{(by induction hypothesis)} \\
&\leqq 2\left(\left\lfloor \frac{|B_{s-1}(w_1)}{4} \right\rfloor + \left\lfloor \frac{|B_{s-1}(w_2)|}{4} \right\rfloor \right) + 4 \\
&= 2|B_s(v)|+4.
\end{aligned}
$$

For the case when $v \in \hat{V} - V$ and $Out(v, \hat{G})$ contains only one node, w, the argument is similar and, in fact, simpler. We simply repeat the above argument, replacing w_1 with w and eliminating those terms that contain w_2. □

In the next lemma we show that the way in which the $B_s(v)$'s grow is "well behaved," much as we did in Lemma 2.2.

LEMMA 3.2. *Let* $[a, b]$ *be an interval with* $a, b \in (-\infty, B'_s(v), \infty)$. *If* $[a, b]$ *intersects* $k+1$ *items in* $(-\infty, B'_s(v), \infty)$, *then it intersects at most* $8k+8$ *items in* $B_s(v)$ *for all* $k \geqq 1$ *and* $s \geqq 1$.

Proof. The proof is structurally the same as that of Lemma 2.2, since that lemma was based on a merge definition similar to that for $B_{s+1}(v)$. □

COROLLARY 3.3. *The list* $(-\infty, B'_s(v), \infty)$ *is a four-cover for* $B'_{s+1}(v)$, *for* $s \geqq 0$.

COROLLARY 3.4. *The list* $(-\infty, B_s(v), \infty)$ *is a* 16*-cover for* $B_s(w)$, *for* $s \geqq 0$ *and* $(v, w) \in \hat{E}$.

The first of these two corollaries implies that we can satisfy all the c-cover input conditions for the Merge Lemma (Lemma 2.4) for performing the merge operations for the computation at stage s in $O(1)$ time using n_s processors, where $n_s = \sum_{v \in \hat{V}} |B_s(v)|$. We use the second corollary to show that when the computation is completed we will have a fractional cascading data structure (after adding the appropriate rank pointers). We maintain the following rank information at the start of each stage s.

(1) For each item in $B'_s(v)$: its rank in $B'_s(w)$ if $In(v, \hat{G}) \cap In(w, \hat{G})$ is nonempty, i.e., if there is a vertex u such that $(u, v) \in \hat{E}$ and $(u, w) \in \hat{E}$.

(2) For each item in $B'_s(v)$: its rank in $B_s(v)$ (and thus, implicitly, its rank in $B'_{s+1}(v)$).

By having this rank information available at the start of each stage s, we satisfy all the ranking input conditions of the Merge Lemma. Thus, we can perform each stage in $O(1)$ time using n_s processors. Moreover, the output computations of the Merge Lemma allow us to maintain all the necessary rank information into the next stage. Note that in stage s it is only necessary to store the lists for $s-1$; we can discard any lists for stages previous to that, as in the generalized cascading merge.

Recall that we perform the computation for $\lceil \log N \rceil$ stages, where N is the size of the largest catalogue. When the computation completes, we take $B(v) = B_s(v)$ for all $v \in \hat{V}$, and for each $(v, w) \in \hat{E}$ we rank $B(v)$ in $B(w)$. We can perform this ranking step by the following method. Assign a processor to each element b in $B(v)$ for all $v \in \hat{V}$ in parallel. The processor for b can find the rank of b in each $B'_s(w)$ such that $w \in Out(v, \hat{G})$ in $O(1)$ time because $B_s(v)$ contains $B'_s(w)$ as a proper subset ($B'_s(w)$ was one of the lists merged to make $B_s(v)$). This processor can then determine the

rank of b in $B(w) = B_s(w)$ for each $w \in Out(v, \hat{G})$ in $O(1)$ time by using the ranking information we maintained (from $B'_s(w)$ to $B_s(w)$) for stage s (rank condition (2) above).

Given a walk $W = (v_1, \cdots, v_m)$, and an arbitrary element x, the query that asks for locating x in every $C(v_i)$ is called the *multilocation* of x in $(v_1, \cdots, v_m)$. To perform a multilocation of x in a walk $(v_1, \cdots, v_m)$, we extend the walk W in G to its corresponding walk $\hat{W} = (\hat{v}_1, \cdots, \hat{v}'_m)$ in $\hat{G}$ and perform the corresponding multilocation in $\hat{G}$, similar to the method given by Chazelle and Guibas [12] for performing multilocations in their data structure. The multilocation begins with the location of x in $B(\hat{v}_1) = B(v'_1)$, the gateway bridge list for v_1, by binary search. For each other vertex in this walk we can locate the position of x in $B(\hat{v}_i)$ given its position in $B(\hat{v}_{i-1})$ in $O(1)$ time. The method is to follow the pointer from x's predecessor in $B(\hat{v}_{i-1})$ to the predecessor of that element in $B(\hat{v}_i)$ and then locate x in $B(\hat{v}_i)$ by a linear search from that position (which will require at most 15 comparisons by Corollary 3.4). In addition, if $\hat{v}_i$ corresponds to a gateway v', then we can locate x in $C(v)$ in $O(1)$ time given its position in $B(v')$ by a similar argument. (See Fig. 3.) Since each edge in the walk W corresponds to a path in $\hat{W}$ of length at most $O(\log d(G))$, this implies that we can perform the multilocation of x in $(v_1, \cdots, v_m)$ in $O(\log |B(v'_1)| + m \log d(G))$ time. In other words, $\hat{G}$ is a fractional cascading data structure. We show that $\hat{G}$ uses $O(n)$ space in the following lemma.

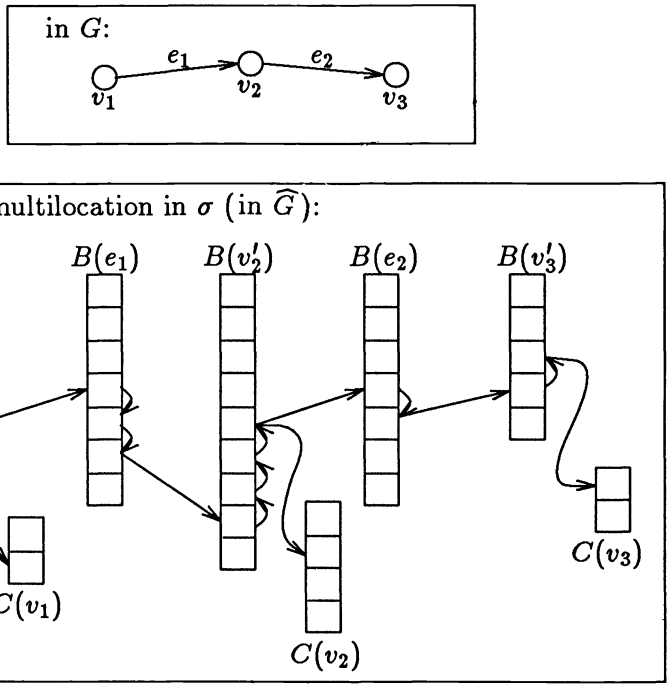


FIG. 3. *Multilocating an element x in* (v_1, v_2, v_3).

LEMMA 3.5. *Let* n_v *denote the amount of space that is added to* $\hat{G}$ *because of the presence of a particular catalogue* $C(v)$, $v \in V$. *Then* $n_v \leqq 2|C(v)|$.

Proof. Recall that while constructing the bridge lists in $\hat{G}$ we copy one-fourth of the elements in each bridge list to at most two of its neighbors. Thus, we have the

following:

$$n_v \leqq |C(v)| + 2\lfloor |C(v)|/4 \rfloor + 2^2 \lfloor |C(v)|/4^2 \rfloor + 2^3 \lfloor |C(v)|/4^3 \rfloor + \cdots$$
$$\leqq 2|C(v)|.$$

(This is obviously an overestimate, but it is good enough for the purposes of the analysis.) □

COROLLARY 3.6. *The total amount of space used by the fractional cascading data structure is* $O(n)$, *where* $n = |V| + |E| + \sum_{v \in V} |C(v)|$.

Proof. The total amount of space used by the fractional cascading data structure is $O(|\hat{V}| + |\hat{E}| + \sum_{v \in \hat{V}} |B(v)|)$. Since all the bridge lists start out empty, $\sum_{v \in \hat{V}} |B(v)| = \sum_{v \in V} n_v$. The previous lemma implies that $\sum_{v \in V} n_v \leqq \sum_{v \in V} 2|C(v)|$. Therefore, since $|\hat{V}| + |\hat{E}|$ is $O(|V| + |E|)$ by the definition of $\hat{G}$, the total amount of space used by the fractional cascading data structure is $O(n)$. □

Note that the upper bound on the space of the fractional cascading data structure holds even if $\hat{G}$ contains cycles. This corollary, then, implies that we can construct a fractional cascading data structure $\hat{G}$ from any catalogue graph G in $O(\log n)$ time and $O(n)$ space using n processors, even if G contains cycles. We have not shown, however, how to assign these n processors to their respective jobs.

The method for performing the processor allocation is as follows. Initially, we assign $2|C(v)|$ virtual processors to each node $v \in V$ and no processors to each node $v \in \hat{V} - V$. This requires at most $2n$ virtual processors; hence, can be easily simulated with n actual processors. Each time we pass k elements from a node v to a node w (in performing the merge at node w) we also pass along (exactly) k virtual processors to go with them. When we say that we are passing a virtual processor from some node v to some node w, all we are actually changing is the node to which that processor is assigned. Since, by Lemma 3.5, $n_v \leqq 2|C(v)|$, we know that there are enough virtual processors assigned to $v \in V$ to do this. To see that this also suffices for $v \in \hat{V} - V$ note that at the beginning of stage s node v has $|B_{s-1}(v)|$ elements (and processors). We "give away" at most $2\lfloor |B_{s-1}(v)|/4 \rfloor$ elements (and processors) from $B_{s-1}(v)$ in stage s and receive $|B_s(v)|$ elements (and processors). Consequently, there are enough processors to perform the merge to construct $B_s(v)$ and repeat the give-away procedure for the next stage. In addition, since we pass a processor for each item we pass to another node, each processor p_i can maintain not only which node it is assigned but p_i can also maintain m_v, the number of other processors that are assigned to that node, as well as maintaining a unique integer identification for itself in the range $[1, m_v]$. Thus, we have the following lemma.

LEMMA 3.7. *Given any catalogue graph* G, *we can construct a fractional cascading data structure for* G *in* $O(\log n)$ *time and* $O(n)$ *space using* n *processors in the* CREW PRAM *model.* □

Thus, we can solve the fractional cascading problem in $O(\log n)$ time using n processors. For the applications we study in this paper, however, we can do even better. The following lemma enumerates two important situations where the method just described can be improved.

LEMMA 3.8. *Given any catalogue graph* G, *if* $d(G)$ *is* $O(1)$ *or if we are given* $Out(v, G)$ *in sorted order for each* $v \in V$, *then the total number of operations performed by the fractional cascading algorithm is* $O(n)$.

Proof. If $d(G)$ is $O(1)$ or we are given $Out(v, G)$ in sorted order, then the construction of the graph $\hat{G}$ (without any bridge lists) requires only $O(n)$ operations, since we do not have to perform any sorting. Let us account for the total work performed

by computing the total number of other operations that are performed because of the fact that the catalogue for each node v contains $|C(v)|$ elements (we will only charge vertices in V). Let s_v be the first stage that $B_s(v')$ becomes nonempty. In this stage $B_s(v')$ receives one element of $C(v)$ from v, and hence we charge one operation in stage s_v for the node v. In stage s_v+1 we will then perform at most 3 operations, at most 7 in stage s_v+2, at most 15 in stage s_v+3, and so on. As soon as $B_s(v')$ contains at least four elements from v (as early as stage s_v+2), then we will perform one more operation, passing one element to the fan-in tree for v. In the next stage, s_v+3, we will perform at most two additional operations, then at most four additional operations in stage s_v+4, and so on. This pattern will "ripple" back through the fan-in tree for v and on through the graph $\hat{G}$ for as long as the computation proceeds. Specifically, the number of operations charged to a node $v \in V$ is, at most, the sum of the following $k_v = \lceil \log_4 |C(v)| \rceil$ rows:

$$
\begin{array}{ccccccccc}
1 & 3 & 7 & 15 & 31 & 63 & 127 & \cdots & |C(v)| \\
 & & 2*1 & 2*3 & 2*7 & 2*15 & 2*31 & \cdots & 2\lfloor |C(v)|/4 \rfloor \\
 & & & & 2^2*1 & 2^2*3 & 2^2*7 & \cdots & 2^2\lfloor |C(v)|/4^2 \rfloor \\
 & & & & & & \vdots & & \vdots \\
 & & & & & & & & 2^{k_v}*1
\end{array}
$$

where the number in row i and column j corresponds to the maximum number of operations performed in stage s_v+j-1 at nodes at distance i from v because of the fact that the catalogue at node v contains $|C(v)|$ elements. (This is actually an overestimate, since not all nodes in $\hat{G}$ have out-degree 2). Summing the number of operations for each row, and then summing the rows, we get that the number of operations charged to $v \in V$ is at most $2(|C(v)|+2\lfloor |C(v)|/4 \rfloor + 2^2\lfloor |C(v)|/4^2 \rfloor + \cdots + 2^{k_v})$, which is at most $4|C(v)|$. Thus, the total number of operations performed by the fractional cascading algorithm is $O(n)$. □

This lemma immediately suggests that we may be able to apply Brent's theorem to the fractional cascading algorithm so that it runs in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors.

THEOREM 3.9 ([11]). *Any synchronous parallel algorithm taking time T that consists of a total of N operations can be simulated by P processors in $O(\lfloor N/P \rfloor + T)$ time.*

Proof of Brent's theorem. Let N_i be the number of operations performed at step i in the parallel algorithm. The P processors can simulate step i of the algorithm in $O(\lceil N_i/P \rceil)$ time. Thus, the total running time is $O(\lfloor N/P \rfloor + T)$:

$$\sum_{i=1}^{T} \lceil N_i/P \rceil \leqq \sum_{i=1}^{T} (\lfloor N_i/P \rfloor + 1) \leqq \lfloor N/P \rfloor + T. \qquad \square$$

There are two qualifications we must make to Brent's theorem before we can apply it in the PRAM model, however. The first is that we must be able to compute N_i at the beginning of step i in $O(\lceil N_i/P \rceil)$ time using P processors. And, second, we must know how to assign each processor to its job. Thus, in order to apply Brent's theorem to our problem of doing fractional cascading, we must deal with these processor allocation problems.

Let $\Gamma = \{p_1, p_2, \cdots, p_m\}$ be the set of virtual processors used in the fractional cascading algorithm (with $m \leqq 2n$), and let $\Gamma' = \{p'_1, p'_2, \cdots, p'_{\lceil n/\log n \rceil}\}$ be the set of processors we will be using to simulate the fractional cascading algorithm. Assuming that $d(G)$ is constant or we are given the list of vertices in $Out(v, G)$ in sorted order, we can compute the graph $\hat{G}$ and the initial assignment of processors from Γ, so that

we assign $2|C(v)|$ virtual processors to each node $v \in V$, in $O(\log n)$ time using the processors in Γ' by a parallel prefix computation. (Recall that the problem of computing all prefix sums $c_k = \sum_{i=1}^{k} a_i$ of a sequence of integers $(a_1, a_2, \cdots, a_n)$ can be done in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors [21], [22].) Let $v(p_i)$ denote the vertex in $\hat{G}$ to which $p_i \in \Gamma$ is assigned. Recall that we will be "passing" the processor p_i around $\hat{G}$ during the computation, so the value of $v(p_i)$ can change from one stage to the next. Once a processor p_i becomes active, it stays active for the remainder of the computation. So, the only thing left to show is how to compute the number of processors active in stage s, and to assign the processors in Γ' to their respective tasks of simulating the processors in Γ. We do this by sorting the set of processors in Γ by the stage in which they become active. It is easy to compute the stage in which a processor p_i becomes active in $O(1)$ time, because this depends only on the initial value of $v(p_i)$ and the size of $C(v(p_i))$ relative to N (the size of the largest catalogue). We can sort the processors in Γ by the stage in which they become active in $O(\log n)$ time using the $\lceil n/\log n \rceil$ processors in Γ', by using an algorithm from Reif [23] (since the stage numbers fall in the range $[1, \lceil \log N \rceil]$). Thus, by performing a parallel prefix computation on this ordered list of processors, we can determine the number of processors active in each stage s, and also know how to assign the processors in Γ' so that they optimally simulate the activities of the processors in Γ during stage s. We thus have established the following theorem.

THEOREM 3.10. *Given a catalogue graph $G = (V, E)$, such that $d(G)$ is $O(1)$ or given each Out(v, G) set in sorted order, we can build a fractional cascading data structure for G in $O(\log n)$ time and $O(n)$ space using $\lceil n/\log n \rceil$ processors in the* CREW PRAM *model, where $n = |V| + |E| + \sum_{v \in V} |C(v)|$. This bound is optimal.* □

4. The plane-sweep tree data structure. In this section we define a data structure, which we call the plane-sweep tree, and show how to use it and the fractional cascading procedure of the previous section to solve the trapezoidal decomposition problem and the planar-point location problem in $O(\log n)$ time using n processors. Since the construction of this data structure is quite involved, we merely define the data structure now, and show how to construct it in these same bounds in § 5.

Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of nonintersecting line segments in the plane, and let $X(S) = (\alpha_1, \alpha_2, \cdots, \alpha_{2n})$ be the (nondecreasing) sorted list of the x-coordinates of the endpoints of the segments in S. To simplify the exposition, we assume that no two endpoints in S have the same x-coordinate, i.e., $\alpha_i < \alpha_{i+1}$. Let $X' = (x_1, x_2, \cdots, x_m)$ be some subsequence of $X(S)$ and let T be the complete binary tree whose $m+1$ leaves, in left to right order, correspond to the intervals $(-\infty, x_1]$, $[x_1, x_2]$, $[x_2, x_3]$, $\cdots$, $[x_{m-1}, x_m]$, $[x_m, +\infty)$, respectively. Associated with each internal node $v \in T$ is the interval I_v which is the union of the intervals associated with the descendants of v. Let Π_v denote the vertical strip $I_v \times (-\infty, +\infty)$. We say a segment s_i *covers a node* $v \in T$ if it spans Π_v but not $\Pi_{parent(v)}$. No segment covers more than two nodes of any level of T; hence, every segment covers at most $O(\log m)$ nodes of T. For each node $v \in T$ we let $Cover(v)$ denote the set of all segments in S that cover v.

The idea of using a tree data structure such as this to parallelize plane-sweeping is due to Aggarwal et al. [1] and is itself based on the "segment tree" of Bentley and Wood [8]. The data structure of Aggarwal et al. consists of the tree T described above with $X' = X(S)$ (i.e., it has $2n+1$ leaves). Aggarwal et al. store the list $Cover(v)$ at each node v sorted by the "above" relation for line segments. They construct these lists by first collecting the segments in each $Cover(v)$ and then sorting all the $Cover(v)$'s in parallel, an operation that requires $\Theta(\log^2 n)$ time using n processors [13], since

there are a total $\Theta(n \log n)$ items to sort. Once these lists are constructed the data structure can then be used to solve various problems by performing certain searches on the nodes of T. These searches are of the following nature: given a set of $O(n)$ input points, for each point p locate the segment in $Cover(v)$ that is directly above (or below) p, for all $v \in T$ such that $p \in \Pi_v$. Notice that for the leaf-to-root walk starting with the leaf v such that $p \in \Pi_v$, this search can be solved by the multilocation of p in that walk. Aggarwal et al. [1] perform all $O(n)$ multilocations in $O(\log^2 n)$ time using n processors by assigning a processor to each point p and doing a binary search for p in all the $Cover(v)$ lists such that $p \in \Pi_v$ (there are $O(\log n)$ such lists for each p).

Although based on the structure of Aggarwal et al., the plane-sweep tree differs from it in some important ways. One such difference is that the plane-sweep tree allows us to perform $O(n)$ multilocations in $O(\log n)$ time using n processors, after a preprocessing step that takes $O(\log n)$ time using n processors. Also, instead of taking X' to be the entire $X(S)$ list, we define X' to be the list consisting of every $\lceil \log n \rceil$th element of $X(S)$, i.e., $X' = SAMP_{\lceil \log n \rceil}(X(S))$. Thus, each vertical strip Π_v associated with a leaf of T in our construction contains $O(\log n)$ segment endpoints. Like Aggarwal et al., we also store each $Cover(v)$ list sorted by the "above" relation. In addition, for every node v of T we define the set $End(v)$ as follows:

$$End(v) = \{s_i | s_i \in S, \text{ has an endpoint in } \Pi_v, \text{ and does not span } \Pi_v\}.$$

Although $End(v)$ is defined for each node of T we only construct a copy of $End(v)$ if v is a leaf node. We do not store the elements of any $End(v)$ in any particular order. This is due to the fact that $End(v)$ contains $O(\log n)$ segments for any leaf node; hence a single processor can search the entire list in $O(\log n)$ time.

Note that all the segments in the $Cover(v)$'s of any root-to-leaf path in T are comparable by the "above" relation. Thus, if we direct all the edges in T so that each edge goes from a child to its parent, then the elements stored in any directed walk in T are all comparable by the "above" relationship. Therefore, we can apply the fractional cascading technique of the previous section to T (with each $Cover(v)$ playing the role of the catalogue $C(v)$). Since T has bounded degree and has $O(n \log n)$ space, we can, by Theorem 3.10, construct a fractional cascading data structure $\hat{T}$ for T in $O(\log n)$ time and $O(n \log n)$ space using n processors. This data structure allows us to perform the multilocation of any point p (in a leaf-to-root walk) in $O(\log n)$ time ($O(\log n)$ for the binary search at the leaf, and an additional $O(1)$ for each internal node on the path to the root). We also store the set $End(v)$ in each leaf v of $\hat{T}$. The plane-sweep tree data structure, then, consists of the tree $\hat{T}$ constructed from T by fractional cascading, where T is defined with $X' = SAMP_{\lceil \log n \rceil}(X(S))$, has $Cover(v)$ stored in sorted order for every node $v \in T$, and the set $End(v)$ stored (unsorted) for each leaf node $v \in T$ (see Fig. 4).

In § 5 we show how to construct this data structure efficiently in parallel. Since the construction is rather involved, before giving the details of the construction, we give two applications of this data structure. We begin with the trapezoidal decomposition problem.

4.1. The trapezoidal decomposition problem. Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of nonintersecting line segments in the plane. For any endpoint p of a segment in S a *trapezoidal segment* for p is a segment of S that is directly above or below p such that the vertical line segment from p to this edge is not intersected by any other segment in S. The trapezoidal decomposition problem is to find the trapezoidal segment(s) for each endpoint of the segments in S. Even in the parallel setting, this problem is often

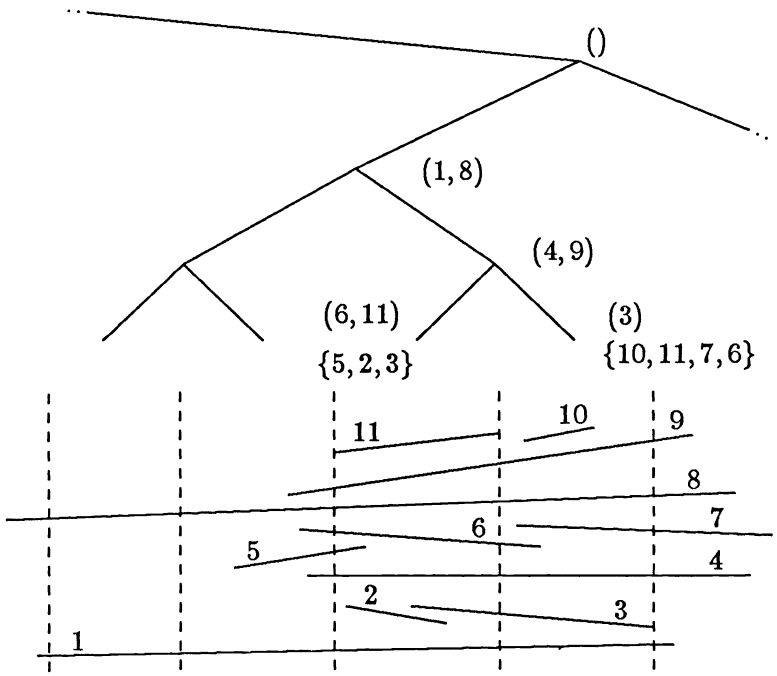


FIG. 4. *A portion of a plane-sweep tree. The segments are numbered in this example by embedding the "above" relation of* § 2 *in the total order* $1, 2, \cdots, 11$. *For simplicity we denote the list* $Cover(v)$ *by parentheses and the set* $End(v)$ *by set braces.*

used as a building block to solve other problems, such as polygon triangulation [1], [19], [28] or shortest paths in a polygon [16].

THEOREM 4.1. *A trapezoidal decomposition of a set* S *of* n *nonintersecting segments in the plane can be constructed in* $O(\log n)$ *time using* n *processors in the* CREW PRAM *model, and this is optimal.*

Proof. Construct the plane-sweep tree data structure T for S. Theorem 5.2 (to be given later, in § 5) shows that this structure can be constructed in $O(\log n)$ time using n processors. And we already know that T can be made into a fractional cascading data structure $\hat{T}$ in these same bounds. We assign a single processor to every segment endpoint (there are $2n$ such points). Let us concentrate on computing the trapezoidal segment below a single segment endpoint p. Let $(v, \cdots, root(T))$ be the leaf-to-root path in $\hat{T}$ that starts with the leaf v such that $p \in \Pi_v$. We first search through $End(v)$ to see if there are any segments in this set that are below p, and take the one that is closest to p (recall that $End(v)$ contains $O(\log n)$ segments). We then perform the multilocation of p in the leaf-to-root walk starting at v, giving us for each w such that $p \in \Pi_w$ the segment in $Cover(w)$ directly below p. We choose among these $\lceil \log n \rceil$ segments the segment that is closest to p. Comparing this segment to the one (possibly) found in $End(v)$, we get the segment in S, if there is one, that is directly below p. Since the length of the walk from v to $root(T)$ is at most $\lceil \log n \rceil$, by the method outlined at the end of § 3 [12], this computation can be done in $O(\log n)$ time using n processors. Since the two-dimensional maxima problem can be reduced to trapezoidal decomposition in $O(1)$ time using n processors [17], and the two-dimensional maxima problem has a sequential lower bound of $\Omega(n \log n)$ in the algebraic computation tree model [7], [20], we cannot do better than $O(\log n)$ time using n processors. □

Solving the trapezoidal decomposition problem efficiently in parallel has proven to be an important step in triangulating a polygon efficiently in parallel [1], [2], [5], [17], [28]. In fact, Theorem 4.1 is used in the algorithms of Goodrich [19] and Yap [28] to achieve an $O(\log n)$ time solution to polygon triangulation using only n processors. We next point out that the plane-sweep tree can also be used to solve the planar point location problem.

4.2. The planar point location problem. The planar point location problem is the following: Given a planar subdivision S consisting of n edges, construct a data structure that, once constructed, enables one processor to determine for a query point p the face in S containing p. This problem has applications in several other parallel computational geometry problems, such as Voronoi diagram construction.

THEOREM 4.2. *Given a planar subdivision S consisting of n edges, we can construct a data structure that can be used to determine for any query point p the face in S containing p in $O(\log n)$ serial time. This construction takes $O(\log n)$ time using n processors in the* CREW PRAM *model.*

Proof. The solution to this problem is to build the plane-sweep tree data structure for S (with fractional cascading) and associate with each edge s_i the name of the face above s_i. As already mentioned, Theorem 5.2 (to be given later, in § 5) shows that the tree T can be constructed in $O(\log n)$ time using n processors. Also recall that T can be made a fractional cascading data structure $\hat{T}$ in these bounds. Let a query point p be given. A planar point location query for p can be solved in $O(\log n)$ serial time by performing a multilocation like that used in the proof of Theorem 4.1 to find the segment in S directly below p. After we have determined the segment s_i in S that is directly below p, we then can read off the face of S containing p by looking up which face is directly above s_i. □

Incidentally, Theorem 4.2 immediately implies that the running time of the Voronoi diagram algorithm of Aggarwal et al. [1] can be improved from $O(\log^3 n)$ to $O(\log^2 n)$, still using only n processors. (We have recently learned that in the final version of their paper [2], they reduce the time bound of their algorithm to $O(\log^2 n)$ using a substantially different technique.)

The results of §§ 4.1 and 4.2 are conditional: they hold if we can construct the plane-sweep tree data structure efficiently in parallel. We next show how to construct the plane-sweep tree in $O(\log n)$ time using only n processors.

5. Cascading with line segment partial orders. In this section we show how to modify the cascading divide-and-conquer technique of § 2 to solve some geometric problems in which the elements being merged belong to the partial order defined by a set of nonintersecting line segments. Recall that in this partial order a segment s_1 is "above" a segment s_2 if there is a vertical line that intersects both segments, and its intersection with s_1 is above its intersection with s_2. We apply this technique to the problems of constructing the plane-sweep tree data structure and of detecting if any two of n segments in the plane intersect.

We now give a brief overview of the problems encountered and our solutions to them. The essential computation is as follows: we have a binary tree with lists stored in its leaves, and we wish to combine them in pairs (up the tree) to construct lists at internal nodes. The main difficulty is that the list stored at some node v is not defined as a simple merge of the lists stored at the children of v. Instead, its definition involves deleting elements from lists stored at children nodes before performing a merge. These deletions are quite troublesome, because if we try to perform these deletions while cascading, then the rank information will become corrupted, and the cascade will fail.

On the other hand, if we try to postpone the deletions to some postprocessing step, then there will be nondeleted elements that are not comparable to others at the same node; hence, there will be instances when processors try to compare two elements that are not comparable, and the cascade will fail. The main idea of our method for getting around these problems is to embed partial orders in total orders "on the fly" while we are cascading up the tree. That is, we change the identity of segments as they are being passed up the tree, so that the segments in any list are always linearly ordered. To be able to do this, however, we must do some preprocessing that involves simultaneously performing a number of cascading merges in parallel. We complete the computation by performing a purging postprocessing step to remove the segments that "changed identity" (as an alternative to being deleted).

For the intersection detection problem, we need to dovetail the detection of intersections with the cascading. That is, we cascade the results of intersection checks along with the segments being passed up the tree. The complication here is that if we should ever detect an intersection on the way up the tree we cannot stop and answer "yes" as this would require $O(\log n)$ time (to "fan-in" all the possible answers). Thus we are forced to proceed with the merging until we reach the root, even though in the case of an intersection the segments being merged no longer even belong to a partial order. We show that in this case we can replace the segment with a special place holder symbol so that the cascades can proceed. After the cascading merge completes we perform some postprocessing to then check if any intersections are present.

The next two subsections give the details.

5.1. Plane-sweep tree construction. In this subsection we describe how to construct the $Cover(v)$ lists for each node v in the plane-sweep tree T. We begin by making a few definitions and observations. We let left (Π_v) (respectively, right (Π_v)) denote the left (right) vertical boundary line for Π_v. We define the *dominator node* of a segment s_i, denoted $dom(s_i)$, to be the deepest node v (i.e., farthest from the root) in T such that s_i is completely contained in Π_v. That is, the dominator of s_i is the node v such that s_i does not intersect left (Π_v) or right (Π_v), but s_i does intersect the vertical boundary separating $\Pi_{lchild(v)}$ and $\Pi_{rchild(v)}$. In addition, we define the following sets for each node $v \in T$:

$$L(v) = \{s_i | s_i \in End(v) \text{ and } s_i \cap \text{left } (\Pi_v) \neq \emptyset\},$$

$$R(v) = \{s_i | s_i \in End(v) \text{ and } s_i \cap \text{right } (\Pi_v) \neq \emptyset\},$$

$$l(v, d) = \{s_i | s_i \in L(v) \text{ and } d = depth(dom(s_i))\},$$

$$r(v, d) = \{s_i | s_i \in R(v) \text{ and } d = depth(dom(s_i))\}.$$

Note that $l(v, d)$ and $r(v, d)$ are only defined for $0 \leqq d < depth(v)$. Any time we construct one of these sets it will be ordered by the "above" relation, so for the remainder of this section we represent these sets as sorted lists. In the following lemma we make some observations concerning the relationships between the various lists defined above.

LEMMA 5.1. *Let v be a node in T with left child x and right child y. Then we have the following (see Fig. 5):*

(1) $l(v, d) = l(x, d) \cup l(y, d)$ *for* $d < depth(v)$,
(2) $r(v, d) = r(x, d) \cup r(y, d)$ *for* $d < depth(v)$,
(3) $L(v) = l(v, 0) \cup l(v, 1) \cup \cdots \cup l(v, depth(v) - 1)$,
(4) $R(v) = r(v, 0) \cup r(v, 1) \cup \cdots \cup r(v, depth(v) - 1)$,
(5) $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$,

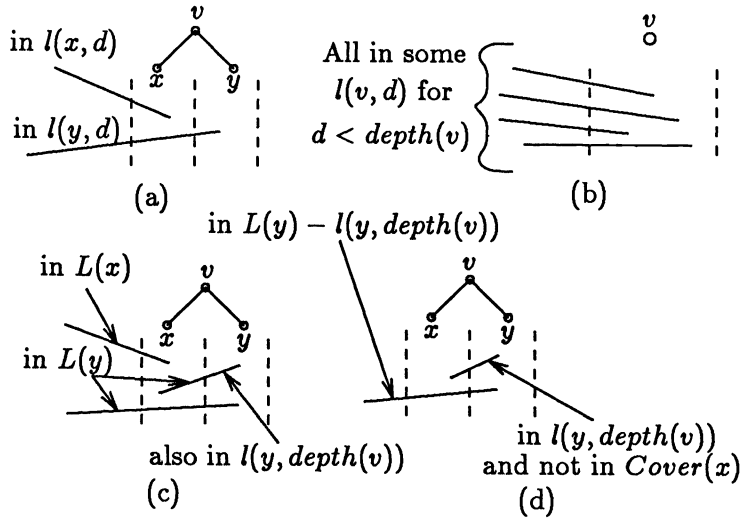


FIG. 5. *The plane-sweep tree equations.* (a) $l(v,d) = l(x,d) \cup l(y,d)$; (b) $L(v) = l(v,0) \cup l(v, depth(v)-1)$; (c) $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$; (d) $Cover(x) = L(y) - l(y, depth(v))$.

(6) $R(v) = (R(x) - r(x, depth(v))) \cup R(y)$,

(7) $Cover(x) = L(y) - l(y, depth(v))$,

(8) $Cover(y) = R(x) - r(x, depth(v))$.

Proof. The proof follows from the definitions. □

Lemma 5.1 essentially states that the lists l, r, L, R, and *Cover* for the nodes on a particular level of T can be defined in terms of lists for nodes on the next lower level of T. We could use this lemma and the parallel merge technique of Valiant [26], as implemented by Borodin and Hopcroft [10], to construct a sorted copy of each $Cover(v)$ list in $O(\log n \log\log n)$ time using n processors, improving on the previous bound of $O(\log^2 n)$ time using the same number of processors, due to Aggarwal et al. [1]. We can do even better, however, by exploiting the structure of the L and R lists. We describe how to do this below, in order to achieve a running time of $O(\log n)$ still using n processors. Before going into the details of the plane-sweep tree construction, we give a brief overview of the algorithm.

HIGH-LEVEL DESCRIPTION OF PLANE-SWEEP TREE CONSTRUCTION.

The construction consists of the following four steps:

Step 1. Construct $l(v,d)$ and $r(v,d)$ for every $v \in T$. To implement this step, we perform $\lceil \log n \rceil$ generalized cascading merges in parallel (one for each d) based on (1) and (2) of Lemma 5.1 (starting with the leaf nodes of T). We implement this step in $O(\log n)$ time using n processors in total for all the merges.

Step 2. Let $d_v = depth(parent(v))$. Compute for each segment in $l(v,d_v)$ (respectively, $r(v,d_v)$) its predecessor segment in $L(v) - l(v,d_v)$ (respectively, $R(v) - r(v,d_v)$) based on (3) and (4). We do this, for each $v \in T$, by making d_v copies of $l(v,d_v)$ and $r(v,d_v)$, and merging $l(v,d_v)$ (respectively, $r(v,d_v)$) with all the $l(v,d)$ (respectively, $r(v,d)$) such that $d < d_v$. Note: we perform this step without actually constructing $L(v)$ or $R(v)$.

Step 3. Construct $L(v)$ and $R(v)$ for every $v \in T$. To implement this step we perform a generalized cascading merge procedure based on (5) and (6) and the information computed in Step 2 (starting with the leaf nodes of T). We never actually perform the set difference operations of (5) and (6), however. Instead, at the point in the merge that a segment in, say, $l(v,d_v)$, should be deleted we "change the identity"

of that segment to its predecessor in $L(v) - l(v, d_v)$ (which we know from Step 2). That is, from this point on in the cascading merge this segment is indistinguishable from its predecessor in $L(v) - l(v, d_v)$. We show below that (i) the cascading merge will not be corrupted by doing this, (ii) the lists never contain too many duplicate entries (that would require us to use more than n processors), and (iii) after the merge completes, we can construct $L(v)$ and $R(v)$ for each node by removing duplicate segments in $O(\log n)$ time using n processors.

Step 4. Construct $Cover(v)$ for every $v \in T$ using (7) and (8) and the lists constructed in Step 3. The implementation of this step amounts to compressing each $L(v)$ (respectively, $R(v)$) so as to delete all the segments in $l(v, d_v)$ (respectively, $r(v, d_v)$), and then copying the list of segments so computed to the sibling node in T.

END OF HIGH-LEVEL DESCRIPTION.

We now describe how to perform each of these high-level steps.

5.2. Step 1: Constructing $l(v, d)$ and $r(v, d)$. We construct the $l(v, d)$ and $r(v, d)$ lists as follows. We make $\lceil \log n \rceil$ copies of T, and let $T(d)$ denote tree number d. Note that by our definition of T the space needed to store the "skeleton" of each $T(d)$ is $O(n/\log n)$. This of course results in a total of $O(n)$ space for all the $T(d)$'s. For each node v of $T(d)$ such that $depth(v) > d$ we wish to construct the lists $l(v, d)$ and $r(v, d)$, as given by (1) and (2) of Lemma 5.1. This implies that if we store $l(v, d)$ (respectively, $r(v, d)$) in every leaf node v of $T(d)$, then for any node $v \in T(d)$, $l(v, d)$ is precisely the sorted merge of the lists stored in the descendants of v. We start with the elements belonging to $l(v, d)$ (respectively, $r(v, d)$) stored (unsorted) in a list $A(v)$ for each leaf v in $T(d)$, and construct each $l(v, d)$ and $r(v, d)$ by the generalized cascading merge technique of Theorem 2.5 (using the $A(v)$'s as in the theorem). Note: since $l(v, d)$ and $r(v, d)$ are only defined for $d < depth(v)$, we only proceed up any tree $T(d)$ as far as nodes at depth $d+1$, terminating the cascading merge at that point. We allocate $\lceil n/\log n \rceil + N_d$ processors to each tree $T(d)$, where N_d denotes the number of segments stored initially in the leaves of $T(d)$. Thus, since $\sum_{d=1}^{\lceil \log n \rceil} N_d = n$, we have shown how to construct all the $l(v, d)$ and $r(v, d)$ lists in $O(\log n)$ time and $O(n \log n)$ space using n processors.

5.3. Step 2: Computing predecessors. In Step 2 we wish to compute for each segment in the list $l(v, d_v)$ (respectively, $r(v, d_v)$) its predecessor segment in $L(v) - l(v, d_v)$ (respectively, $R(v) - r(v, d_v)$), where $d_v = depth(parent(v))$. Without loss of generality, we restrict our attention to the segments in $l(v, d_v)$ (the treatment for the segments in $r(v, d_v)$ is similar). Recall that (3) and (4) state that $L(v) = l(v, 0) \cup l(v, 1) \cup \cdots \cup l(v, d_v)$ and that $R(v) = r(v, 0) \cup r(v, 1) \cup \cdots \cup r(v, d_v)$. We make d_v copies of $l(v, d_v)$ and, using the merging procedure of Shiloach and Vishkin [25] or that of Bilardi and Nicolau [9], we merge a copy of $l(v, d_v)$ with each of $l(v, 0), \cdots, l(v, d_v - 1)$. This takes $O(\log n)$ time using $\lceil |L(v)|/\log n \rceil + \lceil d_v |l(v, d_v)|/\log n \rceil$ processors for each $v \in T$. Since (i) there are $O(n/\log n)$ nodes in each $T(d)$; (ii) each segment appears exactly once in some $l(v, d_v)$; and (iii) $\sum_{v \in T} |L(v)|$ is $O(n \log n)$, we can implement all these merges in parallel using n processors. Once we have completed all the merges, we assign a single processor to each segment s_i and compare the predecessors of s_i in $l(v, 0), \cdots, l(v, d_v - 1)$ so as to find the predecessor of s_i in $L(v) - l(v, d_v)$ $(= l(v, 0) \cup \cdots \cup l(v, d_v - 1))$. This amounts to $O(\log n)$ additional work for each s_i; thus Step 2 can be implemented in $O(\log n)$ time using n processors.

5.4. Step 3: Constructing $L(v)$ and $R(v)$. In this step we perform another cascading merge on T; this time to construct $L(v)$ and $R(v)$ for each $v \in T$ based on (5) and (6)

of Lemma 5.1. Initially, we have $L(v)$ and $R(v)$ constructed only for the leaves. We then merge these lists up the tree based on (5) and (6) as in Theorem 2.5. The computation for this step differs from the cascading merge of Step 2, however, in that we need to be performing set-difference operations as well as list merges as we are cascading up the tree. Unfortunately, it is not clear how to perform these difference operations on-line any faster than $O(\log n)$ time per level, which would result in a running time that is $O(\log^2 n)$. We get around this problem by never actually performing the difference operations. That is, we do not actually delete segments from any lists. Instead, we change the identity of a segment s_i in say, $l(y, d_y)$, to its predecessor in $L(y) - l(y, d_y)$ when we are performing the merge as node v, where $y = rchild\ (v)$ (see Fig. 6). We do this instead of simply marking s_i as "deleted" in $L(v)$, because segments in $l(y, d_y)$ may not be comparable to segments in $L(x)$ (the list with which we wish to merge $L(y) - l(y, d_y)$). Simply marking a segment as being "deleted" could thus result in a processor attempting to compare two incomparable segments.

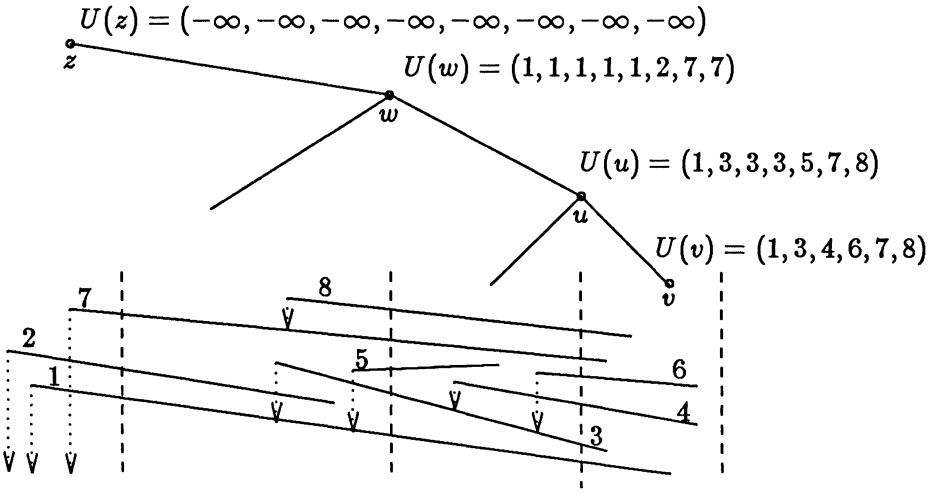


FIG. 6. *Segment identity changing during the cascading merge. We illustrate the way segment names change identity to that of their predecessor as we are performing the cascading merge. In this case we are constructing the $L(v)$'s. We denote the predecessor of each segment by a dotted arrow.*

Clearly, the fact that we change the identity of a segment in $l(y, d_y)$ to its predecessor in $L(y) - l(y, d_y)$ means that there will be multiple copies of some segments. This will not corrupt the cascading merge, however, because one of the properties of the "above" relation for segments is that all duplicate copies of a segment will be contiguous. Moreover, they will remain contiguous as the cascading merge proceeds up the tree. In addition, even though we will have multiple copies of segments in lists as they are merging up the tree, we can still implement this step with a total of n processors, because there will never be more items present in any $L(v)$ than the total number of items stored in the (leaf) descendants of v. At the end of this step we assign $\lceil |L(v)|/\log n \rceil$ processors to each v and compress out the duplicate entries in $L(v)$ in $O(\log n)$ time. Thus, we can construct $L(v)$ and $R(v)$ (compressed and sorted) for each $v \in T$ in $O(\log n)$ time using n processors.

5.5. Step 4: Constructing *Cover*(*v*). In this step we construct $Cover(v)$ for every v in T, based on (7) and (8) of Lemma 5.1. We implement this step by first compressing each $L(v)$ (respectively, $R(v)$) so as to delete all the segments in $l(v, d_v)$ (respectively,

$r(v, d_v)$), and then by copying the list of segments so computed to the sibling of v in T. This can all be done in $O(\log n)$ time using n processors.

Thus, summarizing the entire previous section, we have the following theorem.

THEOREM 5.2. *Given a set S of nonintersecting line segments in the plane, we can construct the plane-sweep tree T for S in $O(\log n)$ time using n processors in the* CREW PRAM *model, and this is optimal.*

Proof. We have already established the correctness and complexity bounds. To see that our construction is optimal, note that the plane-sweep tree requires $\Omega(n \log n)$ space. □

In the previous sections we assumed that segments did not intersect. Indeed, T is defined only if they do not intersect. We show in the next section that we can detect an intersection, if there is one, by constructing T while simultaneously checking for intersections.

5.6. The segment intersection detection problem. The problem we solve in this section is the following: given a set S of n line segments in the plane, determine if any two segments in S intersect. We begin by stating the conditions that we use to test for an intersection.

LEMMA 5.3 [1]. *The segments in S are nonintersecting if and only if we have the following for the plane-sweep tree T of S:*

(1) *For every $v \in T$ all the segments in $Cover(v)$ intersect* left (Π_v) *in the same order as they intersect* right (Π_v).

(2) *For every $v \in T$ no segment in $End(v)$ intersects any segment in $Cover(v)$.* □

Aggarwal et al. [1] used this lemma and their data structure to solve the intersection detection problem in $O(\log^2 n)$ time using n processors. Their method consisted of constructing the $Cover(v)$ lists independently of one another, basing comparisons on segment intersections with left (Π_v), and then testing for condition (1) by checking if each list $Cover(v)$ would be in the same order if they based comparisons on segment intersections with right (Π_v). If no intersection was detected by this step, then they tested for condition (2) by performing $O(n)$ multilocations of segment endpoints. This entire process took $O(\log^2 n)$ time using n processors.

We use this lemma by testing for condition (1) while we are constructing the plane-sweep tree for S (instead of waiting until after it has been built) and in so doing we achieve an $O(\log n)$ time bound for this test (since our construction takes only $O(\log n)$ time). We test condition (2) in the same fashion as Aggarwal et al., that is, by doing $O(n)$ multilocations after the plane-sweep tree has been built. Since with our data structure the multiplications can all be performed in $O(\log n)$ time, the entire intersection-detection process takes $O(\log n)$ time using n processors.

Since we do not construct the $Cover(v)$ lists independently of one another, but instead construct them by performing several cascading merges, we must be very careful in how we base segment comparisons, and in how we test for condition (1). For if two segments intersect, then determining which segment is above the other depends on the vertical line upon which we base the comparison.

We consider each step of the construction in turn, beginning with Step 1. Recall that in Step 1 we construct all the $l(v, d)$ and $r(v, d)$ lists for each $v \in T$. In the following lemma we show that if we base segment comparisons on appropriate vertical lines, Step 1 can be performed just as before.

LEMMA 5.4. *Let $v \in T$ and $0 \leq d < depth(v)$ be given, and let s_1 and s_2 be two segments such that $s_1 \in l(w, d)$ and $s_2 \in l(z, d)$ (or $s_1 \in r(w, d)$ and $s_2 \in r(z, d)$), where $w, z \in Desc(v)$. Then $dom(s_1) = dom(s_2)$.*

Proof. Let $v \in T$ and $0 \leqq d \leqq \lceil \log n \rceil$ be given. Recall that $l(v, d)$ (respectively, $r(v, d)$) is defined to be the list of all segments in $L(v)$ ($R(v)$) that have a dominator node at depth d in T. Note that the dominator node for any segment s_i in $l(w, d)$, $r(w, d)$, $l(z, d)$, or $r(z, d)$, where $w, z \in Desc(v)$, must be an ancestor of v, since $d < depth(v)$ and, by definition, $s_i \in End(v)$ and $s_i \in End(dom(s_i))$. There is only one node that is an ancestor of v and is at depth d in T. □

Thus, we can perform the merges based on (1) and (2) of Lemma 5.1 (e.g., $l(v, d) = l(x, d) \cup l(y, d)$) by basing all segment comparisons on the intersection of the segments with the vertical boundary separating the two children of their dominator node. That is, if s_1 and s_2 are two segments to be compared in Step 1, then we say that s_1 is "above" s_2 if and only if the intersection of s_1 with L is above the intersection of s_2 with L, where L is the vertical boundary line separating the two children of $dom(s_1)$ $(= dom(s_2))$.

In Step 2 we computed for each segment in $l(v, d_v)$ (respectively, $r(v, d_v)$) its predecessor segment in $L(v) - l(v, d_v)$ (respectively, $R(v) - r(v, d_v)$), where $d_v = depth(parent(v))$. Recall that we did this by merging $l(v, d_v)$ with each of $l(v, 0), \cdots, l(v, d_v - 1)$. A similar computation was done for $r(v, d_v)$; without loss of generality, we concentrate on the computation involving $l(v, d_v)$. Also recall that all the segments in $l(v, 0), \cdots, l(v, d_v)$ belong to $L(v)$; hence they intersect left (Π_v). After Step 1 finishes, each list $l(v, d)$ will be sorted based on segment intersections with the vertical boundary line separating the two children of the ancestor of v at depth d (the dominator of all the segments in $l(v, d)$). In $O(\log n)$ time we can check if this order is preserved in each of $l(v, 0), \cdots, l(v, d_v)$ if we base segment intersections on left (Π_v), instead. If the order changed in any $l(v, d)$, then we have detected an intersection, and we are done. Otherwise, we proceed with Step 2 just as before, basing comparisons on segment intersections with left (Π_v).

In Step 3 we performed a cascading merge up the tree T, constructing $L(v)$ and $R(v)$ for every node $v \in T$. Recall that this cascading merge was based on (5) and (6) of Lemma 5.1 (e.g., $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$). Let us concentrate on the testing procedure for the $L(v)$'s, since the method for the $R(v)$'s is similar. Initially, let us start with each $L(v)$ constructed at the leaves of T sorted by segment intersections with left (Π_v). Thus, before we perform the merge based on the equation $L(v) = L(x) \cup (L(y) - l(y, depth(v)))$, we must first check to see if the segments in the sample of $L(y) - l(y, depth(v))$ (to be merged with the sample of $L(x)$) have the same order independent of whether comparisons are based on segment intersections with left (Π_y) or left (Π_v). Unfortunately, to do this completely would require $O(\log n)$ time at every level of the tree, resulting in an $O(\log^2 n)$ time algorithm. So, instead of broadcasting at each level whether an intersection has occurred or not, we cascade that information up along with the merges. More precisely, before doing the merge at a node v, we test if every consecutive pair of items in the sample of $L(y) - l(y, depth(v))$ would remain in the same order independent of whether comparisons were based on segment intersections with left (Π_y) or with left (Π_v). If we detect that an intersection has occurred, then we will have two elements that are out of order. If this should occur, we replace both items by the distinguished symbol \$. Then, as the merges continue up the tree, any time we compare an item with \$, we replace that item with \$ and proceed just as before. This keeps the merging process consistent, and after the cascading merge completes we can then in $O(\log n)$ time test if any of the items in any $L(v)$ or $R(v)$ contain a \$ symbol, by assigning $\lceil |L(v)|/\log n \rceil$ processors to each $v \in T$.

In Step 4 we constructed $Cover(v)$ for each $v \in T$. Recall that we did this by simply performing compressing and copying operations on lists constructed in Step 3. Thus,

assuming that no intersection was detected in Step 3, we can perform Step 4 just as before. After Step 4 completes we can assign $\lceil |Cover(v)|/\log n \rceil$ processors to each $v \in T$ and test condition (1) directly in $O(\log n)$ time, checking if the items in $Cover(v)$ would be in the same order independent of whether comparisons were based on left (Π_v) or on right (Π_v).

If we have not discovered an intersection after Step 4, then the only computation left is to perform fractional cascading on the plane-sweep tree T, constructing a fractional cascading data structure $\hat{T}$. In directing all the edges in T to the root, and performing the fractional cascading preprocessing on T to construct $\hat{T}$, we associate a vertical strip with each node in $\hat{T}$. Since T is a tree then $\hat{T}$ is also a tree (recall the preprocessing step of the fractional cascading algorithm). For each node v in $\hat{T}$ if v is also in T, then we take Π_v for v in $\hat{T}$ to be the same as Π_v for v in T. Then, for any v that is in $\hat{T}$ but not in T (i.e., v is a gateway or a node in a fan-in or fan-out tree), we take Π_v to be the union of all the vertical strips that are descendents of v. Every time we perform the per-stage merge computation we compare adjacent entries in each bridge list $B(v)$ to see if they would be in the same order independent of whether we base comparisons on segment intersections with left (Π_v) or right (Π_v). If we detect that two adjacent segments intersect, then we replace both with the special symbol \$. Then, as before, any time we compare a segment with \$ we replace that segment by \$. Finally, when we complete the computation for Step 5, we assign $\lceil |B(v)|/\log n \rceil$ processors to each node v and check if there are any \$ symbols present in any $B(v)$ list.

If there are no intersections detected during the fractional cascading, then we perform $O(n)$ multilocations of all the segment endpoints as in [1] to test condition (2). Let p be an endpoint of some segment s_i. We perform the multilocation of p in the plane-sweep tree for S, and check if s_i intersects the segment directly above p or the segment directly below p in each $Cover(v)$ list such that $p \in \Pi_v$. This test is sufficient, since if s_i intersects any segment in $Cover(v)$, it must intersect the segment directly above p in $Cover(v)$ or the segment directly below p in $Cover(v)$. Thus, by performing a multilocation for p, we can test for condition (2) in $O(\log n)$ time using n processors. We summarize this discussion in the following theorem.

THEOREM 5.5. *Given a set of n line segments in the plane, we can detect if any two intersect in* $O(\log n)$ *time using n processors in the* CREW PRAM *model.* □

So far in this paper we have restricted ourselves to applications involving line segments. In the next section we show how to apply the cascading divide-and-conquer technique to other geometric problems as well.

6. Cascading with labeling functions. In this section we show how to solve several different geometric problems by combining the merging procedure of § 2 with divide-and-conquer strategies based on merging lists with labels defined on their elements. For most of these problems our divide-and-conquer approach gives an efficient sequential alternative to the known sequential algorithms (which use the plane-sweeping paradigm) and gives rise to efficient parallel algorithms as well. We begin with the three-dimensional maxima problem.

6.1. The three-dimensional maxima problem. Let $V = \{p_1, p_2, \cdots, p_n\}$ be a set of points in $\Re^3$. For simplicity, we assume that no two input points have the same x (respectively, y, z) coordinate. We denote the x, y, and z coordinates of a point p by $x(p)$, $y(p)$, and $z(p)$, respectively. We say that a point p_i *one-dominates* another point p_j if $x(p_i) > (p_j)$, *two-dominates* p_j *if* $x(p_i) > x(p_j)$ and $y(p_i) > y(p_j)$, and *three-dominates*

p_j if $x(p_i) > x(p_j)$, $y(p_i) > y(p_j)$, and $z(p_i) > z(p_j)$. A point $p_i \in V$ is said to be a *maximum* if it is not three-dominated by any other point in V. The three-dimensional maxima problem, then, is to compute the set, M, of maxima in V. We show how to solve the three-dimensional maxima problem efficiently in parallel in the following algorithm.

Our method is based on cascading a divide-and-conquer strategy in which the subproblem merging step involves the computation of two labeling functions for each point. The labels we use are motivated by the optimal sequential plane-sweeping algorithm of Kung, Luccio, and Preparata [20]. Specifically, for each point p_i we compute the maximum z-coordinate from among all points that one-dominate p_i and use that label to also compute the maximum z-coordinate from among all points that two-dominate p_i. We can then test if p_i is a maximum point by comparing $z(p_i)$ to this latter label. The details follow.

Without loss of generality, we assume the input points are given sorted by increasing y-coordinates, i.e., $y(p_i) < y(p_{i+1})$, since if they are not given in this order we can sort them in $O(\log n)$ time using n processors [13]. Let T be a complete binary tree with leaf nodes $v_1, v_2, \cdots, v_n$ (in this order). In each leaf node v_i we store the list $B(v_i) = (-\infty, p_i)$, where $-\infty$ is a special symbol such that $x(-\infty) < x(p_j)$ and $y(-\infty) < y(p_j)$ for all points p_j in V. Initializing T in this way can be done in $O(\log n)$ time using n processors. We then perform a generalized cascading merge from the leaves of T as in Theorem 2.5, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). Using the notation of § 2, we let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store two labels: $zod(p_i, v)$ and $ztd(p_i, v)$, where $zod(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that one-dominate p_i, and $ztd(p_i, v)$ is the largest z-coordinate of the points in $U(v)$ that two-dominate p_i. Initially, *zod* and *ztd* labels are only defined for the leaf nodes of T. That is, $zod(p_i, v_i) = ztd(p_i, v_i) = -\infty$ and $zod(-\infty, v_i) = ztd(-\infty, v_i) = z(p_i)$ for all leaf nodes v_i in T (where $U(v_i) = (-\infty, p_i)$). In order to be more explicit in how we refer to various ranks, we let $\text{pred}(p_i, v)$ denote the predecessor of p_i in $U(v)$ (which would be $-\infty$ if the x-coordinates of the points in $U(v)$ are all larger than $x(p_i)$) (see Fig. 7). As we are performing the cascading merge, we update the labels *zod* and *ztd* based on the equations in the following lemma.

LEMMA 6.1. *Let p_i be an element of $U(v)$ and let $u = lchild(v)$ and $w = rchild(v)$. Then we have the following*:

$$zod(p_i, v) = \begin{cases} \max\{zod(p_i, u), zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u), \\ \max\{zod(\text{pred}(p_i, u), u), zod(p_i, w)\} & \text{if } p_i \in U(w), \end{cases} \tag{9}$$

$$ztd(p_i, v) = \begin{cases} \max\{ztd(p_i, u), zod(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u), \\ ztd(p_i, w) & \text{if } p_i \in U(w). \end{cases} \tag{10}$$

Proof. Consider (9). If $p_i \in U(u)$, then every point that one-dominates p_i's predecessor in $U(w)$ also one-dominates p_i, since p_i's predecessor in $U(w)$ is the point with largest x-coordinate less than $x(p_i)$ (or $-\infty$ if every point in $U(w)$ has larger x-coordinate than p_i). Thus $zod(p_i, v)$ is the maximum of $zod(p_i, u)$ and $zod(\text{pred}(p_i, w), w)$ in this case. The case when $p_i \in U(w)$ is similar. Next, consider (10). We know that every point in $U(w)$ has y-coordinate greater than every point in $U(u)$, by our construction of T. Therefore, if $p_i \in U(u)$, then every point in $U(w)$ that one-dominates p_i's predecessor in $U(w)$ must two-dominate p_i. Thus, $ztd(p_i, v)$ is the

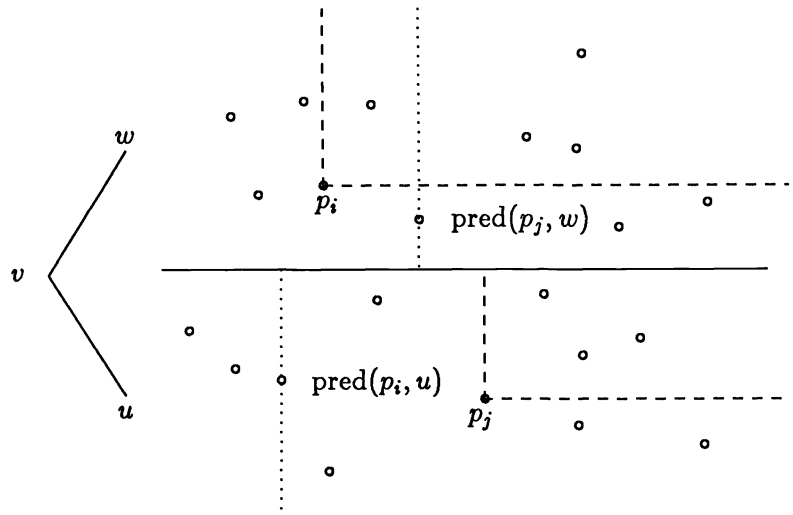


FIG. 7. *The combining step for three-dimensional maxima. Points to the right of the dotted line one-dominate* p_i (*respectively,* p_j), *and points enclosed in the dashed lines two-dominate* p_i(p_j).

maximum of $ztd(p_i, u)$ and $zod(\text{pred}(p_i, w), w)$. On the other hand, if $p_i \in U(w)$ then no point in $U(u)$ can two-dominate p_i; thus, $ztd(p_i, v) = ztd(p_i, w)$. □

We use these equations during the cascading merge to maintain the labels for each point. By Lemma 6.1, when v becomes full (and we have $U(u)$, $U(w)$, and $U(u) \cup U(w)$ available), we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of the cascading merge algorithm, even with these additional label computations, is still $O(\log n)$ using n processors. Moreover, after v's parent becomes full we no longer need $U(v)$, and can deallocate the space it occupies, resulting in an $O(n)$ space algorithm, as outlined in § 2. After we complete the merge, and have computed $U(root(T))$, along with all the labels for the points in $U(root(T))$, note that a point $p_i \in U(root(T))$ is a maximum if and only if $ztd(p_i, root(T)) \leqq z(p_i)$ (there is no point that two-dominates p_i and has z-coordinate greater than $z(p_i)$). Thus, after completing the cascading merge we can construct the set of maxima by compressing all the maximum points into one contiguous list using a simple parallel prefix computation. We summarize in the following theorem.

THEOREM 6.2. *Given a set* V *of* n *points in* $\Re^3$, *we can construct the set* M *of maxima points in* V *in* $O(\log n)$ *time and* $O(n)$ *space using* n *processors in the* CREW PRAM *model, and this is optimal.*

Proof. We have established the correctness and complexity bounds for parallel three-dimensional maxima finding in the discussion above. Kung, Luccio, and Preparata [20] have shown that this problem has an $\Omega(n \log n)$ sequential lower bound (in the comparison model). Thus, we can do no better than $O(\log n)$ time using n processors. □

It is worth noting that we can use roughly the same method as that above as the basis step of a recursive procedure for solving the general k-dimensional maxima problem. The resulting time and space complexities are given in the following theorem. We state the theorem for $k \geqq 3$ (since the two-dimensional maxima problem can easily be solved in $O(\log n)$ time and $O(n)$ space by a sorting step followed by a parallel prefix step).

THEOREM 6.3. *For $k \geqq 3$ the k-dimensional maxima problem can be solved in $O((\log n)^{k-2})$ time using n processors in the* CREW PRAM *model.*

Proof. The method is a straightforward parallelization of the algorithm by Kung, Luccio, and Preparata [20], using a procedure very similar to that described above as the basis for the recursion. We leave the details to the reader. □

Next, we address the two-set dominance counting problem. We also show how the multiple range-counting problem and the rectilinear segment intersection counting problem can be reduced to two-set dominance problems efficiently in parallel.

6.2. The two-set dominance counting problem. In the two-set dominance counting problem we are given a set $A = \{q_1, q_2, \cdots, q_m\}$ and a set $B = \{r_1, r_2, \cdots, r_l\}$ of points in the plane, and wish to know for each point r_i in B the number of points in A that are two-dominated by r_i. For simplicity, we assume that the points have distinct x (respectively, y) coordinates. Our approach to this problem is similar to that of the previous subsection, in that we will be performing a cascading merge procedure while maintaining two labeling functions for each point. In this case the labels maintain for each point p_i(from A or B) how many points of A are one-dominated by p_i and also how many points of A are two-dominated by p_i. As in the previous solution, the first label is used to maintain the second. The details follow.

Let $Y = \{p_1, p_2, \cdots, p_{l+m}\}$ be the union of A and B with the points listed by increasing y-coordinate, i.e., $y(p_i) < y(p_{i+1})$. We can construct Y in $O(\log n)$ time using n processors [13], where $n = l + m$. Our method for solving the two-set dominance counting problem is similar to the method used in the previous subsection. As before, we let T be a complete binary tree with leaf nodes $v_1, v_2, \cdots, v_n$, in this order, and in each leaf node v_i we store the list $U(v_i) = (-\infty, p_i)$ ($-\infty$ still being a special symbol such that $x(-\infty) < x(p_i)$ and $y(-\infty) < y(p_i)$ for all points p_i in Y). We then perform a generalized cascading merge from the leaves of T as in Theorem 2.5, basing comparisons on increasing x-coordinates of the points (not their y-coordinates). We let $U(v)$ denote the sorted array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinate. For each point p_i in $U(v)$ we store two labels: $nod(p_i, v)$ and $ntd(p_i, v)$. The label $nod(p_i, v)$ is the number of points in $U(v)$ that are in A and are one-dominated by p_i, and the label $ntd(p_i, v)$ is the number of points in $U(v)$ that are in A and are two-dominated by p_i. Initially, the *nod* and *ntd* labels are only defined for the leaf nodes of T. That is, $nod(p_i, v_i) = nod(-\infty, v_i) = ntd(p_i, v_i) = ntd(-\infty, v_i) = 0$. For each $p_i \in Y$ we define the function $\chi_A(p_i)$ as follows: $\chi_A(p_i) = 1$ if $p_i \in A$, and $\chi_A(p_i) = 0$ otherwise. (We also use pred (p_i, v) to denote the predecessor of p in $U(v)$. As we are performing the cascading merge, we update the labels *nod* and *ntd* based on the equations in the following lemma (see Fig. 8).

LEMMA 6.4. *Let p_i be an element of $U(v)$ and let $u = lchild\ (v)$ and $w = rchild\ (v)$. Then we have the following:*

$$(11)\quad nod(p_i, v) = \begin{cases} nod(p_i, u) + nod(\text{pred}(p_i, w), w) + \chi_A(\text{pred}(p_i, w)) & \text{if } p_i \in U(u), \\ nod(\text{pred}(p_i, u), u) + nod(p_i, w) + \chi_A(\text{pred}(p_i, u)) & \text{if } p_i \in U(w), \end{cases}$$

$$(12)\quad ntd(p_i, v) = \begin{cases} ntd(p_i, u) & \text{if } p_i \in U(u), \\ nod(\text{pred}(p_i, u), u) + ntd(p_i, w) + \chi_A(\text{pred}(p_i, u)) & \text{if } p_i \in U(w). \end{cases}$$

Proof. Consider (11). For any point $p_i \in U(u)$ the number of points one-dominated by p_i is equal to the number of points in $U(u)$ that are in A and one-dominated by

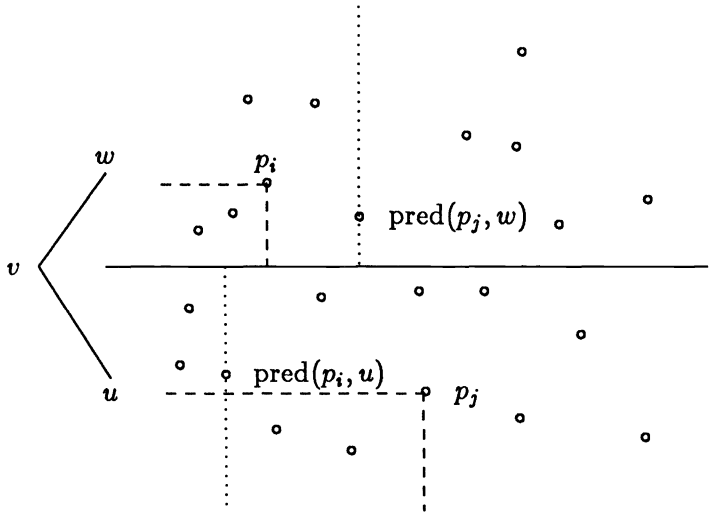


FIG. 8. *The combining step for dominance counting. Points to the left of the dotted line are one-dominated by p_i (respectively, p_j), and points enclosed in dashed lines are two-dominated by p_i (p_j).*

p_i, plus the number of points in $U(w)$ that are in A and one-dominated by pred(p_i, w), plus one if pred(p_i, w) is in A (since the predecessor of p_i is one-dominated by p_i). Thus, we have the equation for the case when $p_i \in U(u)$. The case when $p_i \in U(w)$ is similar. Next, consider (12). By our construction, every point in $U(u)$ has y-coordinate less than the y-coordinate of every point in $U(w)$. So if $p_i \in U(u)$, then the number of points in $U(v)$ that are in A and are two-dominated by p_i is precisely $ntd(p_i, u)$, since p_i cannot two-dominate any points in $U(w)$. If $p_i \in U(w)$, on the other hand, then the number of points in $U(v)$ that are in A and two-dominated by p_i is the number of points in $U(u)$ that are in A and one-dominated by pred(p_i, u), plus the number of points in $U(w)$ that are in A and two-dominated by p_i, plus one if pred(p_i, u) is in A. This is exactly (12) in this case. □

By Lemma 6.4, when v becomes full (and we have $U(u)$, $U(w)$, and $U(v) = U(u) \cup U(w)$ available), we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of the cascading merge algorithm, even with these additional label computations, is still $O(\log n)$ using n processors. After we complete the merge, and have computed $U(\text{root}(T))$, along with all the labels for the points in $U(\text{root}(T))$, then we are done. We summarize in the following theorem.

THEOREM 6.5. *Given a set A of l points in the plane and a set B of m points in the plane, we can compute for each point p in B the number of points in A two-dominated by p in $O(\log n)$ time and $O(n)$ space using n processors in the* CREW PRAM *model, where $n = l + m$, and this is optimal.*

Proof. The correctness and complexity bounds should be apparent from the discussion above. To prove the lower bound note that the two-dimensional maxima problem can be reduced to dominance counting in $O(1)$ time using n processors (see [17]). Since the maxima problem has an $\Omega(n \log n)$ lower bound [20] in the comparison model, we conclude that we can do no better than $O(\log n)$ time using n processors in the CREW PRAM model. □

There are a number of other problems that can be reduced to two-set dominance counting. We mention two here, the first being the multiple range-counting problem:

given a set V of l points in the plane and a set R of m isothetic rectangles (ranges) the multiple range-counting problem is to compute the number of points interior to each rectangle.

COROLLARY 6.6. *Given a set V of l points in the plane and a set R of m isothetic rectangles, we can solve the multiple range-counting problem for V and R in $O(\log n)$ time and $O(n)$ space using n processors, where $n = l + m$.*

Proof. Let $d(p)$ be the number of points in V two-dominated by a point p. Edelsbrunner and Overmars [15] have shown that counting the number of points interior to a rectangle can be reduced to dominance counting. That is, given a rectangle $r = (p_1, p_2, p_3, p_4)$ (where vertices are listed in counterclockwise order starting with the upper right-hand corner), the number of points in V interior to r is $d(p_1) - d(p_2) + d(p_3) - d(p_4)$. Therefore, it suffices to solve the two-set dominance counting problem. □

Another problem that reduces to two-set dominance counting is rectilinear segment intersection counting: given a set S of n rectilinear line segments in the plane, determine for each segment the number of other segments in S that intersect it.

COROLLARY 6.7. *Given a set S of n rectilinear line segments in the plane, we can determine for each segment the number of other segments in S that intersect it in $O(\log n)$ time and $O(n)$ space using n processors in the* CREW PRAM *model.*

Proof. Let U_1 (U_2) be the set of left (right) endpoints of horizontal segments, and let $d_1(p)$ ($d_2(p)$) denote the number of points in U_1 (U_2) two-dominated by p. For any vertical segment s, with upper endpoint p and lower endpoint q, the number of horizontal segments that intersect s is $d_1(p) - d_1(q) + d_2(q) - d_2(p)$. This is because $d_1(p) - d_1(q)$ (respectively, $d_2(p) - d_2(q)$) counts the number of horizontal segments with a left (right) endpoint to the left of s and y-coordinate in the interval $[y(q), y(p)]$. Thus, $d_1(p) - d_1(q) - (d_2(p) - d_2(q))$ counts the number of horizontal segments with left endpoint to the left of s, right endpoint to the right of s, and y-coordinate in the interval $[y(q), y(p)]$ (i.e., the set of horizontal segments that intersect s). □

The final problem we address at is visibility from a point.

6.3. The visibility from a point problem. Given a set of line segments $S = \{s_1, s_2, \cdots, s_n\}$ in the plane that do not intersect, except possibly at endpoints, and a point p, the visibility from a point problem is to determine the part of the plane that is visible from p assuming every s_i is opaque. Intuitively, we can think of the point p as a specular light source, the segments as walls, and the problem to determine all the parts of the plane that are illuminated. We can use the cascading divide-and-conquer technique to solve this problem in $O(\log n)$ time and $O(n)$ space using n processors. Without loss of generality, we assume that the point p is at negative infinity below all the segments. The algorithm is essentially the same if p is a finite point, except that the notion of segment endpoints being ordered by x-coordinate is replaced by the notion that they are ordered radially around p. In other words, it suffices to compute the *lower envelope* of the n segments to give a method for computing the visibility from a point. For simplicity of expression, we also assume that the x-coordinates of the endpoints are distinct.

In the previous two subsections the set of objects consisted of points, but in the visibility problem we are dealing with line segments. The method is slightly different in this case. In this case, we store the segments in the leaves of a binary tree and perform a cascading merge of the x-coordinates of intervals of the x-axis determined by segment endpoints. We maintain a single label for each interval which represents the segment which is visible from $-\infty$ on that interval. The details follow.

Let T be a complete binary tree with leaf nodes $v_1, v_2, \cdots, v_n$ ordered from left to right. We associate the segment s_i with the leaf v_i and at v_i store the list $U(v_i) = (-\infty, p_1, p_2)$, where p_1 and p_2 are the two endpoints of s_i, with $x(p_1) < x(p_2)$, and $-\infty$ is defined such that $x(-\infty) < x(p)$ and $y(-\infty) < y(p)$ for all points p. We then perform a generalized cascading merge from the leaves of T as in Theorm 2.5, basing comparisons on increasing x-coordinates of the points. For each internal node v we let $U(v)$ denote an array of the points stored in the descendants of $v \in T$ sorted by increasing x-coordinates. For each point p_i in $U(v)$ we store a label $vis(p_i, v)$ which stores the segment with endpoints in $U(v)$ that is visible in the interval $(x(p_i), x(succ(p_i, v)))$, where $succ(p_i, v)$ denotes the successor of p_i in $U(v)$ (based on x-coordinates). Initially, the *vis* labels are only defined for the leaf nodes of T. That is, if $U(v) = (-\infty, p_1, p_2)$, where $s_i = p_1 p_2$, then $vis(-\infty) = +\infty$, $vis(p_1) = s_i$, and $vis(p_2) = +\infty$. We use pred (p_i, v) to denote the predecessor of p_i in $U(v)$. As we are performing the cascading merge, we update the *vis* labels based on the equation in the following lemma (see Fig. 9).

LEMMA 6.8. *Let p_i be an element of $U(v)$ and let $u = lchild\ (v)$ and $w = rchild\ (v)$. Then we have the following (if two segments s_i and s_j are comparable by the "above"*

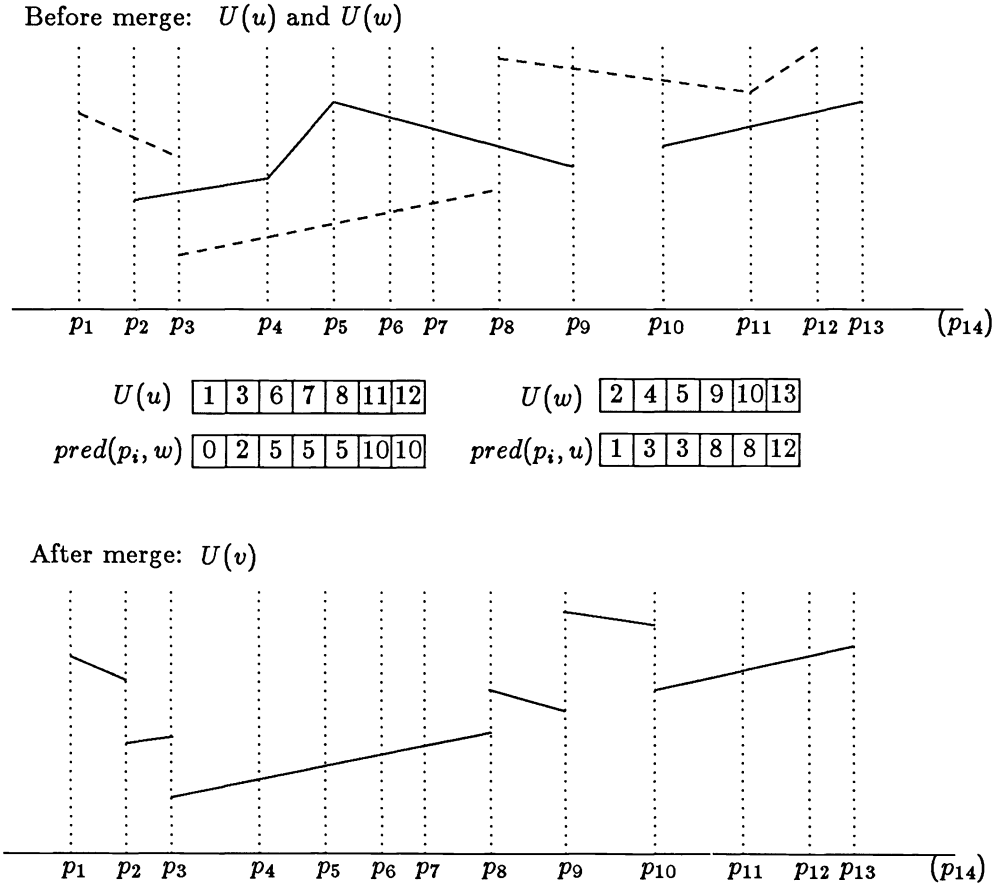


FIG. 9. *An example of visibility merging. The dashed segments correspond to the visible region for $X(u)$ and the solid segments correspond to the visible region for $X(w)$. For simplicity, we store the pointers* pred (p_i, u) *and* pred (p_i, w) *in arrays and denote each point p_i by its index i. Note that points are never removed, even if the same segment defines the visible region for many consecutive intervals (e.g., p_3 through p_7).*

relation, then we let $\min\{s_i, s_j\}$ *denote the lower of the two*):

$$vis(p_i, v) = \begin{cases} \min\{vis(p_i, u), vis(\text{pred}(p_i, w), w)\} & \text{if } p_i \in U(u), \\ \min\{vis(\text{pred}(p_i, u), u), vis(p_i, w)\} & \text{if } p_i \in U(w). \end{cases}$$

Proof. If we restrict our attention to the segments with an endpoint in $U(n)$, then for any point $p_i \in U(u)$ the segment visible (from $-\infty$) on the interval $(x(p_i), x(succ(p_i, v)))$ is the minimum of the segment visible on the interval $(x(p_i), x(succ(p_i, u)))$ and the segment that is visible on the interval $(x(\text{pred}(p_i, w)), x(succ(\text{pred}(p_i, w), w)))$. This is because the interval $(x(p_i), x(succ(p_i, v)))$ is exactly the intersection of the interval $(x(p_i), x(succ(p_i, u)))$ and the interval $(x(\text{pred}(p_i, w)), x(succ(\text{pred}(p_i, w), w)))$, and there is no segment in $U(v)$ with an endpoint interior to the interval $(x(p_i), x(succ(p_i, v)))$. Thus, $vis(p_i, v)$ is equal to minimum of $vis(p_i, u)$ and $vis(\text{pred}(p_i, w), w)$. The case when $p_i \in U(v)$ is similar. □

By Lemma 6.8, after merging the lists $U(u)$ and $U(w)$, we can determine the labels for all the points in $U(v)$ in $O(1)$ additional time using $|U(v)|$ processors. Thus, the running time of this generalized cascading merge algorithm is still $O(\log n)$ using n processors. After we complete the merge and have computed $U(root(T))$, along with all the *vis* labels for the points in $U(root(T))$, then we can compress out duplicate entries in the list $(vis(p_1, root(T)), vis(p_2, root(T)), \cdots, vis(p_{2n}, root(T)))$ using a parallel prefix computation to construct a compact representation of the visible portion of the plane. We summarize in the following theorem.

THEOREM 6.9. *Given a set S of n nonintersecting segments in the plane, we can find the lower envelope of S in* $O(\log n)$ *time and* $O(n)$ *space using n processors in the* CREW PRAM *model, and this is optimal.*

Proof. The correctness and complexity bounds follow from the discussion above. Since we require that the points in the description of the lower envelope be given by increasing x-coordinates, we can reduce sorting to this problem, and thus can do no better than $O(\log n)$ time using n processors. □

7. EREW PRAM implementations. In this section we briefly note that the same techniques as employed by Cole in [13] to implement his merging procedure in the EREW PRAM model (no simultaneous reads) can be applied to our algorithms for generalized merging, fractional cascading, constructing the plane-sweep tree, three-dimensional maxima, two-set dominance counting, and visibility from a point, resulting in EREW PRAM algorithms for these problems. Apparently, we cannot apply his techniques to our algorithms for trapezoidal decomposition and segment intersection detection, however, since our algorithms for these problems explicitly use concurrent reads (in the multilocation steps).

Applying his techniques to our algorithms results in EREW PRAM algorithms with the same asymptotic bounds as the ones presented in this paper, except that the space bounds for the problems addressed in § 6 all become $O(n \log n)$. The reason that his techniques increase the space complexity of these problems is because of our use of labeling functions. Specifically, it is not clear how to perform the merges on-line and still update the labels in $O(1)$ time after a node becomes full. This is because a label whose value changes on level l may have to be broadcasted to many elements in level $l-1$ to update their labels, which would require $\Omega(\log n)$ time in this model if there were $O(n)$ such elements.

We can get around the problem arising from the labeling functions, however. For the three-dimensional maxima problem and the two-set dominance counting problem,

we separate the computation of the $U(v)$ lists and computation of the labeling functions into two separate steps, rather than "dovetailing" the two computations as before. Each of the labeling functions we used for these two problems can be redefined so as to be EREW-computable. Specifically, the label for an element p in $U(v)$, on level l, can be expressed in terms of a label $pref(p, v)$ and a label $up(p, v)$, where $pref(p, v)$ can be computed by performing a parallel prefix computation [21], [22] in $U(v)$ and $up(p, v)$ can be defined in terms of $pref(pred(p, lchild(v)), lchild(v))$, $pref(pred(p, rchild(v)), rchild(v))$, and the up label p had on level $l+1$ (say, in $U(rchild(v))$ if $p \in U(rchild(v))$). In particular, for the three-dimensional maxima problem $pref(p, v) = zod(p, v)$ and $up(p, v) = ztd(p, v)$, and for the two-set dominance counting problem $pref(p, v) = nod(p, v)$ and $up(p, v) = ntd(p, v)$. We can compute all the $pref(p, v)$ labels in $O(\log n)$ time using n processors by assigning $\lceil |U(v)|/\log n \rceil$ processors to each node v [21]. We can then broadcast each $pref(p, v)$ label to the successor of v in $sibling(v)$, which takes $O(\log n)$ time using n processors by assigning $\lceil |U(v)|/\log n \rceil$ processors to each node v. Finally, we can compute all the $up(p, v)$ labels in $O(\log n)$ additional time by assigning a single processor to each point p and tracing the path in the tree from the leaf node that contains p up to the root. This is an EREW operation because computing all the $up(p, v)$ labels only depends upon accessing memory locations associated with the point p.

The EREW solution to the visibility from a point problem requires $O(n \log n)$ space for a different reason, namely, because we can solve it by constructing the plane-sweep tree for the segments (we need not have the $Cover(v)$'s in sorted order, however), computing the lowest segment in each $Cover(v)$, and then performing a top-down parallel *min*-finding computation to find the segment visible on each interval (p_i, p_{i+1}). Since these are all straightforward computations, given the discussion presented earlier in this paper, we leave the details to the reader.

8. Conclusion. In this paper we gave several general techniques for solving problems efficiently using parallel divide-and-conquer. Our techniques are based on nontrivial generalizations of the merge-sorting approach of Cole [13]. It is interesting to note that Cole's algorithm improved the previous results by a constant factor, whereas our algorithms improve the previous results asymptotically.

Two of our techniques involved methods for performing fractional cascading and a generalized version of the merge-sorting problem optimally in parallel. Our method for doing fractional cascading runs in $O(\log n)$ time using $\lceil n/\log n \rceil$ processors, and, if implemented as a sequential algorithm, results in a sequential alternative to the method of Chazelle and Guibas [12] for fractional cascading.

We also showed how to apply the generalized merging procedure and fractional cascading to efficiently solve several problems by "cascading" the divide-and-conquer paradigm. For three of the problems—trapezoidal decomposition, planar point location, and segment intersection detection—the method involved merging in the line segment partial order, and required considerable care to avoid situations in which the algorithm would halt because it attempted to compare two incomparable segments. All three of these algorithms ran in $O(\log n)$ time using n processors, which is optimal for all but the point location problem. In addition, since our algorithm for doing planar point location results in a query time of $O(\log n)$, our result immediately implies an $O(\log^2 n)$ time, n processor solution to the problem of constructing the Voronoi diagram of n planar points, using the algorithm of Aggarwal et al. [1].

We showed how to apply the cascading divide-and-conquer technique to problems that can be solved by merging with labeling functions. We used this approach to solve

the three-dimensional maxima problem, the two-set dominance counting problem, the rectilinear segment intersection counting problem, and the visibility from a point problem. Our algorithms for these problems all ran in $O(\log n)$ time using n processors, which is optimal.

REFERENCES

[1] A. AGGARWAL, B. CHAZELLE, L. GUIBAS, C. Ó'DÚNLAING, AND C. YAP, *Parallel computational geometry*, Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 468-477.

[2] ———, *Parallel computational geometry*, Algorithmica, 3 (1988), pp. 293-328.

[3] M. J. ATALLAH, R. COLE, AND M. T. GOODRICH, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 151-160.

[4] M. J. ATALLAH AND M. T. GOODRICH, *Efficient parallel solutions to some geometric problems*, J. Parallel and Distributed Computing, 3 (1986), pp. 492-507.

[5] ———, *Efficient plane sweeping in parallel*, Proc. 2nd ACM Symposium on Computational Geometry, 1986, pp. 216-225.

[6] ———, *Parallel algorithms for some functions of two convex polygons*, Algorithmica, 3 (1988), pp. 535-548.

[7] M. BEN-OR, *Lower bounds for algebraic computation trees*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 80-86.

[8] J. L. BENTLEY AND D. WOOD, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., C-29 (1980), pp. 571-576.

[9] G. BILARDI AND A. NICOLAU, *Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines*, TR 86-769, Department of Computer Science, Cornell University, August 1986.

[10] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130-145.

[11] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. ACM, 21 (1974), pp. 201-206.

[12] B. CHAZELLE AND L. J. GUIBAS, *Fractional cascading: I. A data structuring technique*, Algorithmica, 1 (1986), pp. 133-162.

[13] R. COLE, *Parallel merge sort*, Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 511-516; SIAM J. Comput., 17 (1988), pp. 770-785.

[14] N. DADOUN AND D. KIRKPATRICK, *Parallel processing for efficient subdivision search*, Proc. 3rd ACM Symposium on Computational Geometry, 1987, pp. 205-214.

[15] H. EDELSBRUNNER AND M. H. OVERMARS, *On the equivalence of some rectangle problems*, Inform. Process. Lett., 14 (1982), pp. 124-127.

[16] H. ELGINDY AND M. T. GOODRICH, *Parallel algorithms for shortest path problems in polygons*, The Visual Computer: Internat. J. Comput. Graphics, 3 (1988), pp. 371-378.

[17] M. T. GOODRICH, *Efficient parallel techniques for computational geometry*, Ph.D. thesis, Department of Computer Science, Purdue University, W. Lafayette, IN, 1987.

[18] ———, *Finding the convex hull of a sorted point set in parallel*, Inform. Process. Lett., 26 (1987), pp. 173-179.

[19] ———, *Triangulating a polygon in parallel*, J. Algorithms, to appear.

[20] H. T. KUNG, F. LUCCIO, AND F. P. PREPARATA, *On finding the maxima of a set of vectors*, J. ACM, 22 (1975), pp. 469-476.

[21] C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *The Power of Parallel Prefix*, Proc. 1985 IEEE Internat. Conference on Parallel Processing, St. Charles, IL, 1985, pp. 180-185.

[22] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. ACM (1980), pp. 831-838.

[23] J. H. REIF, *An optimal parallel algorithm for integer sorting*, Proc. 26th IEEE Symposium on Foundations of Computer Science, 1985, pp. 496-504.

[24] J. H. REIF AND S. SEN, *Optimal Randomized Parallel Algorithms for Computational Geometry*, Proc. 1987 IEEE Internat. Conference on Parallel Processing, 1987, pp. 270-277.

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

[26] L. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348-355.

[27] H. WAGENER, *Optimally parallel algorithms for convex hull determination*, manuscript, 1985.

[28] 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. 18, No. 3, pp. 533-549, June 1989

007

A NEW PEBBLE GAME THAT CHARACTERIZES PARALLEL COMPLEXITY CLASSES*

H. VENKATESWARAN† AND MARTIN TOMPA‡

Abstract. A new two-person pebble game that models parallel computations is defined. This game extends the two-person pebble game defined by Dymond and Tompa [*J. Comput. System Sci.*, 30 (1985), pp. 149-161] and is used to characterize two natural parallel complexity classes, namely LOGCFL and AC^1. The characterizations show a fundamental way in which the computations in these two classes differ. This game model also unifies the proofs of some well-known results of complexity theory.

Key words. pebbling, alternation, parallel complexity, LOGCFL, AC^k

AMS(MOS) subject classifications. 68Q15, 90D05, 68Q25, 94C99

1. Introduction. Games have provided a useful framework for studying computational models. The game abstraction helps us to focus on the fundamental aspects of computation in the models of interest by stripping away some of the inessential details. The view of computations by an alternating Turing machine as a two-person game is one of the most noted examples of such use. The one-person pebble game (see the survey by Pippenger [Pi80]) that models deterministic and nondeterministic evaluation of straight-line programs is another game that has found wide applications in computer science.

This paper defines and studies a new two-person pebble game that models certain synchronous parallel computations. This game extends the two-person pebble game defined by Dymond and Tompa [DT85] in two ways: (a) the game is played on a Boolean circuit taking into consideration the types of the gates, rather than on an uninterpreted graph, and (b) the two players' roles are made completely symmetric.

Although the original game defined by Dymond and Tompa [DT85] modeled some essential features of alternating Turing machine computations, the extension proposed here more accurately models general alternating computations. As an indication of this improved accuracy, the new game is used to characterize two natural parallel complexity classes. Previous versions of pebble games apparently could not be used to characterize standard complexity classes, since they were played on uninterpreted graphs. The classes characterized in this paper are LOGCFL and AC^1. LOGCFL is the class of languages log space reducible to context-free languages. AC^1 is the class of languages accepted by (1) a concurrent-read, concurrent-write parallel random access machine (CRCW PRAM) in polynomial hardware and $O(\log n)$ time or, equivalently, (2) an alternating Turing machine in space $O(\log n)$ and alternation depth $O(\log n)$ or, equivalently, (3) a uniform family of unbounded fan-in circuits of polynomial size and $O(\log n)$ depth [SV84].

As another application, this game is shown to unify in a single framework the proofs of the following three well-known results of complexity theory: (1) Savitch's

* Received by the editors December 1, 1986; accepted for publication (in revised form) August 11, 1988. This material is based on work supported by the National Science Foundation under grants DCR-8301212 and DCR-8352093. A preliminary version of this paper appeared in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, October 1986, © IEEE Computer Society Press.

† Department of Computer Science, FR35, University of Washington, Seattle, Washington 98195. Present address, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India.

‡ IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.

theorem that nondeterministic space S is contained in deterministic space S^2 [Sa70]; (2) Ruzzo's NC algorithm for context-free language recognition [Ru80]; and (3) Borodin and Ruzzo's simulation of simultaneous space and alternation bounded alternating Turing machines by simultaneous space and time bounded alternating Turing machines [CKS81], [Ru81]. More generally, the new game provides a simple understanding of the combinatorial structure of parallel computations. This makes it a useful device for discovering new parallel algorithms.

In order to motivate the new pebble game, it is helpful to question why it has certain characteristics:

(1) *Why does the game use two players, rather than one as in the standard pebble game*? The two competing players model, respectively, the existential and universal moves of an alternating Turing machine, just as the single player in the "black" or "black and white" pebble games [Pi80] models the moves of a sequential machine. The close relationship between alternating Turing machines and other models of parallel computation such as Boolean circuits and PRAMs [Ru81], [SV84] makes the two-person game a good model of parallel computation.

(2) *Why should the gate type* (AND, OR) *enter into the rules of the game*? It is often the case that a circuit has a more efficient pebbling when the interpretation of gates is exploited than it does in the uninterpreted game defined by Dymond and Tompa [DT85]. This translates into more efficient parallel algorithms. The Cocke-Kasami-Younger algorithm for context-free language recognition [HU69] is an example of a circuit for a natural problem for which such a speedup can be demonstrated [Ve86].

(3) *Why are the roles of the two players made symmetric*? Without this symmetry, the game captures those alternating computations of the form "existentially guess and universally verify" [Sa70], [Ru80]. By giving the universal player a more active role than merely verifying the guesses of the existential player, the game captures the symmetry between existential and universal moves found in more general alternating computations. In fact, it will be seen that the new computations thus captured are exactly those in $AC^1 - LOGCFL$.

As noted earlier, the game defined in this paper provides a setting for examining the relationship between the two parallel complexity classes LOGCFL and AC^1. It is known that $LOGCFL \subseteq AC^1$ [Ru80]. Showing that the inclusion is proper is likely to be difficult, as it would imply that $\mathcal{P}$ is not equal to $DSPACE(\log n)$, settling this celebrated open problem in complexity theory. This is because of the following relationship among these complexity classes [Co85]:

$$DSPACE(\log n) \subseteq LOGCFL \subseteq AC^1 \subseteq NC^2 \subseteq NC \subseteq \mathcal{P}.$$

For many problems in LOGCFL the algorithms that show their membership in that class also show their membership in AC^1. However, these algorithms do not use the full power of AC^1 computations. The two-person game defined in this paper provides a model of computation in which this perceived difference can be quantified. This is done by characterizing the two classes using the same measures of resources in the game model. The results so obtained not only illustrate the striking similarity between these two classes, but also isolate the fundamental way in which they differ: the recognition of languages in LOGCFL does not utilize the symmetry between the two players, whereas the recognition of languages in AC^1 does. Thus, the results indicate why these two classes may not be equal.

Both LOGCFL and AC^1 have been characterized using different models of computation [Co85]. These characterizations fall into three categories: models on which characterizations of both LOGCFL and AC^1 are known, models on which only

characterizations of LOGCFL are known, and models on which only characterizations of AC^1 are known. Among the characterizations of the first category, the only result that characterized both of them using the same type of resources on a model of computation is that of Immerman [Im82]. This result used the size and number of variables needed to express properties in first-order logic as resources in the model, and showed that LOGCFL is the class of properties expressible in $O(1)$ variables and $O(\log n)$ size when the universal quantifiers are restricted to being Boolean, and that AC^1 is the class of properties expressible using $O(1)$ variables and $O(\log n)$ size when the universal quantifiers are unrestricted. Considering the arity of universal quantification as a resource, this showed how the languages in the two classes differed in a fundamental way. The characterizations in this paper are in the same spirit and show another fundamental way in which these classes differ.

As a final note, the new game suggests a new complexity measure, called "role switches," for Boolean circuits and alternating machines. This measure captures a higher-level notion of alternation than the standard measure of alternations between existential and universal moves. The characterizations of LOGCFL and AC^1 demonstrate that these two classes differ in how much of the role-switch resource is used: languages in LOGCFL use no role switches and those in AC^1 use $O(\log n)$ role switches. This resource thus can be used to define a natural hierarchy of parallel complexity classes between LOGCFL and AC^1.

Section 2 contains some basic definitions. In § 3, the original two-person pebble game defined by Dymond and Tompa [DT85] is presented. In § 4, the new game is defined. In § 5, the classes LOGCFL and AC^1 are characterized using this game. In § 6, the game is used to unify the three results mentioned in a previous paragraph. Section 7 contains some concluding remarks and open problems.

2. Preliminaries. A *Boolean circuit* G_n with n inputs is a finite acyclic directed graph with vertices having indegree zero or two and labeled as follows. Vertices of indegree zero are labeled from the set $\{0, 1, x_1, x_2, \cdots, x_n, \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n\}$. All vertices with indegree two (also called *gates*) are labeled either AND or OR. Vertices with outdegree zero are called *outputs*. The evaluation of G_n on inputs of length n is defined in the standard way. Typically, only circuits with one output vertex will be considered. This makes it convenient to consider circuits as language acceptors.

A *family* of circuits is a sequence $\{G_n | n = 0, 1, 2, \cdots\}$, where the nth circuit G_n has n inputs.

The *size* $C(G_n)$ of a circuit G_n is the number of gates in G_n. The *depth* of a vertex v in a circuit is the length of a longest path from any input to v. The depth of a circuit is the depth of its output vertex.

Not including negation gates in the definition of a Boolean circuit is done with no loss of generality as there is a well-known technique to simulate, with a doubling of size and no increase in depth, a Boolean circuit with negations by a Boolean circuit in which the negations appear only at the inputs. (See, for example, [Go77].)

A family $\{G_n\}$ of circuits is said to be *uniform* if, on input 1^n, a reasonable encoding of G_n can be generated by a deterministic Turing machine using space $O(\log C(G_n))$. This uniformity condition is sometimes referred to as log-space uniformity in the literature. This uniformity condition suffices for the purposes of this paper. See the paper by Ruzzo [Ru81] for a more detailed treatment of uniformity conditions for circuits.

For the rest of the paper, $\{G_n\}$ will denote a *uniform* family of *polynomial-size* Boolean circuits, where G_n has n inputs and one output. L will denote the language accepted by this family.

In a directed acyclic graph or Boolean circuit, u is an *immediate predecessor* of v if and only if (u, v) is a directed edge, and u is a *predecessor* of v if and only if there is a directed path from u to v. The predecessor relation is an irreflexive and transitive relation.

The notion of an accepting subtree can be defined for Boolean circuits by analogy to the notion of accepting subtrees for alternating Turing machines [Ru80]. Consider the tree-equivalent $T(G_n)$ of the circuit G_n. (The tree-equivalent of a graph is obtained by replicating vertices whose outdegree is greater than one until the resulting graph is a tree.) Let $x \in L$ be of length n. An *accepting subtree* H of a circuit G_n on input x is a subtree of $T(G_n)$:

- That includes the output vertex;
- That includes both the immediate predecessors of any AND vertex included in H;
- That includes exactly one immediate predecessor of any OR vertex included in H; and
- In which any included vertex of indegree zero has value 1 as determined by the input x.

The following fact is easy to verify.

FACT. G_n evaluates to 1 on input x if and only if there exists an accepting subtree of G_n on input x.

3. The uninterpreted game. The two-person pebble game defined by Dymond and Tompa [DT85] is played on the vertices of a directed acyclic graph by two players called the *Challenger* and the *Pebbler*.

Rules. The Challenger begins the game by challenging any vertex v. The game now proceeds in rounds with each round consisting of a *pebbling* move followed by a *challenging* move. In a pebbling move, the Pebbler picks up zero or more pebbles from vertices already pebbled and places pebbles on any nonempty set of vertices. In a challenging move, the Challenger either rechallenges the currently challenged vertex v, or challenges one of the vertices that acquired a pebble in the current round.

The Challenger loses the game at a vertex v if, immediately following the Challenger's move, v is the current challenged vertex and all immediate predecessors of v have pebbles on them.

This two-person game will be referred to as the *uninterpreted game* in the rest of the paper.

If G is thought of as a circuit computing some function, then a play of this two-person game corresponds to an alternating implementation of that circuit, in the following sense. A pebble placed on a vertex v by the Pebbler corresponds to existentially guessing the value computed at v. A move of the Challenger corresponds to universally verifying each of those guesses, plus the fact that those guesses lead to the correct value computed at the current challenged vertex.

Resources. There are three resources of interest in a play of this game: *space*, *time*, and *rounds*. The space used is the maximum number of pebbles on the graph at any point in the game, the time is the number of pebble placements, and the rounds is the number of rounds in the play.

The game on a graph with n inputs is said to take space $p(n)$ (time $t(n)$, rounds $r(n)$), if and only if there is a winning strategy for the Pebbler such that, for all plays by the Challenger, the Pebbler uses at most space $p(n)$ (time $t(n)$, rounds $r(n)$, respectively).

Illustrations of the game on a simple path with N vertices and a tree with N

vertices are presented below. These two results will be useful later.

LEMMA 1. *The uninterpreted game on a path of N vertices can be played with two pebbles and $O(\log N)$ time.*

Proof. A simple divide-and-conquer strategy is used to prove this lemma. □

The game on a binary tree uses the following tree-cutting lemma. This tree-cutting lemma is a classical result that was first used by Lewis, Stearns, and Hartmanis [LSH65] to show that context-free languages are in DSPACE($\log^2 n$).

LEMMA 2. *Let T be a tree with N vertices, each of which has at most two children. Then there is a vertex s of T such that the subtree rooted at s has p vertices, where $N/3 \leqq p < 2N/3+1$.*

(See the paper by Lewis, Stearns, and Hartmanis [LSH65] or the book by Hopcroft and Ullman [HU69] for a proof.)

Ruzzo [Ru80] discovered an efficient parallel algorithm for context-free language recognition, based on Lemma 2. The essence of his algorithm is captured in the following lemma.

LEMMA 3. *Let T be a tree with N vertices, each of which has at most two children. Then the uninterpreted game on T can be played using $O(1)$ pebbles and $O(\log N)$ time.*

Proof. Let r be the root of T with the initial challenge. Let s be an internal vertex of T as in Lemma 2. Let T_2 be the subtree rooted at s and T_1 be the subtree rooted at r with all vertices except s of T_2 deleted from it. Suppose the Pebbler pebbles s. If the Challenge is moved to s, the game is confined to T_2. If the Challenge is retained at r, the game is confined to T_1. In either case, the size is decreased by a constant factor. Therefore, in $O(\log N)$ steps the game will be over.

The problem with this strategy is that the number of pebbled vertices is $\Omega(\log N)$, as no pebbles are picked up. To keep the number of pebbled vertices constant, a two-phase strategy is adopted. Let r_1, the current challenged vertex, be the root of a subtree T_1 with three pebbled leaves. Given three distinguished leaves of a binary tree, there is an internal vertex v that is an ancestor of exactly two of them. The Pebbler pebbles v. This divides the tree T_1 into two subtrees T_{12}, the subtree with v as the root, and T_{11}, the subtree rooted at r_1 with all vertices except v of T_{12} deleted from T_1. Whether the Challenge is moved or retained, the subtrees involved have only two pebbled leaves, so two pebbles can be picked up. It is possible that v may not induce a balanced split of T_1. This is taken care of in the next round, when the Pebbler chooses a vertex that causes a balanced split as guaranteed by Lemma 2. Therefore, the size of the subtree to which the game is confined decreases by a constant factor every two rounds.

This strategy uses four pebbles and $O(\log N)$ time. It is possible to reduce the number of pebbles used to two with $O(\log N)$ time by a slightly more involved pebbling strategy. The details are left to the interested reader. □

3.1. Motivation for extensions. This section examines the two-person pebble game played on the graph underlying a Boolean circuit. A play of this game on a circuit can be viewed as a parallel evaluation of the circuit on some input. The Pebbler in the game is like a prover asserting values for the gates on which it places pebbles and the Challenger is like a verifier who verifies these assertions in parallel. Given an input x in the language accepted by a circuit G_n, the time to play the game on G_n corresponds to the time to discover a "proof" that the circuit evaluates to 1 on input x. This suggests a correspondence between the resources of the game and the resources of circuits that accept the language. In fact, the following result by Dymond and Tompa [DT85] formalizes such a relationship.

THEOREM. *Let G be a Boolean circuit that accepts a finite language L. If the uninterpreted game can be played on G in time t, then L is accepted by a Boolean circuit H of depth $O(t)$.*

This motivates the discovery of efficient pebbling strategies on Boolean circuits. One way of achieving efficiency is to exploit some of the properties of Boolean circuits. Two such properties are discussed below.

Consider the n input Boolean circuit G_n that is a complete binary tree. The uninterpreted game on the graph underlying this circuit can be played in $\log n$ time and two pebbles, by always pebbling the immediate predecessors of the challenged vertex. It can be shown that $\Omega(\log n)$ time is required to play the uninterpreted game on G_n using any number of pebbles. Suppose now that all gates of G_n are OR gates. Then a "proof" that the circuit evaluates to 1 on some input x is a path of $\log n$ vertices from some input with value 1 to the output. By Lemma 1, the game can be played on this path in $\log \log n$ time and two pebbles. There are circuits of other natural problems, such as Boolean matrix multiplication and context-free language recognition, for which similar speedups are made possible by exploiting the types of the gates in the circuit.

Consider the following natural circuit G_n for computing the inner product of two Boolean vectors of length m each. G_n consists of one output and $n = 2m$ inputs. The output is the root of a balanced binary tree of OR gates whose m leaves are AND gates with two inputs each. The argument given above for playing the game on a tree of OR gates applies to this circuit also. That is, without taking into consideration the gate types, any pebbling of this circuit with only a constant number of pebbles takes $\Omega(\log n)$ time. If the gate types are taken into account a suitably modified game can be played with two pebbles and $O(\log \log n)$ time.

For context-free language recognition, consider the Cocke-Kasami-Younger algorithm. It can be shown that the game takes $\Omega(n)$ time on this circuit if the gate types are not taken into account, but can be played on this circuit with two pebbles and $O(\log n)$ time if they are [Ve86].

This motivates extending the uninterpreted game to take into consideration the types of the gates of the circuit on which the game is played.

Suppose now that all the gates of the binary tree circuit G_n are AND gates. In this case, a proof that the circuit evaluates to 0 on some input x is a path of $\log n$ vertices from some input with value 0 to the output. By Lemma 1, the game can be played on this path in $\log \log n$ time with two pebbles. A proof that G_n evaluates to 1 will be simply the failure to demonstrate such a path in $\log \log n$ time with two pebbles. In other words, a proof that a circuit G_n evaluates to 1 on x can consist of showing that it does not evaluate to 0 on x. Intuitively, we would like to take advantage if the complement of the problem is easier. In general, when the circuit consists of both OR and AND gates, sometimes it will be more efficient to demonstrate that a given gate evaluates to 1, and sometimes 0.

This motivates the second extension of the uninterpreted game to incorporate duality between the two players.

4. The dual game. The new game obtained by extending the uninterpreted game will be referred to as the *dual interpreted two-person pebble game* (or *dual game*, for short). This game is played by two players called *Player* 0 and *Player* 1 on the vertices of a Boolean circuit G_n together with its input x. The objective of Player 0 (Player 1) is to establish that the output of the circuit evaluates to 0 (1). Thus, a pebble placement or challenge on a gate v by Player 0 (Player 1) corresponds to asserting that v evaluates

to 0 (1). At any point, one of the players takes on the role of the Challenger and the other that of the Pebbler. The role of a player is automatically determined as part of the circuit information as follows. The gates in G_n are partitioned into two sets, those of "challenge type" 0 and those of "challenge type" 1. A challenge placed on a gate of challenge type 0 (challenge type 1) causes Player 0 (Player 1) to be the Challenger in the next round. For the remainder of this paper, all circuits will be assumed to contain this additional bit per vertex. A Boolean circuit augmented with this role information for each of its gates will be referred to as an *augmented* circuit. For augmented uniform circuits, it will be assumed that this role information is available from the log space uniformity machine.

A challenge by Player 0 (Player 1) will be referred to as a 0-challenge (1-challenge). Similarly, a pebble placed by Player 0 (Player 1) will be referred to as a 0-pebble (1-pebble).

Rules. The initial challenge is on the output gate. The game proceeds in rounds with a round consisting of the following three parts: (a) If the game is not over at the currently challenged vertex u according to the conditions below, then Player 0 is the Challenger for this round if u is of challenge type 0 and the Pebbler otherwise. (b) In the *pebbling move*, the Pebbler picks up zero or more of its own pebbles from vertices already pebbled and places pebbles on any nonempty set of vertices. (c) In the *challenging move*, the Challenger either rechallenges the currently challenged vertex, or challenges one of the vertices that acquired a pebble in the current round.

Winning/losing conditions. Player 1 wins the game if, immediately following the Challenger's move, the current challenged vertex is an input with value 1, or an OR gate at least one of whose immediate predecessors is 1-pebbled, or an AND gate both of whose immediate predecessors are 1-pebbled. Player 0 wins if, immediately following the Challenger's move, the current challenged vertex is an input with value 0, or an OR gate both of whose immediate predecessors are 0-pebbled, or an AND gate at least one of whose immediate predecessors is 0-pebbled. It is also possible to have a winner in an infinite play of the game, namely that player (if either) who is the Pebbler in only finitely many rounds. (The purpose of this last rule is to force the eventual loser to make progress as the Pebbler.)

Resources. There are four resources of interest in a play of this game: *space*, *time*, *rounds*, and *role switches*.

The game on an augmented circuit G_n with input $x \in L$ of length n is said to use space $p(n)$ (time $t(n)$, rounds $r(n)$, role switches $s(n)$, respectively) if and only if there is a strategy for Player 1 such that, for all plays by Player 0, Player 1 wins using at most $p(n)$ 1-pebbles ($t(n)$ 1-pebble placements, $r(n)$ rounds in which Player 1 is the Pebbler, $s(n)$ role switches between pebbling and challenging roles, respectively). Resources when $x \notin L$ are defined by interchanging Player 0 and Player 1. The augmented circuit G_n is said to be *pebbleable* in space $p(n)$ (time $t(n)$, rounds $r(n)$, role switches $s(n)$, respectively) if and only if, for all x of length n, the game on G_n with input x uses at most space $p(n)$ (time $t(n)$, rounds $r(n)$, role switches $s(n)$, respectively). Note that the loser's space, time, and rounds are not even counted.

These notions should all be defined more precisely. This is done by considering the game tree associated with the game. The reader willing to proceed on an intuitive level may skip the remainder of this section.

The game tree is defined in terms of configurations and moves of the game. Fix an augmented circuit G_n and its input x.

A *configuration* of the game is a tuple $(t, P_1, P_0, R_1, R_0, v)$ where:

$t \in \{P, C\}$ indicates whether it is the Pebbler's or the Challenger's turn to move;

P_1 is the set of vertices with 1-pebbles on them from previous rounds;
P_0 is the set of vertices with 0-pebbles on them from previous rounds;
R_1 is the set of vertices 1-pebbled in the current round;
R_0 is the set of vertices 0-pebbled in the current round; and
v is the current challenged vertex.

The initial configuration of the game is $(P, \varnothing, \varnothing, \varnothing, \varnothing, s)$, where s is the output of the circuit.

A configuration $(P, P_1, P_0, \varnothing, \varnothing, v)$ is *terminal* if v is an input, or an OR (AND) gate with some (both) of its immediate predecessors in P_1 or both (some) of its immediate predecessors in P_0.

A *move* in the game is made in accordance with a binary relation $\vdash$ on configurations defined as follows (where P_0, P_1, S_0, and S_1 are arbitrary sets of vertices, and R_0 and R_1 are arbitrary nonempty sets of vertices):

$(P, P_1, P_0, \varnothing, \varnothing, v) \vdash (C, P_1 - S_1, P_0, R_1, \varnothing, v)$, for all configurations $(P, P_1, P_0, \varnothing, \varnothing, v)$ that are not terminal and where v is of challenge type 0;

$(P, P_1, P_0, \varnothing, \varnothing, v) \vdash (C, P_1, P_0 - S_0, \varnothing, R_0, v)$, for all configurations $(P, P_1, P_0, \varnothing, \varnothing, v)$ that are not terminal and where v is of challenge type 1;

$(C, P_1, P_0, R_1, \varnothing, v) \vdash (P, P_1 \cup R_1, P_0, \varnothing, \varnothing, u_1)$, for all $u_1 \in R_1 \cup \{v\}$;

$(C, P_1, P_0, \varnothing, R_0, v) \vdash (P, P_1, P_0 \cup R_0, \varnothing, \varnothing, u_0)$, for all $u_0 \in R_0 \cup \{v\}$.

The *game tree* T is a maximal rooted tree whose nodes are labeled by configurations of the game, and whose root is labeled by the initial configuration, and whose edge relation is given by $\vdash$. Note that the leaves of the tree are labeled by terminal configurations.

A *finite play* of the game is a finite path in the game tree from the root to some leaf labeled by the configuration $(P, P_1, P_0, \varnothing, \varnothing, v)$. It is a *winning finite play for Player* 1 if v is an input with value 1, or if v is an OR gate at least one of whose immediate predecessors is in P_1, or if v is an AND gate both of whose immediate predecessors are in P_1; otherwise it is a *winning finite play for Player* 0. An infinite path Π in the game tree is a *winning infinite play* for the player (if either) that is the Pebbler in only finitely many configurations on Π.

A *winning strategy for Player* 1 (if it exists) is a subtree W of T such that:

(1) W contains the root of T;

(2) W contains exactly one child of every nonterminal node in W that is labeled by a configuration in which it is Player 1's turn to move;

(3) W contains all children of every nonterminal node in W that is labeled by a configuration in which it is Player 0's turn to move; and

(4) All paths in W are winning (finite or infinite) plays for Player 1.

A *winning strategy for Player* 0 is defined dually.

It is not hard to show that, for all augmented circuits G_n, and for all inputs x, exactly one of Player 0 and Player 1 has a winning strategy. In particular, Player 1 has a winning strategy if and only if G_n evaluates to 1 on input x.

Let $\{G_n\}$ accept the language L, where each member G_n of the family is an augmented circuit. The game on G_n with input $x \in L$ can be *played in space* $p(n)$ if and only if there is a winning strategy for Player 1 in which every pebbling configuration $(P, P_1, P_0, \varnothing, \varnothing, v)$ along every path satisfies $|P_1| \leqq p(n)$. It can be *played in time* $t(n)$ if and only if there is a winning strategy for Player 1 for which every path Π satisfies $\sum |R_1^i| \leqq t(n)$, where the sum is over all challenging configurations $(C, P_1^i, P_0^i, R_1^i, R_0^i, v^i)$ on Π. It can be *played in* $r(n)$ *rounds* if and only if there is a winning strategy for Player 1 for which every path has at most $r(n)$ pebbling configurations $(P, P_1, P_0, \varnothing, \varnothing, v)$ with v of challenge type 0. It can be *played in* $s(n)$ *role*

switches if and only if there is a winning strategy for Player 1 in which, on any path, there are at most $s(n)$ edges $(C, P_1, P_0, R_1, R_0, v) \vdash (P, P_1', P_0', \varnothing, \varnothing, u)$ having the challenge types of v and u unequal.

Resources on inputs $x \notin L$ are defined dually, considering winning strategies for Player 0 in place of those for Player 1.

Finally, G_n is *pebbleable* in space $p(n)$ (time $t(n)$, rounds $r(n)$, role switches $s(n)$, respectively) if and only if, for all x of length n, the game on G_n can be played in space $p(n)$ (time $t(n)$, rounds $r(n)$, role switches $s(n)$, respectively). Note again that only the resources used by the winning player are counted.

It is important to note the similarity between the definition of a winning strategy and the definition of an accepting subtree of an alternating Turing machine. This relationship between the two structures is critical in the correctness proof of Theorem 11.

4.1. Example. An example is presented here to illustrate how role switches can help in reducing the time to play the game on some Boolean circuits.

Consider a two-layered binary tree circuit G_n defined as follows. The top layer F_m is a complete binary tree of OR gates with m leaves $y_1, y_2, \cdots, y_m$. Each of these m leaves is the root of a complete binary tree of AND gates with m leaves. (See Fig. 1.) Thus, G_n has $n = m^2$ inputs.

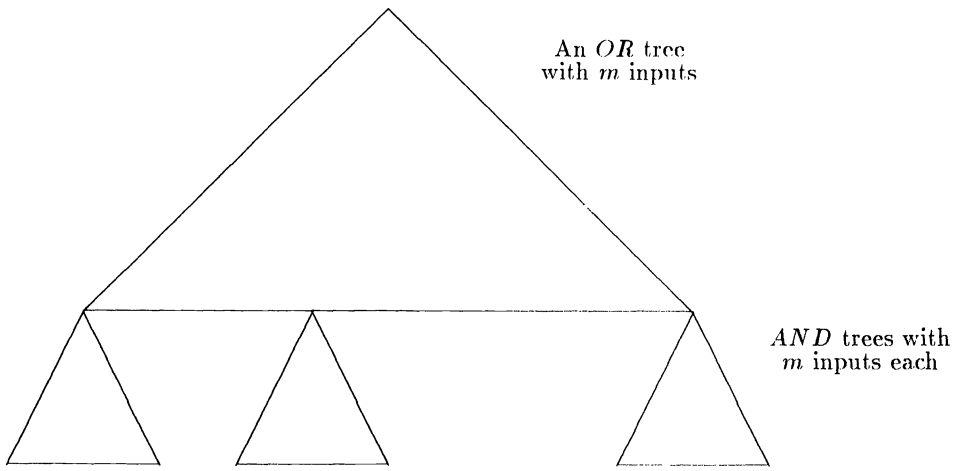


FIG. 1. *Two-layered binary tree circuit.*

Suppose all gates of G_n are designated as of the same challenge type, say, challenge type 0. Then, given an input x the game on G_n will take $\Omega(\log m)$ time no matter how many pebbles are available. Consider now allowing one role switch by designating all OR gates as gates of challenge type 0 and all AND gates as gates of challenge type 1. Then G_n is pebbleable in $O(\log \log m)$ time and two pebbles as follows.

Let x be an input on which G_n evaluates to 1. Thus, the goal is a winning strategy for Player 1 that does not exceed the stated bounds. Let $G_{n,x}$ be an accepting subtree of G_n. The output of G_n is of challenge type 0 and therefore Player 1 begins the game as the Pebbler. The subtree $G_{n,x}$ starts off as a path of OR gates. Let u be the first AND gate on this path. Player 1 pebbles u. If Player 0 retains its challenge on the output, then Player 1 can win the game on the path using two pebbles, $\log \log m$ time, and no role switches, as in Lemma 1. Suppose Player 0 moves its challenge to u. Then a role switch occurs, so that Player 0 is the pebbler for the next round. If Player 0

never pebbles a gate in the subtree of $G_{n,x}$ rooted at u, Player 1 retains its challenge on u and wins using no additional resources. Suppose Player 0 pebbles a gate in this subtree. Consider the first such move. Let w be such a gate of minimum depth among the gates that are pebbled. Player 1 moves its challenge to w. The depth of the challenged gate decreases without Player 1 using any additional resources. Therefore, in a finite number of rounds Player 1 wins without expending any additional resources.

Suppose now that x is an input on which G_n evaluates to 0, and a winning strategy for Player 0 must be described. If Player 1 never pebbles a predecessor of the challenged gate that evaluates to 0, Player 0 retains its challenge and wins using no resources. Whenever Player 1 pebbles one or more predecessors of the challenged gate that evaluate to 0, Player 0 moves its challenge to one such gate of minimum depth. In this manner, the challenge eventually reaches an AND gate v that evaluates to 0, without expending any of Player 0's resources. At this point Player 0 becomes the Pebbler, and can win the game using two pebbles and log log m time by pebbling the path from some input with value 0 to v.

This pebbling strategy forms the basis of the proof of Theorem 10.

5. Characterizations of LOGCFL and AC^k. This section contains the main results of the paper, namely the characterizations of the classes LOGCFL and AC^1.

Let PEBBLE, TIME, SWITCHES($p(n)$, $t(n)$, $s(n)$) be the class of languages L accepted by a uniform family $\{G_n\}$ of polynomial-size augmented circuits such that Player 1 begins the game as the Pebbler, and such that G_n is pebbleable in $p(n)$ pebbles, $t(n)$ time, and $s(n)$ role switches.

In §§ 5.1–5.3, the following characterizations are demonstrated.

THEOREM 4. LOGCFL = PEBBLE, TIME, SWITCHES($O(1)$, $O(\log n)$, 0).

Proof. This follows from Theorem 8 and Corollary 12 below. □

AC^k is defined as the class of languages recognized by alternating Turing machines using space $O(\log n)$ and alternation depth $O(\log^k n)$ [Co85].

THEOREM 5. *For any* $k \geq 1$,

$$AC^k = \text{PEBBLE, TIME, SWITCHES}(O(1), O(\log^k n), O(\log^k n)).$$

Proof. This follows from Theorems 10 and 11 below. □

In Theorem 5 note that, given the time bound, the number of role switches is as great as possible. Such a class will be denoted simply as PEBBLE, TIME($p(n)$, $t(n)$).

As a consequence of Theorem 5, the class NC can also be characterized in terms of the pebble game. This is because $NC = \bigcup_{k \geq 0} AC^k$ [Co85].

THEOREM 6. $NC = \text{PEBBLE, TIME}(O(1), (\log n)^{O(1)})$.

It is interesting to note that the class $\mathcal{P}$ also can be defined in this model by bounding only the space used, i.e., $\mathcal{P} = \text{PEBBLE}(O(1))$, where PEBBLE($p(n)$) denotes the class of languages L accepted by a uniform family $\{G_n\}$ of polynomial-size augmented circuits such that G_n is pebbleable in space $p(n)$. The characterization so obtained in § 5.4 is similar to the one by Immerman [Im82], where it is shown that $\mathcal{P}$ is the class of properties expressible in first-order logic using $O(1)$ variables and polynomial size.

5.1. Game on a LOGCFL circuit. The notion of tree-size for Boolean circuits defined below is analogous to the notion of tree-size for alternating Turing machines [Ru80], and is useful in obtaining a circuit characterization of LOGCFL.

DEFINITION. Let x be a length n input on which G_n evaluates to 1. G_n is said to use tree-size $Z(n)$ on input x if and only if there is an accepting subtree of G_n of size at most $Z(n)$. If for every $x \in L$, G_n uses tree-size $Z(n)$ on input x, then G_n is said to have *tree-size* $Z(n)$.

The following lemma is the circuit analogue of the alternating Turing machine characterization of LOGCFL [Ru80].

LEMMA 7. *If $L \in$ LOGCFL, then there exists a uniform family of polynomial-size Boolean circuits $\{G_n\}$ accepting L such that $\{G_n\}$ has polynomial tree-size.*

Proof. Let $L \in$ LOGCFL. Then, by the characterization of LOGCFL by Ruzzo [Ru80], there is an alternating Turing machine M that accepts L within space $O(\log n)$ and $n^{O(1)}$ tree-size. Such an alternating Turing machine can be simulated by a uniform family of Boolean circuits with polynomial size and polynomial tree-size by using the method of Ruzzo [Ru81]. (Actually, this simulation requires that the simulated alternating Turing machine be in a normal form such that only one input symbol is read along any path of the machine's computation tree. This is accomplished as follows. If a read is encountered in the middle of a path, M existentially guesses the value to be read and universally does two things: verifies that the read value is correct and, in parallel, continues with the successor of the original read configuration as though the guess is correct. This does not increase the tree-size by more than a constant factor.) □

The converse of Lemma 7 can also be proved by adapting Ruzzo's proofs [Ru80], [Ru81].

THEOREM 8. LOGCFL $\subseteq$ PEBBLE, TIME, SWITCHES($O(1)$, $O(\log n)$, 0).

Proof. Let $L \in$ LOGCFL. By Lemma 7, there is a uniform family $\{G_n\}$ of polynomial-size Boolean circuits that have polynomial tree-size and accept L. All the gates of G_n are designated as gates of challenge type 0.

If the input x is not in L, then Player 0 has a winning strategy identical to that of Player 0 in the subgraph F_m in § 4.1. This strategy uses no resources, since all vertices are of challenge type 0. If $x \in L$, let H be a polynomial-size accepting subtree of G_n. By Lemma 3, H can be pebbled in the uninterpreted game of § 3 using $O(1)$ pebbles and $O(\log n)$ time. Player 1 simulates this strategy on G_n by pebbling a gate whenever any of its copies in H is pebbled, and removing the pebble from a gate whenever all of its copies in H become pebble-free. That Player 1 wins on G_n in the same round as the Pebbler would win on H follows from the definition of accepting subtree and the rules of the games as follows. If the Challenger loses at an input of H, then the corresponding input in G_n must have value 1, so Player 1 wins. If the Challenger loses at an OR gate v of H, then the gate in G_n corresponding to the child of v in H is also 1-pebbled, so Player 1 wins. Finally, if the Challenger loses at an AND gate v of H, then both inputs of the corresponding AND gate in G_n are 1-pebbled, so Player 1 wins. □

5.2. Game on an AC^k circuit. The notion of alternation can be defined naturally for Boolean circuits analogously to the notion of alternation in alternating Turing machines. This definition is used in Lemma 9 below to obtain a circuit characterization of AC^k.

DEFINITION. A language L is said to be accepted by a family $\{G_n\}$ of Boolean circuits within *alternation* bound $A(n)$ if and only if, for all paths p in G_n from some input of G_n to the output gate, the number of edges on p connecting an AND gate to an OR gate or vice versa is at most $A(n)$.

LEMMA 9. *For any $k \geq 1$, if $L \in AC^k$, then there exists a uniform family of polynomial-size Boolean circuits $\{G_n\}$ accepting L such that G_n has $O(\log^k n)$ alternations.*

Proof. The proof is similar to that of Lemma 7. However, the method used there of putting the simulated alternating Turing machine M in normal form will not do here, as it introduces at least one alternation for every input symbol read. Instead, M is simulated as in Borodin and Ruzzo's theorem [Ru81] (see also Corollary 17) by an

alternating Turing machine in normal form that uses $O(S)$ space and $O(A+S)$ alternations, where $S = O(\log n)$ and $A = O(\log^k n)$ are the space and alternations, respectively, used by M. □

The converse of Lemma 9 can also be proved using the alternating Turing machine simulation of circuits by Ruzzo [Ru81].

THEOREM 10. *For any* $k \geq 1$,

$$\mathrm{AC}^k \subseteq \mathrm{PEBBLE, TIME, SWITCHES}(O(1), O(\log^k n), O(\log^k n)).$$

Proof. Let $\{G_n\}$ accept L using $O(\log^k n)$ alternations. All OR gates are designated as gates of challenge type 0 and all AND gates are designated as gates of challenge type 1. Let x be an input to G_n.

Case 1. Suppose G_n evaluates to 1 on input x. Then there exists an accepting subtree H of G_n. The proof that Player 1 can win the game on G_n within the stated bounds follows from the claim below.

CLAIM. If v is the current challenged gate in H, no predecessor of v is 0-pebbled, and there are A alternations in the subtree of H rooted at v, then Player 1 can win the game with two pebbles, at most $A+t$ steps, and at most A role switches, where t is the number of steps required to play the uninterpreted game on the longest path of OR gates in H. (By Lemma 1, $t = O(\log n)$.)

Proof of the Claim. The claim is proved by induction on A.

Basis ($A = 0$). Suppose all gates are OR gates. Then Player 1 is the Pebbler. The subtree of H rooted at v is a simple path of OR gates and the result follows immediately.

Suppose all gates are AND gates. Then Player 1 has a winning strategy using no resources, since Player 1 is never the Pebbler.

Induction ($A > 0$). Assume that the claim is true if the number of alternations is less than A. Let the subgraph rooted at v have A alternations.

Let v be an OR gate. Then Player 1 is the Pebbler. The subtree of H rooted at v starts off as a simple path of OR gates. Let u be the first AND gate on this path. Player 1 removes all 1-pebbles from the graph and pebbles u. If Player 0 retains its challenge on v, then Player 1 can win the game on the path using two pebbles, t steps, and no role switches. Suppose Player 0 moves its challenge to u. Then a role switch occurs, so that Player 0 is the Pebbler for the next round. The subtree rooted at u has $A-1$ alternations and no 0-pebbles. Therefore, by the induction hypothesis, Player 1 can win the game on the subtree rooted at u using two pebbles, at most $(A-1)+t$ steps and at most $A-1$ role switches. It took one pebble placement and one role switch to get the game to u, so the total number of steps is at most $A+t$ and the total number of role switches is at most A.

Let v be an AND gate. Then Player 1 is the Challenger. If Player 0 never pebbles a predecessor of v that evaluates to 1, Player 1 retains its challenge on v and wins using no resources. Suppose Player 0 pebbles a predecessor of v that evaluates to 1. Consider the first such move. Let u be such a predecessor of minimum depth among the gates that are pebbled. Player 1 moves its challenge to u. Note that no predecessor of u is 0-pebbled. If u is an input, Player 1 wins immediately. If u is an OR gate, the number of alternations decreases at least by one at the cost of one role switch, and the result follows by induction. If u is an AND gate, Player 0 is the Pebbler again. The depth of the challenged gate is decreased without Player 1 using any resources. Therefore in a finite number of rounds the game reaches one of the two cases above without Player 1 expending any resources. This proves the claim.

Case 2. Suppose G_n evaluates to 0 on x. The proof that Player 0 can win the game on G_n within the claimed bounds is dual to the proof above. □

5.3. An alternating Turing machine simulation of the game.

THEOREM 11. *For $r \geqq \log n$, if L is accepted by a uniform family $\{G_n\}$ of polynomial-size augmented circuits that is pebbleable in p pebbles, t time, and r rounds in the dual game, then L is accepted by an alternating Turing machine within space $O(p \log n)$, time $O(t \log n)$, and alternations $O(r)$. If, in addition, Player 1 is always the Pebbler, then L is accepted within space $O(p \log n)$ and tree-size $p^{O(r)}$.*

Proof. Without loss of generality, assume that the players each have $2p$ pebbles, and they remove them p at a time. This does not affect the other resources.

An alternating Turing machine M simulates the game using existential (universal) configurations for the moves of Player 1 (Player 0). M begins by guessing the values of p, t, and r.

M keeps track of the time used by each player by initializing two counters T_0 and T_1 to t, and decrementing the appropriate one for each pebble placement. Likewise, M keeps track of the rounds charged to each player by initializing counters R_0 and R_1 to r, and decrementing the appropriate one for each round. M also uses four tapes to record the names of gates: (1) previously 1-pebbled, (2) previously 0-pebbled, (3) pebbled in the current round, and (4) challenged. Each round of the game is simulated as follows.

M first existentially guesses whether Player 1 has won according to the basis rules. If the guess is *yes*, M verifies this using the uniformity machine. M accepts if this is the case, and rejects otherwise. If M guessed that Player 1 has not yet won, M universally does the following: (a) It checks whether Player 0 has won the game, using the uniformity machine. M rejects if this is the case, and accepts otherwise. (b) M existentially determines who should be the current Challenger and universally does two things: verifies with the uniformity machine that the guess is correct and, in parallel, continues with the simulation as though the guess is correct. (See Fig. 2.)

Suppose Player 1 (Player 0) is the Pebbler. M existentially (universally) decides whether or not p pebbles are to be removed in this round. If so, M existentially (universally) deletes p gate names from the "1-pebbled" ("0-pebbled") tape, using a temporary tape for copying.

M then existentially (universally) records one or more gate names on the "recently pebbled" tape, up to a total not exceeding $2p$ on both tapes. If t' new names are recorded, then T_1 (T_0) is decremented by t' and R_1 (R_0) is decremented by 1. If $T_1 < 0$ ($T_0 < 0$) or $R_1 < 0$ ($R_0 < 0$), then M rejects (accepts).

Next M universally (existentially) chooses v as the new challenged gate, where v is chosen from among the gates on the "challenged" and "recently pebbled" tapes. v is written on the "challenged" tape, and the "recently pebbled" tape is appended to the "1-pebbled" ("0-pebbled") tape and erased.

M continues these steps until the basis conditions are reached, or one of the players runs out of time or rounds.

Correctness. Suppose G_n evaluates to 1 on x. The fact that the alternating Turing machine M accepts x is shown by arguing that there is an accepting subtree of the alternating Turing machine's computation tree on x.

Since G_n evaluates to 1 on x, there is a winning strategy S_1 for Player 1 in which Player 1 uses at most $2p$ pebbles, t steps, and r rounds. By construction, there is a corresponding subtree A of M's computation tree that contains the initial configuration, and in which one successor of each existential configuration and all successors of each universal configuration are retained. The major difference between S_1 and A is that some plays (in particular, all infinite plays) in S_1 are cut off in A due to the conditions $T_0 < 0$ and $R_0 < 0$. (Neither $T_1 < 0$ nor $R_1 < 0$ ever occurs, since Player 1's strategy uses

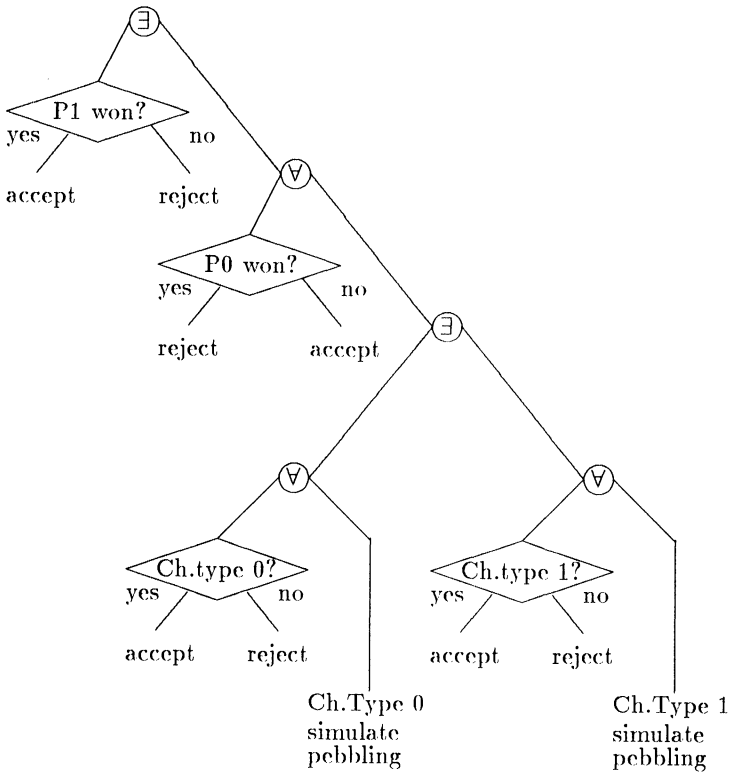


FIG. 2. *Beginning of a round.*

time at most t and rounds at most r.) All leaves of A that correspond to leaves of S_1 will be accepting, since Player 1 wins at all leaves of S_1. Furthermore, all leaves of A that do not correspond to leaves of S_1 are accepting by construction, since they arise from the cutoff due to $T_0 < 0$ or $R_0 < 0$. Thus, M accepts.

The proof that there is a rejecting subtree of M's computation tree for the case when G_n evaluates to 0 on input x is analogous.

Analysis. For the purposes of the analysis here, the uniformity machine will be assumed to be an alternating Turing machine with the simultaneous resource bounds of space $O(\log n)$, time $O(\log^2 n)$, alternations $O(\log n)$ and tree-size $O(n^{O(1)})$. This is possible since an $O(\log n)$ space deterministic Turing machine can be simulated by an alternating Turing machine with these resource bounds [CKS81]. Note that the uniformity machine is consulted at most once on any path of the computation tree. Hence, the resources required by the uniformity machine can be accommodated within the stated bounds, since $t \geqq r \geqq \log n$.

The counters T_0, T_1, R_0, and R_1 take space $O(\log t)$. Since t is at most the size of the circuit G_n, this space is $O(\log n)$. Since only $O(p)$ gate names each requiring $O(\log n)$ space are recorded on the tapes, the total space used by M is $O(p \log n)$.

There are at most six alternations per round of the game and therefore M makes $O(r)$ alternations.

The overall time for pebble placements, and hence for updating the tapes at the ends of the rounds, is $O(t \log n)$. Because pebbles are removed p at a time, the $O(p \log n)$ cost of updating the tapes to reflect removals is only incurred every p placements. Hence, the overall time due to removals is also $O(t \log n)$.

If Player 1 is the Pebbler throughout the game, then the following argument shows that the tree-size of M is $p^{O(r)}$. Consider the actions of the machine corresponding to a round of the game. For the Pebbler's move M existentially guesses at most $2p$ gates to pebble, and for the Challenger's moves it universally chooses a gate from the "challenged" and "recently pebbled" tapes. This universal move results in at most $2p+1$ configurations that correspond to the start of the next round. Thus, each round of the game increases the tree-size by at most a factor of $2p+1$. Over all the r rounds simulated by M, its tree-size is bounded by $p^{O(r)}$. More formally, let $z(k)$ be the tree-size required to demonstrate acceptance from a configuration reachable after k rounds from the initial configuration. Then,

$$z(k) \leqq \begin{cases} (2p+1)z(k+1)+n^{O(1)}, & k<r, \\ n^{O(1)}, & k=r. \end{cases}$$

So, $z(0)=(2p+1)^{O(r)}$. □

COROLLARY 12. *If L is accepted by a uniform family $\{G_n\}$ of polynomial-size augmented circuits that is pebbleable in the dual game in $t=O(\log n)$ steps and $p=O(1)$ pebbles, then L is in* AC^1. *If, in addition, Player* 1 *is always the Pebbler, then L is in* LOGCFL.

Proof. The proof is a direct application of the theorem along with Ruzzo's characterization of LOGCFL [Ru80]. □

5.4. Characterization of $\mathscr{P}$.

THEOREM 13. $\mathscr{P}=\mathrm{PEBBLE}(O(1))$.

Proof. If $L\in\mathscr{P}$, then it is known that there is a uniform family $\{G_n\}$ of polynomial-size Boolean circuits accepting L [La75]. All the gates in G_n are designated as gates of challenge type 0. Suppose G_n evaluates to 1 on x. A winning strategy for Player 1 is to pebble those immediate predecessors of the challenged gate that evaluate to 1. This strategy uses two pebbles. Suppose G_n evaluates to 0 on x. Then Player 0 has a winning strategy that uses no resources, since Player 0 is never the Pebbler.

Conversely, suppose $\{G_n\}$ is a uniform family of polynomial-size augmented circuits accepting L such that G_n is pebbleable with a constant number of pebbles. Then the result follows from Theorem 11 and the fact that $\mathrm{ASPACE}(\log n)=\mathscr{P}$ [CKS81]. □

6. Unifying framework. The dual game defined in § 4 can be played on computation graphs of certain models of computation that have properties similar to Boolean circuits. In this section the dual game is played on the computation graphs of alternating Turing machines. This is used to show that the following three important results in complexity theory are described naturally as pebbling results: (a) Savitch's theorem showing that $\mathrm{NSPACE}(S)\subseteq\mathrm{DSPACE}(S^2)$ [Sa70]; (b) Ruzzo's simulation of an alternating Turing machine with simultaneous space and tree-size bound by an alternating Turing machine with simultaneous space and time bound [Ru80]; and (c) Borodin and Ruzzo's simulation of an alternating Turing machine with simultaneous space and alternation bound by an alternating Turing machine with simultaneous space and time bound [Ru81]. The latter two results generalized Savitch's theorem in two different ways. These three theorems are derived as corollaries of a theorem on the resource requirements for pebbling alternating Turing machine computation graphs.

Let M be an $S(n)\geqq\log n$ space bounded alternating Turing machine that accepts L. Given an input x of length n in L, let $G(M,x)$ be the *computation graph* of M on x; that is, the vertices of $G(M,x)$ correspond to the configurations of the machine M on input x, and there is a directed edge from a vertex u to a vertex v if and only if there is a transition of M from the configuration corresponding to v to the configuration

corresponding to u. It will be assumed that $G(M, x)$ does not have any cycles. This can be arranged by using an extra tape as a "clock" that is incremented each step.

For defining the dual game on $G(M, x)$, the vertices in this graph are partitioned into two sets, those of challenge type 0 and those of challenge type 1. A computation graph is said to be an *augmented* computation graph if this role information is also available for each of its vertices. It will be assumed that for all $x \in L$, there is an alternating Turing machine that can determine this role information for each vertex in $G(M, x)$ in time $O(S(n))$.

The dual game on $G(M, x)$ is defined similarly to the definition in § 4 of the game on Boolean circuits. Its description will be omitted here.

Let ASPACE, TIME $(S(n), T(n))$ denote the class of languages accepted by alternating Turing machines within space $O(S(n))$ and time $O(T(n))$ simultaneously. The classes ASPACE, ALTERNATIONS$(S(n), A(n))$ and ASPACE, TREESIZE$(S(n), Z(n))$ are defined similarly.

THEOREM 14. *Fix any $x \in L$ and an augmented computation graph $G(M, x)$ of a space $S(n)$ bounded alternating Turing machine M on input x. If the dual game on $G(M, x)$ can be played in space $p(n)$ and time $t(n)$, then*

$$L \in \text{ASPACE, TIME}(p(n)S(n), t(n)S(n)).$$

Sketch of Proof. The proof of this theorem is similar to the proof of Theorem 11. □

COROLLARY 15 (Savitch's theorem). *For $S(n) \geqq \log n$,*

$$\text{NSPACE}(S(n)) \subseteq \text{ASPACE, TIME}(S(n), S^2(n)).$$

Proof. Let M in Theorem 14 be a nondeterministic Turing machine. All vertices of $G(M, x)$ are designated to be of challenge type 0. $G(M, x)$ can be pebbled with $O(1)$ pebbles and $O(S(n))$ time, as in Lemma 1. □

COROLLARY 16 (Ruzzo's theorem). *For $S(n) \geqq \log n$,*

$$\text{ASPACE, TREESIZE}(S(n), Z(n)) \subseteq \text{ASPACE, TIME}(S(n), S(n) \log Z(n)).$$

Proof. All vertices of $G(M, x)$ are designated to be of challenge type 0. $G(M, x)$ can be pebbled with $O(1)$ pebbles and $O(\log Z(n))$ time, as in Theorem 8. □

COROLLARY 17 (Borodin and Ruzzo's theorem). *For $S(n) \geqq \log n$,*

$$\text{ASPACE, ALTERNATIONS}(S(n), A(n)) \subseteq \text{ASPACE, TIME}(S(n), S(n)(S(n)+A(n))).$$

Proof. All existential configurations are designated as of challenge type 0 and all universal configurations are designated as of challenge type 1. $G(M, x)$ can be pebbled with $O(1)$ pebbles and $O(S(n)+A(n))$ time, as in Theorem 10. □

7. Conclusion and open problems. A new combinatorial game that abstracts certain synchronous parallel computations has been defined and this game has been used to study the relationship between the complexity classes LOGCFL and AC^1.

Although the dual interpreted two-person pebble game has been defined in § 4 in terms of Boolean circuits, the game is easily extended to computation graphs of other models of computation. An alternating Turing machine is an example of such a model and this was exploited in § 6 to provide a unifying framework for some well-known simulation results involving alternating Turing machines.

The basic difference in the way Player 0 contributes to the computations in LOGCFL and AC^1 has motivated the discovery of new characterizations of LOGCFL on models such as bounded fan-in Boolean circuits, unbounded fan-in Boolean circuits, and alternating Turing machines [Ve86]. These characterizations are in terms of the same resources used to characterize AC^1 on these models.

There are many possible approaches used to compare two complexity classes. The approach taken in this paper is to obtain characterizations of the classes using the same measures of resources on a model of computation, namely the game model. This approach seems general enough to be applicable in the study of problems of this type: given two complexity classes A and B with $A \subseteq B$, is A properly contained in B?

This study raises several open questions. Some of them are listed below.

The ultimate justification for this game would be to use Theorem 11 to discover new efficient parallel algorithms for natural problems. In retrospect, we could imagine Ruzzo's parallel context-free language recognition algorithm [Ru80] being discovered this way. It would be particularly appealing if new algorithms exploited the symmetry available between the two players.

The uninterpreted game is used by Dymond and Tompa [DT85] to provide an alternative proof of the result of Paterson and Valiant [PV76] that $\mathrm{SIZE}(T) \subseteq \mathrm{DEPTH}(O(T/\log T))$ for (nonuniform) Boolean circuits. Is it possible to improve this result using the dual game? What circuits are hard to pebble?

There are natural circuits such as the Cocke-Kasami-Younger circuits on which the dual game without role switches is exponentially faster than the uninterpreted game. A simple example where one role switch provided such a speedup has been seen in § 4.1. It would be interesting to show similar speedups for circuits for natural problems. A related question is to prove that there are AC^1 circuits for natural problems that require $\Omega(\log n)$ role switches if the time is $O(\log n)$.

Acknowledgments. We are indebted to Larry Ruzzo and Richard Ladner for very useful discussions.

REFERENCES

[CKS81] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[Co85] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2-22.

[DT85] P. W. DYMOND AND M. TOMPA, *Speedups of deterministic machines by synchronous parallel machines*, J. Comput. System Sci., 30 (1985), pp. 149-161.

[Go77] L. M. GOLDSCHLAGER, *The monotone and planar circuit value problems are log space complete for P*, SIGACT News, 9 (1977), pp. 25-29.

[HU69] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[Im82] N. IMMERMAN, *Upper and lower bounds for first order expressibility*, J. Comput. System Sci., 25 (1982), pp. 76-98.

[La75] R. E. LADNER, *The circuit value problem is log space complete for P*, SIGACT News, 7 (1975), pp. 18-20.

[LSH65] P. M. LEWIS, R. E. STEARNS, AND J. HARTMANIS, *Memory bounds for recognition of context-free and context sensitive languages*, in Proc. IEEE 6th Annual Symposium on Switching Theory and Logic Design, 1965, pp. 191-202.

[PV76] M. S. PATERSON AND L. G. VALIANT, *Circuit size is nonlinear in depth*, Theoret. Comput. Sci., 2 (1976), pp. 397-400.

[Pi80] N. PIPPENGER, *Pebbling*, Proc. 5th IBM Symposium on Mathematical Foundations of Computer Science, IBM Japan, May 1980.

[Ru80] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. System Sci., 20 (1980), pp. 218-235.

[Ru81] ———, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365-383.

[Sa70] W. J. SAVITCH, *Relationship between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177-192.

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

[Ve86] H. VENKATESWARAN, *Characterizations of parallel complexity classes*, Ph.D. thesis, University of Washington, Seattle, WA, August 1986; available as Tech. Report No. 86-08-06.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 550–558, June 1989

008

SUCCINCT CERTIFICATES FOR ALMOST ALL SUBSET SUM PROBLEMS*

MERRICK L. FURST† AND RAVI KANNAN†

Abstract. Given n natural numbers $a_1, \cdots, a_n$ and a target integer b, the *SubsetSum* problem is to determine whether some subset of the a_i sums to b. That is, to recognize members of the following set:

$$\text{SubsetSum} = \{\langle a_1, \cdots, a_n; b\rangle \mid a_i \in \mathbf{N}, b \in \mathbf{Z}, \text{ and } \exists x \in \{0,1\}^n \text{ such that } a \cdot x = b\}.$$

For a given vector $a = (a_1, \cdots, a_n)$ and integer b, if a subset of the a_i sums to b, then listing which subset provides a short proof that $\langle a; b\rangle \in$ SubsetSum. However, in general there are no short (polynomial-length) proofs of nonmembership unless NP equals coNP.

The main result in this paper provides a proof system that contains polynomial-length nonmembership proofs for a vast majority of the problem instances that do not belong to SubsetSum.

Key words. SubsetSum, certificates, NP-complete, complexity, algorithm, shortest vector, lattice, circuit

AMS(MOS) subject classifications. 68C25, 10E05

1. Introduction.

A *SubsetSum* problem instance is of a set of n positive integer *coefficients* $a_1, \cdots, a_n$ together with a *target* (or right-hand side) integer b. The "answer" to a SubsetSum problem is *yes* if some subset of the coefficients sum to the target and *no* otherwise. For positive integers n, M, let

$$G_n(M) = \{(a_1, a_2, \cdots, a_n) \mid a_i \text{ an integer in the range } \{1 \cdots M\}\}.$$

Let FEA be the set of feasible SubsetSum problems and let INF be the set of infeasible SubsetSum problems, i.e.,

$$\text{FEA} = \{\langle (a_1, \cdots, a_n); b\rangle \mid \text{a subset of the } a_i \text{ sums to } b\},$$

$$\text{INF} = \{\langle (a_1, \cdots, a_n); b\rangle \mid \text{no subset of the } a_i \text{ sums to } b\}.$$

The set FEA is NP-complete, therefore, the set INF is coNP-complete.

Let T be a nondeterministic Turing machine accepting the set INF. The collection of valid computations of Turing machine T may be thought of, in the sense of Cook and Reckhow [8], as a *formal proof system* $\mathcal{F}$ for demonstrating the infeasibility of SubsetSum problems. Define a *generic SubsetSum problem* to be an n-tuple $a = (a_1, a_2, \cdots, a_n)$, with no sum b specified. Then define

$$\text{INF}(a) = \{b \in N \mid b \leqq \sum a_i \text{ and } \langle a; b\rangle \in \text{INF}\}.$$

The *proof complexity* of the generic instance a with respect to $\mathcal{F}$ is the maximum length, over all $b \in \text{INF}(a)$, of proofs of infeasibility using $\mathcal{F}$ of the SubsetSum instance $\langle a; b\rangle$. Let $f(n, \mathcal{F})$ be the maximum proof complexity (using $\mathcal{F}$) taken over all instances a requiring n or fewer bits as input. Since INF is coNP-complete, if $N \neq \text{coNP}$ then $f(n, \mathcal{F})$ grows more rapidly than any polynomial function of n. The main result asserts that, despite this fact, we can isolate a large, interesting class of generic instances a that has small proof complexity.

The main result of this paper is the following.

* Received by the editors October 16, 1987; accepted for publication (in revised form) August 31, 1988. This research was supported by National Science Foundation grants ECS-8418392, CCR-8805199, and PYI DCR-8352081.

† Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.

THEOREM 1. *Let* $M \geqq 2^{3/2n \log n + 5n}$. *There is a proof system* $\mathcal{F}$ *and a polynomial* p *such that*

$$\left(1 - \frac{1}{2^n}\right) \cdot |G_n(M)|$$

of the generic subset sum problems a *in* $G_n(M)$ *have proof complexity bounded above by* $p(n)$.

We also show the following theorem.

THEOREM 2. *There is a deterministic algorithm* A *that solves the SubsetSum problem correctly in all instances and moreover, there exists a polynomial* $p(\cdot)$ *such that, for any* $M \geqq 2^{n^2/2+2n} \cdot n^{3n/2}$, *some subset* S *of* $G_n(M)$ *has the following property*:

- $|S| \geqq (1 - 1/2^n)|G_n(M)|$, *and*
- *For each vector* $a = (a_1, \cdots, a_n) \in S$, *and any* b, A *runs in* $p(|\langle a; b\rangle|)$*-time on* $\langle a; b\rangle$.

This theorem is a strengthening of a result of Lagarias and Odlyzko [22]. They exhibit a deterministic polynomial-time algorithm B that accomplishes the following. Pick any constant c, and let $M \geqq 2^{cn^2}$. For all but $(1 - 1/2^n)$ of the vectors $a = (a_1, \cdots, a_n)$ in $G_n(M)$, if integer b *is* the sum of a subset of the a_i, then their Algorithm B finds which a_i sum to b. Frieze [11] simplifies and sharpens their argument. Aside from the constants in the lower bound on M, the algorithm of our Theorem 2 improves on these results in that it solves instances that have a negative answer as well as those that have a positive answer.

1.1. Comparison with previous results. There are several results known to lead to proof systems for nonmembership for portions of other NP-complete languages. The proof systems so obtained share a similar structure. Typically, a simple polynomial-time property Q is given. This property is chosen so as to imply nonmembership, i.e., if property Q is true of an instance I then I is not in the NP-complete set. Property Q is then shown to hold for almost all instances within a restricted class. In this way, since Q is polynomial time and holds for almost all instances, simply testing for Q becomes a way of giving short proofs of nonmembership. A more detailed background can be found in § 4.

Our proof system differs from these in two respects. The property Q we identify is not known to be polynomial-time computable. The power of nondeterminism seems to be needed to prove that Q holds. As well, the set of instances we consider is comprehensive and not very restrictive. It contains all SubsetSum problems with coefficients $O(n \log n)$-bits long.

It is an interesting open problem to find a general proof system for other NP-complete sets such as three-CNF satisfiability, vertex cover, and clique. There are indications that the techniques we describe could be used for such problems.

2. Main theorem. The algorithms implicit in both our theorems use techniques from the geometry of numbers. Good general references for this subject are Cassels [5] and Lekkerkerker [25]. For a recent survey, see [19]. We begin with a definition and some lattice terminology. A *lattice* in $\Re^n$ (Euclidean n space) is the set of all *integer* linear combinations of some linearly independent vectors $b_1, b_2, \cdots, b_m$ in $\Re^n$. The set of vectors $b_1, b_2, \cdots, b_m$ is a *basis* and the lattice it generates is denoted $L(b_1, b_2, \cdots, b_m)$. A lattice has many bases; the cardinality is always the same and is called the *dimension.*

Suppose $a = (a_1, a_2, \cdots, a_n)$ is a vector in $\mathbf{Z}^n$. Consider the set of vectors

$$L_a = \{x \mid x \in \mathbf{Z}^n \text{ and } a \cdot x = 0\}.$$

This collection of vectors, L_a, is a lattice and moreover, given the vector a, a basis for L_a can be found in polynomial time by using the algorithms of Kannan and Bachem [17].

Our approach to the SubsetSum problem follows. Wishing to determine whether there is a 0-1 vector x such that $a \cdot x = b$, we start by computing some particular integral solution x^0 to $a \cdot x = b$. An integer solution exists if and only if the greatest common divisor of all the components of a divides b. This criterion can be checked in polynomial time using the Euclidean algorithm.

Any 0-1 vector x such that $a \cdot x = b$ must be of the form $x^0 - y$, where y is a vector belonging to L_a. Since $x^0 - y$ is a 0-1 vector its length is at most $\sqrt{n}$. Thus, for a 0-1 solution to exist, there must be a point y in the lattice L_a that is at a distance at most $\sqrt{n}$ from x^0. Having found x^0 and a basis for L_a, we determine whether there is a vector y in L_a which is this close to x^0. It turns out that finding whether such a y exists is not hard if the shortest nonzero element of L_a is longer than $2n^{3/2}$. It also turns out that most a in $G_n(M)$, for large enough M, give rise to lattices L_a whose shortest nonzero vectors are longer than $2n^{3/2}$.

Let L be a lattice. The length of the shortest nonzero vector in L is denoted

$$\lambda_1(L) = \min \{|x|: x \in L \text{ and } x \neq 0\}$$

where $|\cdot|$ is Euclidean length.

LEMMA 1. *Recall that* $G_n(M) = \{(a_1, \cdots, a_n) \mid a_i \in \{1, \cdots, M\}\}$. *For any* k, *the length of the shortest vector in* L_a, $\lambda_1(L_a)$, *is greater than* k *for all but* $M^{n-1}(2k+1)^n$ *of the nonzero vectors* a *in* $G_n(M)$.

Proof. Consider a nonzero vector $v = (v_1, \cdots, v_n) \in \mathbf{Z}$ such that $|v_i| \leqq k$. Without loss of generality, assume $v_n \neq 0$. Suppose Suppose $a \in G_n(M)$ and $a \cdot v = 0$. Then

$$a_n v_n = -\sum_{i=1}^{n-1} a_i v_i,$$

and, hence,

$$a_n = \frac{-\sum_{i=1}^{n-1} a_i v_i}{v_n}.$$

Thus, the number of nonzero a's in $G_n(M)$ such that $a \cdot v = 0$ is at most M^{n-1}.

The number of vectors v in $\mathbf{Z}^n$ with components bounded in magnitude by k is $(2k+1)^n$. Therefore, at most $M^{n-1}(2k+1)^n$ of the elements a of $G_n(M)$ are such that $\lambda_1(L_a) \leqq k$. □

Remark 1. In the other direction, it is easy to argue using the pigeon-hole principle that every a in $G_n(M)$ has $\lambda_1(L_a) \leqq \sqrt{n} \cdot \lceil 1 + (Mn)^{1/(n-1)} \rceil$. Fix an a in $G_n(M)$. Let $k = [(Mn)^{1/(n-1)} + 1]$. The dot product of each v in $G_n(k)$ with a is an integer between 0 and Mnk. If any v in $G_n(k)$ makes a dot product 0 with a, then we will know that $\lambda_1(L_a) \leqq |v| \leqq k\sqrt{n}$ as desired. So assume this does not happen. Since $|G_n(k)| = k^n > Mnk$ (via a simple calculation), there must exist two distinct elements u, v in $G_n(k)$ such that $u \cdot a = v \cdot a$, whence $(v-u) \cdot a = 0$. Furthermore, $|u - v| \leqq \sqrt{n}k$, which proves the claim. A better bound can be obtained by observing that the determinant of L_a equals $|a|$, provided the gcd of $a_1, \cdots, a_n$ is 1. Thus, by Minkowski's convex body theorem [25, § 5], it follows that $\lambda_1(L_a) \leqq \sqrt{n} \cdot |a|^{1/n}$.

COROLLARY 1. *For* $M \geqq (20)^n n^{3n/2}$ *in the above lemma, the fraction of vectors* a *in* $G_n(M)$ *for which* $\lambda_1(L_a) \leqq 2n^{3/2}$ *is at most* $1/2^n$.

Proof. The number of vectors a in $G_n(M)$ is M^n. From the lemma we know that the number of a's with $\lambda_1(L_a) \leqq 2n^{3/2}$ is at most

$$M^{n-1}(4n^{3/2}+1)^n.$$

Thus, the fraction of such a's is at most

$$\frac{(4n^{3/2}+1)^n}{M} \leqq \frac{1}{2^n}.$$

□

We need some notation now. Suppose $b_1, b_2, \cdots, b_m$ is a basis of a lattice L_a (so $m = n-1$). Using the Gram-Schmidt orthonormalization process we may obtain from these b_i mutually orthogonal vectors $b_1^*, b_2^*, \cdots, b_m^*$ as follows:

$$b_1^* = b_1$$

and

$$b_{i+1}^* = b_{i+1} - \sum_{j=1}^{i} \mu_{i+1,j} b_j^* \quad \text{for } i = 1, \cdots, m-1$$

where

$$\mu_{kl} = \frac{b_k \cdot b_l^*}{b_l^* \cdot b_l^*} \quad \text{for } 1 \leqq l < k \leqq m.$$

These b_i^* are generally useful.

LEMMA 2. *Suppose $a = (a_1, \cdots, a_n) \in \mathbf{Z}^n$ and $\lambda_1(L_a) > 2n^{3/2}$. There exists a basis $b_1, \cdots, b_{n-1}$ of L_a such that we have the following:*

- $|b_i^*| > 2\sqrt{n}$, *for* $1 \leqq i \leqq n-1$, *and*
- *The vectors $b_1, \cdots, b_{n-1}$ can be expressed in polynomially many bits.*

Proof. Lagarias, Lenstra, and Schnorr [23, Thm. 5.1] show that for any m-dimensional lattice L, there is a basis $b_1, \cdots, b_m$ such that $|b_i^*| \geqq \lambda_1(L)/m$, and the b_i can be expressed in polynomially many bits. From this, Lemma 2 follows readily. □

A basis $b_1, \cdots, b_m$ of a lattice is *proper* if the μ_{ij} defined in the Gram-Schmidt process all satisfy $|\mu_{ij}| \leqq \frac{1}{2}$.

PROPOSITION 1. *Suppose L is an m-dimensional lattice given by a basis $b_1, \cdots, b_m$ satisfying*

$$|b_i^*| > 2k.$$

Let x be any given point in span (L). *There is at most one lattice point $v \in L$ such that $|x - v| \leqq k$. We can determine whether such a lattice point exists and if so, we can find it in polynomial time.*

Proof. First we show uniqueness. If v and v' are two *distinct* elements of L such that $|x - v|$ and $|x - v'|$ are less than or equal to k, then $|v - v'| \leqq 2k$ and $v - v' \neq 0$. Let $v - v' = \sum_{i=1}^{s} \alpha_i b_i$, $s \leqq m$, $\alpha_i \in \mathbf{Z}$, and $\alpha_s \neq 0$. Then $v - v'$ has a component $\alpha_s b_s^*$ along b_s^* of length $|\alpha_s||b_s^*| > 2k$, giving the needed contradiction.

Now we show how to compute the coordinates of a lattice vector $v \in L$ which is within a ball of radius k about x, if one exists. Suppose there is a lattice vector $v \in L$ such that $|x - v| \leqq k$. Write

$$v = \sum_{i=1}^{m} \alpha_i b_i$$

with $\alpha_i \in \mathbf{Z}$. Let

$$x = \sum_{i=1}^{m} \beta_i b_i^*$$

with $\beta_i \in \mathfrak{R}$. The projection of $x - v$ in the direction of b_m^* is of length $|\beta_m - \alpha_m||b_m^*|$. Thus,

$$|x - v| \leqq k$$

implies

$$|\beta_m - \alpha_m||b_m^*| \leqq k,$$

which means

$$|\beta_b - \alpha_m| \leqq \frac{k}{|b_m^*|} < \frac{k}{2k} = \frac{1}{2}.$$

Therefore, there is at most one possible integer value for α_m, namely $[\beta_m]$, the integer closest to β_m. Having found α_m, the coordinates $\alpha_{m-1}, \cdots, \alpha_1$ can be calculated in like fashion. □

2.1. Short proofs of infeasibility. Let $a = (a_1, \cdots, a_n)$ be an instance of a generic SubsetSum problem such that $\lambda_1(L_a) \geqq n^{3/2}$. Suppose B is an integer. If a subset of the a_i sums to B, a list of the coefficients that sum to B gives a short proof of that fact. Suppose no subset of the a_i sums to B. We describe how to write a short proof.

Begin the proof by exhibiting a *nice* basis $b_1, \cdots, b_m$ such that we have the following:

- $b_1, \cdots, b_m$ is a basis of L_a, and
- The inequalities $b_i^* > 2\sqrt{n}$ are satisfied.

(Such a nice basis of L_a is guaranteed to exist by Lemma 2.)

Then verify that $b_1, \cdots, b_m$ is a basis of L_a by doing the following:

(1) Checking that each b_i belongs to L_a; and also

(2) Checking that the determinant of the lattice spanned by the b_i equals the determinant of the lattice spanned by a basis of L_a found using the Hermite normal form algorithm of [17].

Next we write down any solution x^0 of x in $a \cdot x = b$; $x \in \mathbf{Z}^n$. It goes without saying that it is easy to check that a particular x^0 works. Observe that x^0 is not a 0-1 vector since we are assuming this SubsetSum problem has no 0-1 solution.

Following the deterministic algorithm given in the proof of Proposition 1, we find a unique $v \in L_a$, which is a candidate for satisfying $|v - x^0| \leqq \sqrt{n}$. If the original problem is infeasible, either the candidate v fails to satisfy this inequality or $v - x^0$ is not a 0-1 vector. This gives a polynomial-length proof in either case.

We now prove Theorem 1. By Corollary 1, if $M \geqq 2^{3/2 \log n + 5n}$, then at least $(1 - 1/2^N)|G_n(M)|$ such A have $\lambda_1(L_a) > n^{3/2}$. Then we can find the short proofs constructed above.

To summarize, with the *nice* basis on hand there is at most one easily identified v in L_a with $|x^0 - v|$ bounded above by $\sqrt{n}$. We determine whether such a v exists and if it does we check whether $x^0 - v$ has each component equal to 0 or 1. There is a solution to the subset sum problem if and only if this happens. This provides polynomial-length proofs of infeasibility and proves Theorem 1.

It is important to observe that this is not a polynomial-time *algorithm* for feasibility or infeasibility. Since the first step involves guessing a nice basis we have only given a proof that there are short proofs.

An algorithm. Under what conditions can the above proofs, or nondeterministic procedures, be made into deterministic polynomial-time algorithms? Examining Lagarias, Lenstra, and Schnorr's theorem [23] observe that they obtain a nice basis as follows. Start with a basis of the polar (dual) lattice and reduce it so that it is a "Korkine-Zolotarev-reduced" basis. The dual to this new basis is a nice basis of the original lattice. The best-known algorithm for finding such a "Korkine-Zolotarev-reduced" basis runs in exponential time, [18]. Furthermore, if a polynomial-time algorithm could be found for this problem, then the shortest vector problem for lattices could be solved in polynomial time. The latter question is open. It is perhaps the most important open question in the area of lattice algorithms.

In this way we obtain one more indication of the significance of the shortest vector problem (SVP). If SVP can be solved in polynomial time, then all the instances of the SubsetSum problem for which our theorems give a proof system would be determinisitic polynomial-time solvable.

In their famous paper, Lenstra, Lenstra, and Lovász [24] show how to approximate the length of the shortest vector in a lattice within a factor of $2^{n/2}$. That result can be used to obtain a deterministic algorithm for generic SubsetSum problems when the parameter M, which determines the number of bits in each coefficient, is large enough.

THEOREM 3. *Let $M \geqq (2^{n^2/2+2n})(n^{3n/2})$, i.e., $|M| = O(n^2)$. There is a deterministic algorithm A, and a polynomial p such that A solves all instances of the SubsetSum problem correctly and, for $(1-1/2^n)|G_n(M)|$ of the generic SubsetSum problems in $G_n(M)$, A runs in time bounded above by polynomial $p(n)$.*

Proof. Following the proof of Lemma 1, for most generic SubsetSum problems a in $G_n(M)$, the associated lattice L_a satisfies

$$\lambda_1(L_a) \geqq 2\sqrt{n}2^{(n-1)/2}.$$

If the Lenstra, Lenstra, and Lovász basis reduction algorithm is run on such a lattice L_a, a reduced basis $b_1, \cdots, b_{n-1}$ is produced. The vectors b_i in this representation of L_a satisfy

$$|b_i^*| \geqq 2\sqrt{n} \quad \text{for each } i.$$

Following the argument for our main theorem, the computed reduced basis can be used to solve SubsetSum problems derived from the generic problem a. Since the reduced basis may be computed in polynomial time, the whole algorithm is polynomial-time bounded. □

This provides a rigorous analysis of Brickell's algorithm [2].

3. Small circuits. The question of whether the proof system we have described contains short proofs of infeasibility for a given set of integer weights $a_1, \cdots, a_n$ and target b, depends only on the coefficients a_i. It is not difficult to show that for those $(a_1, \cdots, a_n)$'s that admit small proofs, there is also a polynomial-sized Boolean circuit $C(a_1, \cdots, a_n)$ that accepts precisely those b's that equal the sum of some subset of the a_i's. To see this, encode into the circuit $C(a_1, \cdots, a_n)$ the basis of the lattice L_a used in the proof of Theorem 2. The rest of the procedure inherent in the proofs are deterministic polynomial-time computable, so they can be simulated by the polynomial-sized circuit. The details are omitted.

THEOREM 4. *Suppose $M \geqq 2^{3/2n \log n+5n}$. There is a subset S of $G_n(M)$ of cardinality at least $(1-1/2^n)|G_n(M)|$ such that, for each $a \in S$, there is a polynomial-sized circuit that accepts precisely those b's that are the sum of a subset of the a_i.*

The reason this result is interesting is the following. We know that the SubsetSum problem is NP-complete. So, if M is any polynomial-time bounded nondeterministic

Turing machine, there is a many-one reduction f of $L(M)$ to the SubsetSum problem. For any string x, $f(x)$ has two parts: the SubsetSum coefficients $a_1, \cdots, a_n$ and the right-hand side integer b. It can be shown that the number n and the coefficients a_i depend only on the machine M and the length of x. The input x only specifies the target b. Getting a small circuit C is thus equivalent to finding a small circuit for another language in NP. We do not, at present, know of other natural languages in NP for which this works.

4. Previous results. A survey of results related to solving almost all instances of NP-complete problems is found in Johnson's NP-completeness column [16]. Here we discuss several of them.

4.1. Hamilton cycles. A considerable study has been made on the subject of Hamiltonicity in random graphs [30], [21], [1]. The culmination of which is an expected polynomial-time algorithm for the Hamilton cycle problem in random graphs having enough edges [3]. The elegant paper of Bollobás, Fenner, and Frieze demonstrates an expected polynomial-time bound on the run-time of a straightforward Hamiltonicity algorithm.

Prior to their work it was known that random graphs with many edges almost surely contain Hamilton cycles. What Bollobás, Fenner, and Frieze show is that a fairly simple algorithm actually finds Hamilton cycles almost all the time. However, for each graph that is non-Hamiltonian, the algorithm takes exponential time before it can positively conclude non-Hamiltonicity. It is useful in practice. However, it never provides nontrivial proofs of non-Hamiltonicity.

4.2. Satisfiability. There has been some work on algorithms for solving random instances of the satisfiability problem. Suppose $G(n, m, p)$ denotes the class of CNF-formulae with n random clauses and m variables generated as follows. For each variable v_i, $1 \leqq i \leqq m$, a random clause contains the literal v_i, or the literal $\bar{v}_i$, or neither with probabilities p, p, $1-2p$, respectively. (Obviously, $0 \leqq p \leqq \frac{1}{2}$.) The choices are made independently for each variable and the clauses are generated independently. Goldberg, Purdom, and Brown [12] show that a version of the Davis–Putnam procedure [9] runs in expected polynomial time on such problems. The expected time is polynomially bounded in n, m but it is exponential in $1/p$.

Following the analysis of Franco and Paull [10], it is possible to show that there are constants c, $d > 0$ depending only on p such that if n, the number of clauses, is at most 2^{cm} (m was the number of variables), then for almost all CNF-formulae, the number of truth assignments that make the formula false is at most $2^{(1-d)m}$. Almost any truth assignment renders the formula true in this case; thus, the Davis–Putnam procedure does not yield proofs of nonsatisfiability for very many instances. In the other case, when $n > 2^{cm}$, the value 2^m is polynomially bounded in n. Therefore, the satisfiability of the formula can be checked in polynomial time by enumerating all the 2^m truth assignments. It is only in this case, once again by doing enumeration, that proofs of unsatisfiability are seen.

4.3. Vertex coloring. The problem of three-coloring (in general k-coloring for any fixed k) a random graph has been considered by Wilf [32]. If each edge of a graph is put in with probability $p > 0$, then for a large enough number of vertices, with high probability, there will be a clique of size four in the graph. Thus, most "random" graphs contain four-cliques and cannot be colored with three colors. Unlike the other problems discussed, in this case nonmembership in the NP-complete language can be proved for most instances. However, nonmembership is for an absolutely local reason.

4.4 Minimum bisection width. Determining the minimum number of edges that must be removed from a graph in order to separate the vertices into two equal-sized sets is NP-complete. Bui et al. [4] suggest an algorithm for minimum bisection width that is based on min-cut max-flow methods. Their algorithm always runs in polynomial time. For almost all regular, degree d graphs that have minimum bisection width bounded above by $o(n^{1-1/\lfloor (d+2)/2 \rfloor})$, their algorithm calculates the correct bisection width. However, the class of graphs the algorithm works for is a vanishing fraction of the class of all graphs on n vertices. Moreover, membership in the class is itself NP-hard. Thus, their algorithm is providing proofs of nonmembership for a large fraction of a noncomprehensive set.

REFERENCES

[1] D. ANGLUIN AND L. VALIANT, *Fast probabilistic algorithms for Hamilton circuits and matchings*, in Proc. 9th Annual ACM Symposium on Theory of Computing, 1977, pp. 30–41.

[2] E. BRICKELL, *Are low-density knapsacks solvable in polynomial time?*, Congress. Numer., 39 (1983), pp. 145–156.

[3] B. BOLLOBÁS, T. I. FENNER, AND A. M. FRIEZE, *An algorithm for finding Hamilton cycles in random graphs*, in Proc. 26th Annual ACM Symposium on Theory of Computing, 1985.

[4] T. BUI, S. CHAUDHURI, T. LEIGHTON, AND M. SIPSER, *Graph bisection algorithms with good average case behavior*, in Proc. 25th Annual ACM Symposium on Theory of Computing, 1984, pp. 181–192.

[5] J. W. S. CASSELS, *An Introduction to Geometry of Numbers*, Springer-Verlag, Berlin, 1971.

[6] V. CHVÁTAL, *Determining stability number of a graph*, SIAM J. Comput., 6 (1977), pp. 643–662.

[7] ———, *Hard knapsack problems*, Oper. Res. 28 (1980), pp. 1402–1411.

[8] S. A. COOK AND R. A. RECKHOW, *The relative efficiency of propositional proofs systems*, J. Symbolic Logic, 44 (1979), pp. 36–50.

[9] M. DAVIS, G. LOGEMANN, AND D. LOVELAND, *A machine program for theorem proving*, Comm. ACM, 5 (1962), pp. 394–397.

[10] J. FRANCO AND M. PAULL, *Probabilistic analysis of the Davis–Putnam procedure for solving the satisfiability problem*, Discrete Appl. Math., 5 (1983), pp. 77–87.

[11] A. M. FRIEZE, *On the Lagarias–Odlyzko algorithm for the subset sum problem*, SIAM J. Comput., to appear.

[12] A. GOLDBERG, P. PURDOM, AND C. BROWN, *Average time analysis of simplified Davis–Putnam procedures*, Inform. Process. Lett., 15 (1982), pp. 72–75.

[13] M. GRÖTSCHEL, *On the symmetric travelling salesman problem*, Math. Programming Stud., 12 (1980), pp. 61–77.

[14] M. GRÖTSCHEL AND M. W. PADBERG, *On the symmetric travelling salesman problem*, Math. Programming, 16 (1979), pp. 265–302.

[15] A. HAKEN, *The intractability of resolution*, Ph.D. thesis, Department of Mathematics, University of Illinois, Urbana, IL, 1982.

[16] D. S. JOHNSON, *The* NP*-completeness column: an ongoing guide*, J. Algorithms, 5 (1984), pp. 284–299.

[17] R. KANNAN AND A. BACHEM, *Polynomial-time algorithms for computing the Smith and Hermite normal forms of an integer matrix*, SIAM J. Comput., 8 (1979), pp. 499–507.

[18] R. KANNAN, *Improved algorithms for integer programming and related lattice problems*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 193–206.

[19] ———, *Algorithmic geometry of numbers*, Tech. Report Institute for Operations Research, University of Bonn, Bonn, FRG; Ann. Rev. Comput. Sci., (1987), pp. 231–267.

[20] R. M. KARP, *Reducibility among combinatiorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press NY, 1972, pp. 85–103.

[21] ———, *The probabilisitic analysis of some combinatorial search algorithms*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, NY, 1976, pp. 1–19.

[22] J. C. LAGARIAS AND A. M. ODLYZKO, *Solving low-density subset sum problems*, in Proc. 24th Annual IEEE Symposium on Computer Science, 1983, pp. 1–10.

[23] J. C. LAGARIAS, H. W. LENSTRA, AND C. SCHNORR, *Korkhine–Zolotarev bases and successive minima of a lattice and its reciprocal lattice*, Combinatorica, to appear.

[24] A. K. LENSTRA, H. W. LENSTRA, AND L. LOVÁSZ, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 513–534.

[25] C. G. LEKKERKERKER, *Geometry of Numbers*, North-Holland, Amsterdam, 1969.
[26] H. W. LENSTRA, *Integer programming with a fixed number of variables*, Math. Oper. Res., 8 (1983), pp. 538–548.
[27] L. LOVÁSZ, *An algorithmic theory of numbers, graphs and convexity*, CBMS–NSF Regional Conference Series in Applied Mathematics 50, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.
[28] C. J. H. MCDERMITT, *Determining the chromatic number of a graph*, SIAM J. Comput., 8 (1977), pp. 1–14.
[29] P. MILIOTIS, *Integer programming obstacles to the travelling salesman problem*, Math. Programming, 10 (1976), pp. 367–378.
[30] L. POSA, *Hamilton circuits in random graphs*, Discrete Math., 14 (1976), pp. 359–364.
[31] J. A. ROBINSON, *A machine oriented logic based on the resolution principle*, J. Assoc. Comput. Mach., 12 (1965), pp. 23–41.
[32] H. S. WILF, *An $O(1)$ expected time graph coloring algorithm*, Inform. Process. Lett., to appear.

SIAM J. COMPUT.
Vol. 18, No. 3, p. 559-578, June 1989

009

TWO APPLICATIONS OF INDUCTIVE COUNTING FOR COMPLEMENTATION PROBLEMS*

ALLAN BORODIN†, STEPHEN A. COOK†, PATRICK W. DYMOND‡, WALTER L. RUZZO§, AND MARTIN TOMPA¶

Abstract. Following the recent independent proofs of Immerman [*SIAM J. Comput.*, 17 (1988), pp. 935-938] and Szelepcsényi [*Bull. European Assoc. Theoret. Comput. Sci.*, 33 (1987), pp. 96-100] that nondeterministic space-bounded complexity classes are closed under complementation, two further applications of the inductive counting technique are developed. First, an errorless probabilistic algorithm for the undirected graph s-t connectivity problem that runs in $O(\log n)$ space and polynomial expected time is given. Then it is shown that the class *LOGCFL* is closed under complementation. The latter is a special case of a general result that shows closure under complementation of classes defined by semi-unbounded fan-in circuits (or, equivalently, nondeterministic auxiliary pushdown automata or tree-size bounded alternating Turing machines). As one consequence, it is shown that small numbers of "role switches" in two-person pebbling can be eliminated.

Key words. complementation, inductive counting, connectivity, symmetric computation, probabilistic algorithm, random walk, *LOGCFL*, *NC*, semi-unboundedness, pebbling, hierarchy

AMS(MOS) subject classifications. 68Q15, 68Q25, 68Q75, 68R10, 60J15, 94C99, 03D55, 90D05

1. Introduction.

Let *NL* denote the class of languages accepted by nondeterministic Turing machines running in $O(\log n)$ space. Let

$$STCON = \{(G, s, t) \mid G \text{ is a directed graph containing a directed path from vertex } s \text{ to vertex } t\}.$$

Using any reasonable encoding of graphs, it is well known that *STCON* is in *NL* and, moreover, is complete for *NL* with respect to deterministic log space reductions (Savitch [29]). In a surprising development, Immerman [17] and Szelepcsényi [33] have shown independently that $\overline{STCON}$, the complement of *STCON*, is also in *NL*; that is, *NL* is closed under complementation.

Their proofs rely on an inductive counting technique similar to counting techniques used in related results, for instance, Mahaney [24], Lange, Jenner, and Kirsig [22], Hemachandra [15], Toda [34], Buss, Cook, Dymond, and Hay [4], and Schöning and Wagner [30]. (For additional background and references see Hartmanis [14].) It seems inevitable that this technique should have further application and in this paper we develop two such applications.

For our first application, we consider reachability in undirected graphs, a problem

* Received by the editors October 28, 1987; accepted for publication (in revised form) May 26, 1988. This material is based on work supported in part by the Natural Sciences and Engineering Research Council of Canada, and by the National Science Foundation under grants DCR-8604031 and CCR-8703196. Much of the work was performed while Allan Borodin and Larry Ruzzo were visitors at the IBM Thomas J. Watson Research Center, and Patrick Dymond was a visitor at the University of Toronto.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4.

‡ Department of Computer Science and Engineering, C-014, University of California at San Diego, La Jolla, California 92093.

§ Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195.

¶ IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.

not known to be complete for *NL*. Indeed, Aleliunas et al. [2] have shown that

$$USTCON = \{(G, s, t) \mid G \text{ is an undirected graph containing a path from vertex } s \text{ to vertex } t\}$$

is in *RLP*, the class of languages accepted by probabilistic Turing machines running in $O(\log n)$ space and polynomial time. In this model, the machine never reaches an accepting state on any input not in *USTCON*, and has probability at least $\frac{1}{2}$ of reaching an accepting state on any input in *USTCON*. *USTCON* is a complete problem for *SL*, the class of languages accepted by symmetric Turing machines (Lewis and Papadimitriou [23]) running in $O(\log n)$ space. It follows that $SL \subseteq RLP \subseteq NL$.

It is tempting to believe that Immerman's and Szelepcsényi's proofs extend to show that *SL* is closed under complementation. However, a direct translation of their technique fails to establish this result, as explained in § 2.2. In that section we prove the somewhat weaker result that $\overline{USTCON}$ is in *RLP* or, equivalently, that *USTCON* is in *ZPLP*, the class of languages accepted by errorless probabilistic Turing machines running in $O(\log n)$ space and polynomial expected time. (The equivalence is due to the fact that $RLP \cap \text{co}RLP = ZPLP$.) This answers a problem raised by Aleliunas et al. [2]. In § 2.3 we extend this result to show that Reif's symmetric log space complementation hierarchy [26] is also contained in *ZPLP*.

In our second application we show the closure under complementation of a number of complexity classes that are (seemingly) more powerful than *NL*. The classes we consider may be characterized in terms of several different models. The most intuitively appealing model is perhaps the semi-unbounded fan-in circuit model (see Venkateswaran [37]). In this model, we allow OR gates with arbitrary fan-in, whereas all AND gates have bounded fan-in. Input variables and their negations are supplied, but negations are prohibited elsewhere.

For simplicity we will restrict the discussion to polynomial-size circuits, although the results can be generalized. Of particular interest is the class SAC^k of languages accepted by polynomial-size, $O(\log^k n)$ depth, uniform semi-unbounded fan-in circuits. (See Cook [7] for an appropriate definition of uniformity.) SAC^1 is the most often studied of these classes. SAC^1 is equal to the class *LOGCFL* of languages log space reducible to context-free languages [32], [37]. It is known that $NC^1 \subseteq NL \subseteq SAC^1 \subseteq AC^1$ where, as is customary, we let NC^k and AC^k denote the classes analogous to SAC^k for bounded fan-in and (respectively) unbounded fan-in uniform circuits. More generally, $NC^k \subseteq SAC^k \subseteq AC^k \subseteq NC^{k+1}$.

Both NC^k and AC^k are easily seen to be closed under complementation by application of De Morgan's laws. However, SAC^k does not yield to the same technique, since it would produce circuits with unbounded fan-in AND gates. In fact, it is known that there is a language accepted by polynomial size, constant depth, uniform semi-unbounded fan-in circuits, but whose complement is not accepted by semi-unbounded fan-in circuits of depth $o(\log n)$ and arbitrary size, even nonuniformly (Venkateswaran [36], [37]). The main result of § 3.1 is to show that polynomial-size semi-unbounded circuit classes *are* closed under complementation for all depths that are $\Omega(\log n)$. Closure under complementation of classes defined by auxiliary pushdown automata, tree-size bounded alternating Turing machines, and simple first-order formulae then follows by known equivalences (see § 3.1).

In § 3.2, we examine some consequences of the closure of semi-unbounded circuit classes under complementation. In the same way that the alternating and oracle hierarchies based on *NL* [5], [28] collapse because of Immerman's and Szelepcsényi's result, hierarchies based on semi-unbounded circuit classes also collapse. As a con-

sequence, the "role switch" resource in the pebble game introduced by Venkateswaran and Tompa [38] is shown to be much weaker than previously seemed plausible. For instance, we demonstrate that any fixed number of role switches can be eliminated.

2. Errorless algorithms related to undirected reachability.

2.1. Probabilistic complexity classes. An explanation of our taxonomy for probabilistic complexity classes is in order. Table 1 illustrates the sense in which the names *RLP* and *ZPLP* encountered in § 1 are consistent with the notation *ZPP*, *BPP*, and *PP* of Gill [12], *RP* of Welsh [40], and *BPL* and *PL* of Ruzzo, Simon, and Tompa [28].

TABLE 1
Taxonomy for probabilistic complexity classes.

Type of error	Polynomial expected time	$O(\log n)$ space	$O(\log n)$ space and polynomial expected time
Zero-sided	*ZPP*	*ZPL*	*ZPLP*
One-sided	*RP*	*RL*	*RLP*
Bounded two-sided	*BPP*	*BPL*	*BPLP*
Unbounded two-sided	*PP*	*PL*	*PLP*

Since this section concentrates on the classes *RLP* and *ZPLP*, we give them careful definitions here. We say that a language A is in *RLP* if and only if there is a probabilistic Turing machine M such that

(1) For all inputs, M runs in space $O(\log n)$ and expected time $n^{O(1)}$,

(2) If $w \in A$, $\Pr[M \text{ reaches an accepting state on input } w] \geqq \frac{1}{2}$, and

(3) If $w \notin A$, $\Pr[M \text{ reaches an accepting state on input } w] = 0$.

ZPLP is the class obtained by replacing condition 2 by 2′:

(2′) If $w \in A$, $\Pr[M \text{ reaches an accepting state on input } w] = 1$.

The class *RLP* remains unchanged if we require polynomial time rather than just polynomial expected time. Results of Gill [12], Immerman [17], and Szelepcsényi [33] show that, when the polynomial time bound is removed, the corresponding one-sided (*RL*) and zero-sided (*ZPL*) classes are equal to *NL*. Ruzzo, Simon, and Tompa [28] and Simon [31] have shown that *PL* and *BPL* are closed under complementation. Jung [20] has shown that $PL = PLP$. These relations and others (including those proved in this paper) are summarized in Fig. 1. (Those complexity classes whose complements are not explicitly shown in Fig. 1 are closed under complementation. *DL* denotes $DSPACE(\log n)$. *DET* is the set of languages reducible to computing integer matrix determinants [7]. $\bigcup_k C\sum_k^{SL}$ is the symmetric log space hierarchy, discussed in § 2.3.)

2.2. An errorless algorithm for undirected reachability. Immerman's and Szelepcsényi's proofs that $\overline{STCON} \in NL$ rely on computing

$$N_k = \#\{v \mid v \text{ is within distance } k \text{ of } s\}$$

by induction on k. As mentioned in § 1, it is tempting to believe that the same method can be used to show that *SL* is closed under complementation. Perhaps the easiest way to see the difficulty (and importance) of such a result is to observe that Immerman's and Szelepcsényi's algorithm also computes the length of a shortest path from s to t. By a known reduction (Ladner (personal communication)), *STCON* is log space reducible to the problem of determining if the length of a shortest path from s to t in an undirected graph is $n-1$. Hence a direct translation of Immerman's and Szelepcsényi's proof to *SL* would also solve this problem, thus implying $SL = RLP = NL$.

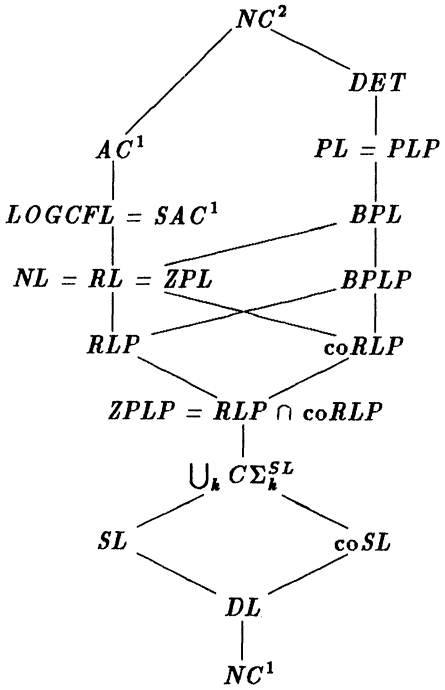


FIG. 1. *Inclusion relations among* $O(\log n)$ *space bounded complexity classes.*

The specific obstacle in applying Immerman's and Szelepcsényi's proof technique to *SL* is that a symmetric computation cannot "nondeterministically count," which seems to be the key feature in their method. (This was also noted in [16].) In particular, suppose that, for any value of *count*, there is a configuration (*R*, *count*) from which the computation can proceed nondeterministically on either of the following paths:

(1) $(R, count) \vdash^* (R', count)$.

(2) $(R, count) \vdash^* (R', count+1)$.

Since all moves in a symmetric machine are reversible, the machine can realize the computation sequence $(R, count) \vdash^* (R', count+1)\ {}^*\!\dashv (R, count+1)$. Hence the *count* can be increased (or decreased) arbitrarily.

The main result of this section is Theorem 1.

THEOREM 1. *USTCON* $\in$ *ZPLP* (*or, equivalently,* $SL \subseteq ZPLP$).

This theorem will follow immediately from Lemma 6.

In proving Theorem 1 we no longer face the handicap of symmetric computations discussed above, but we do face another difficulty: the random walk approach of Aleliunas et al. [2] does not seem to provide any useful information on the distance between vertices. To circumvent our inability to compute undirected distances with an *RLP* computation, we use the following idea, which is basic to the algorithms of Kleene [21], Floyd [9], and Warshall [39]. Let $u \to_k^* v$ denote the existence of a path between u and v that does not pass through any intermediate vertices l with $l > k$. Let

$$P_k = \{(u, v) \mid u \to_k^* v\}.$$

The *ZPLP* algorithm for *USTCON* is outlined in the remainder of this paragraph, and described fully thereafter. It proceeds similarly to Immerman's and Szelepcsényi's algorithm, by iteratively computing $\# P_k$, the cardinality of P_k. Having computed $\# P_n$,

we alternately attempt to verify that $(G, s, t) \in USTCON$ via a random walk, or that $(G, s, t) \in \overline{USTCON}$ by verifying that there are $\# P_n$ pairs $(u, v) \neq (s, t)$ that are in P_n. We compute $\# P_k$ from $\# P_{k-1}$ by determining, for each pair (u, v), whether or not it is in P_k. We note that $(u, v) \in P_k$ if and only if either $(u, v) \in P_{k-1}$, or both $(u, k) \in P_{k-1}$ and $(k, v) \in P_{k-1}$. Again, $\# P_{k-1}$ is used to insure that the algorithm never makes a mistake when claiming some (g, h) is or is not in P_{k-1}.

The following lemma is obvious.

LEMMA 2. *$(u, v) \in P_k$ if and only if u and v are in the same connected component of the subgraph $G(k, u, v)$ of G induced by vertices $\{1, 2, 3, \cdots, k, u, v\}$.*

The basis for our algorithm is the random walk technique as used in Aleliunas et al. [2]. It is modified slightly for our purposes in the following algorithm.

ALGORITHM *WALK*(k, u, v).
comment: *WALK*(k, u, v) simulates a random walk on the subgraph $G(k, u, v)$ starting at u, returns *FOUND* if v is encountered, and *NOT FOUND* otherwise;
begin
  if $u = v$ **then return** *FOUND*;
  $n \leftarrow \# V$;
  $x \leftarrow u$;
  repeat $2n^3 \log_2 (2n^2)$ **times**
    begin
      choose y randomly and uniformly from among the neighbors of x in $G(k, u, v)$;
      if $y = v$ **then return** *FOUND* **else** $x \leftarrow y$
    end;
  return *NOT FOUND*
end.

LEMMA 3. (1) *If* $(u, v) \in P_k$, $\Pr[WALK(k, u, v) = FOUND] \geqq 1 - 1/2n^2$.
(2) *If* $(u, v) \notin P_k$, $\Pr[WALK(k, u, v) = FOUND] = 0$.
(3) *WALK*(k, u, v) *uses space* $O(\log n)$.
(4) *WALK*(k, u, v) *uses expected time* $O(n^5 \log n)$ *(on a probabilistic Turing machine).*

Proof. The main result of Aleliunas et al. [2] is that the expected number of edge traversals a random walk requires to visit all vertices of a connected undirected graph, beginning at any vertex, is at most n^3. By Markov's inequality [3] and Lemma 2, the probability is at most $\frac{1}{2}$ that *WALK*(k, u, v) does not encounter v within any specified $2n^3$ iterations of the **repeat** loop, given that $(u, v) \in P_k$. The correctness assertions follow from this.

For the time complexity, we assume that G is presented as an $n \times n$ adjacency matrix. Locating the row of this matrix corresponding to x, computing the degree in $G(k, u, v)$ of x, and selecting a neighbor y can be done in $O(n^2)$ time. There is a technical detail if we assume $\{0, 1\}$ valued probabilistic choices when the degree need not be a power of 2. Suppose $2^r \leqq \text{degree}(x) < 2^{r+1}$. We then choose $r+1$ random bits to compute a random integer $i \in [0, 2^{r+1} - 1]$. If $i > \text{degree}(x)$ we discard i and try again. The expected number of random integers i that need be generated (to obtain $i \leqq \text{degree}(x)$) is at most 2. Thus *WALK*(k, u, v) uses expected time $O(n^2 n^3 \log n) = O(n^5 \log n)$. □

If $\# P_k$ is known exactly, we need an errorless probabilistic algorithm that determines whether or not (u, v) is in P_k.

ALGORITHM *PATH*(k, cpk, u, v).
comment: *PATH*$(k, \#P_k, u, v)$ returns *TRUE* iff $(u, v) \in P_k$;
repeat forever
begin
if *WALK*$(k, u, v) = FOUND$ **then return** *TRUE*;
$c \leftarrow 0$;
for all $(g, h) \neq (u, v)$ **do**
if *WALK*$(k, g, h) = FOUND$ **then** $c \leftarrow c + 1$;
if $c = cpk$ **then return** *FALSE*
end.

LEMMA 4. (1) *If PATH*$(k, \#P_k, u, v)$ *returns TRUE, then* $(u, v) \in P_k$.
(2) *If PATH*$(k, \#P_k, u, v)$ *returns FALSE, then* $(u, v) \notin P_k$.
(3) *PATH*$(k, \#P_k, u, v)$ *uses space* $O(\log n)$.
(4) *PATH*$(k, \#P_k, u, v)$ *uses expected time* $O(n^7 \log n)$.

Proof. The correctness and space complexity of *PATH* are immediate from Lemma 3. For the expected time, note that each iteration of *PATH's* **repeat** loop uses at most n^2 calls to *WALK*, that is, expected time $O(n^7 \log n)$. The probability that a given iteration of *PATH* fails to return a value is equal to the probability that an incorrect answer is given by one or more of its invocations to *WALK*, which is at most $n^2/2n^2 = \frac{1}{2}$, by Lemma 3. Hence, the expected number of iterations is at most 2. □

We now show how to extend $\#P_{k-1}$ to $\#P_k$.

ALGORITHM *COUNT*$(k, cpkm1)$.
comment: *COUNT*$(k, \#P_{k-1})$ returns the correct value for $\#P_k$;
begin
$cpk \leftarrow 0$;
for all (u, v) **do**
if *PATH*$(k-1, cpkm1, u, v)$
or (*PATH*$(k-1, cpkm1, u, k)$ **and** *PATH*$(k-1, cpkm1, k, v)$)
then $cpk \leftarrow cpk + 1$;
return *cpk*
end.

LEMMA 5. (1) *COUNT*$(k, \#P_{k-1})$ *returns* $\#P_k$.
(2) *COUNT*$(k, \#P_{k-1})$ *uses space* $O(\log n)$.
(3) *COUNT*$(k, \#P_{k-1})$ *uses expected time* $O(n^9 \log n)$.

Proof. By Lemma 4, the calls to *PATH* correctly determine whether or not $(u, v), (u, k), (k, v)$ are in P_{k-1}. Correctness follows since $(u, v) \in P_k$ if and only if $(u, v) \in P_{k-1}$ or $((u, k) \in P_{k-1}$ and $(k, v) \in P_{k-1})$. For the time complexity, there are $O(n^2)$ invocations of *PATH*, each of which runs in expected time $O(n^7 \log n)$, by Lemma 4. □

It only remains to state the main routine.

ALGORITHM *USTCON*(G, s, t).
begin
comment: Initialize $\#P_0$: $P_0 = \{(u, u)\} \cup \{(u, v) | \{u, v\} \in E\}$;
$cpk \leftarrow \#V + 2\#E$;
for k **from** 1 **to** $\#V$ **do** $cpk \leftarrow$ *COUNT*(k, cpk);
comment: *cpk* is now set to $\#P_n$;
return *PATH*$(\#V, cpk, s, t)$
end.

LEMMA 6. (1) *USTCON*(G, s, t) returns *TRUE* if and only if $(G, s, t) \in$ *USTCON*.

(2) *USTCON* uses space $O(\log n)$.

(3) *USTCON* uses expected time $O(n^{10} \log n)$.

Proof. This follows immediately from Lemmas 4 and 5. □

It is interesting to compare the running time of the errorless algorithm (Lemma 6) to that of the version with one-sided error (Lemma 3).

2.3. An errorless algorithm for symmetric space hierarchies. As previously mentioned, *USTCON* is a complete problem for *SL*, the class of languages accepted by nondeterministic $O(\log n)$ space machines whose next move relation is symmetric. Reif [26] defined an "alternating" hierarchy based on *SL* in a manner analogous to the alternating hierarchy based on *NL* defined by Chandra, Kozen, and Stockmeyer [5]. While Immerman and Szelepcsényi's result shows that the *NL*-based hierarchy collapses to *NL*, the *SL* hierarchy may be infinite. For example, "bounded degree planarity" is in the hierarchy but is not known to be in *SL* [26]. The main result of this section is Theorem 9, which extends Theorem 1 by showing that the entire *SL* hierarchy is in *ZPLP*.

For technical reasons related to the problem of nondeterministic counting discussed in § 2.2, Reif's hierarchy is defined in terms of Turing machines with complementing moves, rather than existential and universal states as is standard for alternating machines. In Reif's complementing machines, a configuration p_0 is "accepting" if and only if there is a finite computation sequence $p_0 \vdash p_1 \vdash p_2 \cdots \vdash p_j, j \geqq 0$, with no complementing moves such that either

(1) p_j is in an accepting state, or

(2) there is at least one complementing move from p_j and for all complementing moves (p_j, p'), p' is not "accepting."

In a symmetric complementing machine, all noncomplementing moves must be symmetric. The kth level of the symmetric complementation hierarchy is

$$C\Sigma_k^{SL} = \{B \mid B \text{ is accepted by an } O(\log n) \text{ space-bounded symmetric complementing machine making at most } k-1 \text{ complement moves on any computation sequence}\}.$$

The following result is an easy modification of Theorem 5 in Ruzzo, Simon, and Tompa [28]. (See that reference for a discussion of space-bounded oracle machines.)

LEMMA 7. $\bigcup_k C\Sigma_k^{SL} \subseteq DL^{SL}$.

Proof. Consider a $C\Sigma_k^{SL}$ computation. Let $E(p)$ be the set of configurations q reachable from p using only noncomplementing moves. Note that an *SL* oracle can decide if $q \in E(p)$, since all noncomplementing moves are symmetric. Let $ACC(p, k) =$ *TRUE* if and only if there is a $p_j \in E(p)$ such that either

(1) p_j is an accepting state, or

(2) $k \geqq 1$, and there is a complementing move from p_j, and $\neg ACC(p', k-1)$ for all complementing moves (p_j, p').

If p_0 is the initial configuration, then the $C\Sigma_k^{SL}$ computation accepts if and only if $ACC(p_0, k-1) =$ *TRUE*. We determine the existence of p_j by deterministically enumerating all configurations and calling the appropriate *SL* oracle. The recursive calls on *ACC* are tested by maintaining a stack of k configurations. □

Ruzzo, Simon, and Tompa [28] use the notation $A^{\langle B \rangle}$ to denote a restricted form of relativization in which the query tape is subject to the relativized machine's space

bound, but the oracle receives that machine's input together with each query. This restriction is required to ensure the space bound in the following simulation.

LEMMA 8. $ZPLP^{\langle ZPLP\rangle} = ZPLP$.

Proof. Consider a relativized *ZPLP* machine M_1 that uses an oracle A accepted by a *ZPLP* machine M_2. Let w be the input to M_1. $M_1^{\langle A\rangle}$ is simulated by a *ZPLP* machine M_3. M_3 operates as if it were M_1 until M_1 invokes the oracle with some query x. M_3 then simulates M_2 on input $w\$x$. Whenever M_2 decides its output, M_3 is able to continue its simulation of M_1. If $M_1(M_2)$ runs in expected time p_1 (respectively, p_2) then the expected time of M_3 is $O(p_1(n)p_2(n+O(\log n))) = n^{O(1)}$. □

THEOREM 9. $\bigcup_k C\Sigma_k^{SL} \subseteq ZPLP$.

Proof. By Theorem 1 and Lemmas 7 and 8,

$$\bigcup_k C\Sigma_k^{SL} \subseteq DL^{SL} \subseteq DL^{ZPLP} \subseteq ZPLP^{\langle ZPLP\rangle} = ZPLP.$$

The third inclusion follows from the fact that the deterministic oracle machine can be assumed to write short queries, namely its configuration as it is about to write a long query. (See [28, Lemma 7] for more details.) □

This improves Reif's result that $\bigcup_k C\Sigma_k^{SL} \subseteq BPLP$ (defined in § 2.1).

Reif also considered the implications of his result for probabilistic parallel models. Simulation of *DL* by $O(\log n)$ time parallel models such as concurrent-read, exclusive-write, parallel random access machines (CREW-PRAMs) [10] and hardware modification machines [8] was well known. It has been observed (see, for example, Reif [25], [26]) that these simulations extend to the simulation of *RLP* and *BPLP* by one-sided and (respectively) bounded two-sided error probabilistic parallel machines. Reif noted that, as a corollary, any language in $\bigcup_k C\Sigma_k^{SL}$ can be recognized by such a probabilistic parallel machine with bounded two-sided error in $O(\log n)$ time.

Theorem 9 improves this result also, since any language in *ZPLP* can be accepted by an errorless probabilistic hardware modification machine (and thus by an errorless probabilistic CREW-PRAM) in expected time $O(\log n)$ using polynomially many processors. The simulation of an expected time $p(n)$ *ZPLP* algorithm proceeds as follows. Simulate $2p(n)$ steps of the *ZPLP* algorithm in time at most $c \log n$ using at most $(p(n))^c$ processors (for some constant c) as in Reif [25], [26]. If the simulated algorithm has not halted within that time, restart the simulation, using independently chosen random moves. The expected number of repetitions of this procedure is at most two.

Finally, an oracle hierarchy based on *SL* could be defined in the same manner as the $O\Sigma_k^L$ hierarchy of Ruzzo, Simon, and Tompa [28]. If we are careful about the definition of a symmetric oracle machine (see [16] for one possible definition), we would expect to find that, for all oracles A, $SL^{\langle A\rangle} \subseteq ZPLP^{\langle A\rangle}$ and thus by induction that $\bigcup_k O\Sigma_k^{SL} \subseteq ZPLP$. However, we have not pursued this question.

3. Semi-unbounded circuits, LOGCFL, and pebbling.

3.1. Complementation of semi-unbounded fan-in circuits. In this section we show closure under complementation of the class of languages accepted by semi-unbounded fan-in circuits.

The class of languages recognizable by size- and depth-bounded semi-unbounded fan-in circuits has been characterized in terms of several other models. The oldest is the nondeterministic auxiliary pushdown automaton of Cook [6]. Ruzzo has related space and time on such machines to space and *tree-size* of alternating Turing machines, where tree-size is the number of nodes in the smallest accepting subtree of the computation tree [27]. Venkateswaran has related them to space and alternations on

semi-unbounded alternating Turing machines—ones where there are no two consecutive universal configurations on any path in the computation [37]. The relation to semi-unbounded fan-in circuits follows from this. Immerman has shown relations to uniform families of short first-order formulae with a fixed number of variables and Boolean universal quantifiers—a property analogus to the semi-unbounded fan-in restriction [18]. These equivalences are summarized in the propositions below. They also generalize to larger space bounds.

PROPOSITION 10 [18], [27], [37]. *For* $T(n) = \Omega(\log n)$, *the following* (*uniform*) *complexity classes are identical.*

(1) NauxPDA Space, Time $(O(\log n), 2^{O(T(n))})$.

(2) ATM Space, Tree-size $(O(\log n), 2^{O(T(n))})$.

(3) Semi-Unbounded ATM Space, Alternations $(O(\log n), O(T(n)))$.

(4) Uniform Semi-Unbounded Fan-in Circuit Size, Depth $(2^{O(\log n)}, O(T(n)))$.

(5) Uniform Var&Sz $(B\forall)[O(1), O(T(n))]$.

PROPOSITION 11. *The equivalences in Proposition* 10 *also hold among the nonuniform versions of the models.*

In all these models, closure under complementation seems surprising. In nondeterministic auxiliary pushdown automata we face the usual problems of nondeterminism, in addition to the difficulties introduced by large stacks, and perhaps by super-polynomial running times. Although alternating Turing machine space and/or time classes are easily seen to be closed under complementation, the same proof (basically De Morgan's laws) converts a computation of small tree-size into one of large tree-size. Similar issues thwart the De Morgan approach to complementing circuits with bounded fan-in AND gates and formulate with Boolean universal variables—they become bounded fan-in OR gates and Boolean existentials instead. Nevertheless, closure under complementation follows for all these models from the theorem below.

THEOREM 12. *There is a constant* $\bar{c}$ *such that, for any n-input semi-unbounded fan-in circuit* α_n *of size* (*number of gates plus inputs*) Z *and depth* D, *there is a semi-unbounded fan-in circuit* $\overline{\alpha_n}$ *of size at most* $\bar{c}DZ^2 \log Z$ *and depth at most* $\bar{c}(D + \log Z)$ *that computes the complement of the function computed by* α_n. *The same result holds uniformly for uniform families of circuits* $\{\alpha_n\}$.

Proof. The key idea in the proof is again the use of inductive counting to verify "negative" information. In this case we are interested in counting the number of gates at a given depth that evaluate to 1.

Suppose we are given a circuit α_n of size Z and depth D. It is convenient to first convert it to a circuit β_n of size $2DZ + 2n$ and depth $2D$ that is:

(1) Synchronous—i.e., vertices can be assigned to levels so that input variables and their negations are on level 0, any gate on level i receives all its inputs from vertices on level $i-1$, and all output gates are on level $2D$;

(2) Fixed width—i.e., each level $i \geqq 1$ contains exactly Z gates;

(3) Strictly alternating—i.e., for all $i \geqq 0$, all gates on level $2i+1$ are OR gates and all gates on level $2i+2$ are AND gates; and

(4) Equivalent—i.e., it computes the same function as α_n.

This normal form is easily achieved. For example, for each level of β_n except the 0th, make a replica of each vertex of α_n. The replica on level i of an input vertex g is a trivial gate (of the appropriate type) whose only input is the replica of g on level $i-1$. Similarly, the replica of an AND gate g on an OR level i is a trivial OR gate whose only input is the replica of g on level $i-1$. A replica of an AND gate g on an AND level i has as inputs the replicas on level $i-1$ of g's inputs. OR gates are handled similarly, but with nontrivial replicas only on OR levels. Level 0 contains only the $2n$

vertices representing the input literals. Level 1 replicas of gates whose inputs are not all input literals are arbitrarily connected to literals. We can show by induction that all vertices of depth at most i in α_n will be correctly computed by their replicas on levels greater than or equal to $2i$ in β_n.

The "counting" we need in the construction is easily accomplished by threshold functions. Thus, rather than producing the circuit advertised by the theorem directly, it is easier to first produce a circuit γ_n containing THRESHOLD gates. (A c-THRESHOLD gates has unbounded fan-in, and outputs 1 if and only if at least c of its inputs have value 1.) This circuit will have the following properties:

(1) It has bounded fan-in AND gates and unbounded fan-in OR gates, i.e., it has semi-unbounded fan-in.

(2) It contains arbitrary THRESHOLD gates.

(3) No path from output to input contains more than two THRESHOLD gates.

(4) It has size $4D(Z+1)^2+2DZ+2n-1=O(DZ^2)$.

(5) It has depth at most $2D+3$.

(6) It computes the complement of the function computed by β_n.

The theorem will then follow by replacing the THRESHOLD gates by monotone SAC^1 threshold circuits. (Monotonicity is needed to preserve the semi-unbounded fan-in property. By property (3) above, the depth of the threshold subcircuits will increase the overall depth only additively.)

For $0 \leqq k \leqq D$, let

$$P_k = \{g \mid g \text{ is a vertex of } \beta_n \text{ on level } 2k \text{ having value } 1\}.$$

The main quantities we will be interested in counting are $\# P_k$, the cardinalities of the sets P_k.

Our main construction, of γ_n from β_n, follows. It is sketched in Fig. 2.

Construction Step 1. The entire circuit β_n is a subcircuit of γ_n. The gates in this subcircuit will be referred to as "original" gates. (Note that this means original to β_n, not to α_n.)

Construction Step 2. For $1 \leqq k \leqq D$, each original gate g on level $2k-1$ or $2k$, and each $0 \leqq c \leqq Z$, there is a "contingent complement" gate $\text{CC}(g \mid c)$ whose value will be the complement of the value of g *if* $c = \# P_{k-1}$; if $c \neq \# P_{k-1}$, then the value of $\text{CC}(g \mid c)$ is irrelevant. We compute $\text{CC}(g \mid c)$ as follows:

(1) If g is an AND gate, say $\text{AND}(a, b)$, then $\text{CC}(g \mid c)$ is the OR of $\text{CC}(a \mid c)$ and $\text{C}(b \mid c)$. (Fan-in greater than two is handled similarly.)

(2) If g is an OR gate (on level $2k-1$), then $\text{CC}(g \mid c)$ is a c-THRESHOLD gate whose inputs are all original gates on level $2k-2$ that are *not* inputs to g.

We now argue the correctness of this part of the construction.

LEMMA 13. *For* $k \geqq 1$, *and all original gates* g *on level* $2k-1$ *or* $2k$,

$$\text{CC}(g \mid \# P_{k-1}) \equiv \neg g.$$

Proof.

Case 1. g is an OR gate on level $2k-1$, $k \geqq 1$. Then g evaluates to 0 if and only if all g's inputs evaluate to 0, if and only if all $\# P_{k-1}$ original gates on level $2(k-1)$ that evaluate to 1 are *not* inputs to g, if and only if the THRESHOLD gate $\text{CC}(g \mid \# P_{k-1})$ evaluates to 1.

Case 2. g is an AND gate on level $2k$, $k \geqq 1$. Its inputs a and b are OR gates on level $2k-1$. From Case 1 we know that $\text{CC}(a \mid \# P_{k-1}) \equiv \neg a$ and $\text{C}(b \mid \# P_{k-1}) \equiv \neg b$, so

$$\neg g \equiv \neg(a \wedge b) \equiv \neg a \vee \neg b \equiv \text{CC}(a \mid \# P_{k-1}) \vee \text{CC}(b \mid \# P_{k-1}) \equiv \text{CC}(g \mid \# P_{k-1}).$$

The proof for fan-in greater than two is analogous. □

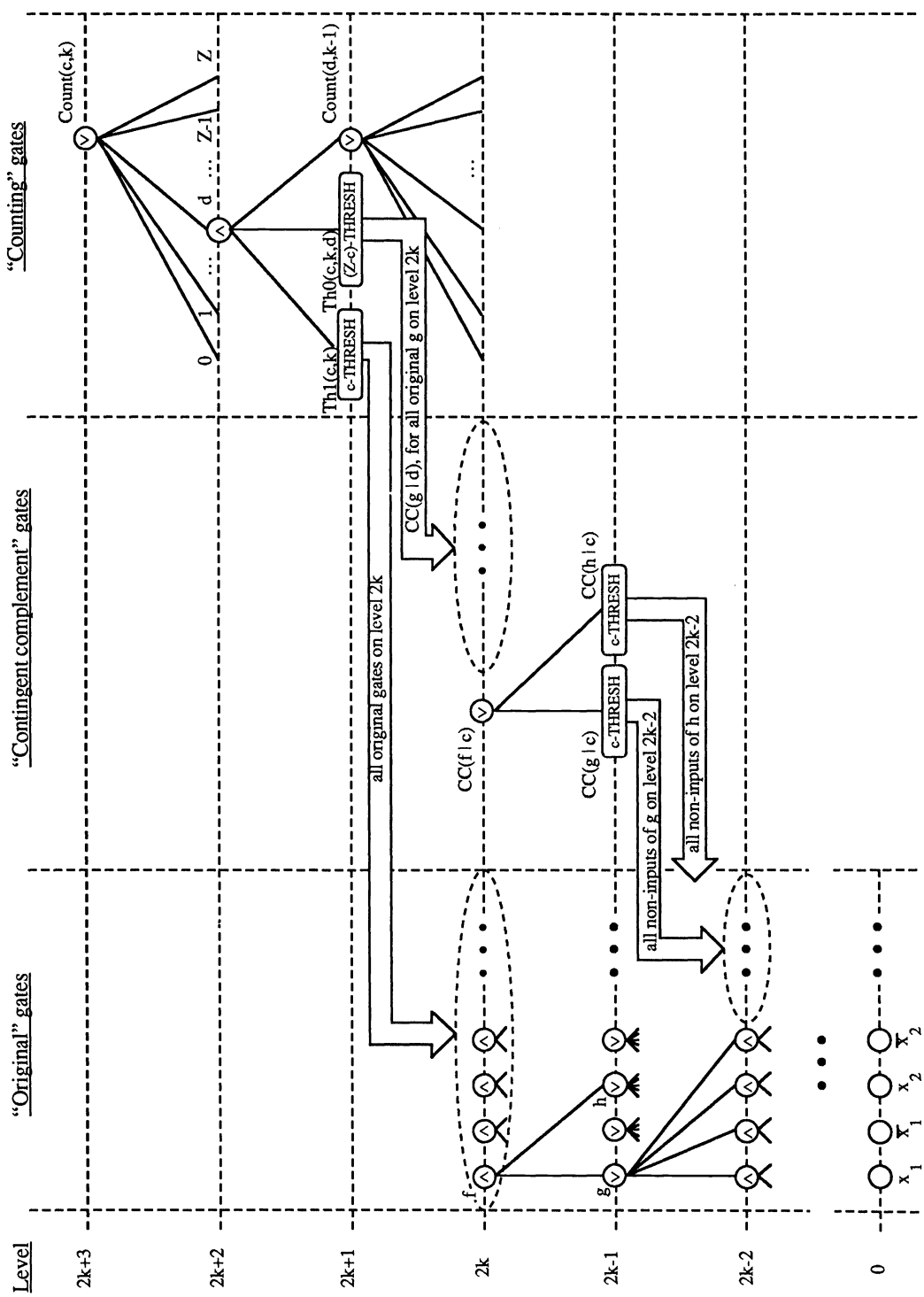


FIG. 2. *Construction of* γ_n.

Construction Step 3. Continuing the construction, there is a "counting" gate COUNT(c, k), for all $0 \leqq c \leqq Z$ and $0 \leqq k \leqq D-1$, whose value will be 1 if and only if $\# P_k = c$. This is computed as follows:

$$\text{COUNT}(c,k) = \begin{cases} 1 \Leftrightarrow (c = n) & \text{if } k = 0, \\ \bigvee_{d=0}^{Z} \text{AND}(\text{COUNT}(d, k-1), \text{TH1}(c,k), \text{TH0}(c,k,d)) & \text{if } k \geqq 1, \end{cases}$$

where

TH1(c, k) is the c-THRESHOLD of all original gates on level $2k$, and
TH0(c, k, d) is the $(Z-c)$-THRESHOLD of the contingent complement gates CC$(g \mid d)$, where g ranges over all original gates on level $2k$.

LEMMA 14. COUNT$(c, k) = 1$ *if and only if* $\# P_k = c$.

Proof. The proof proceeds by induction on k.

Basis $(k = 0)$. There are exactly $2n$ vertices on level 0, namely the n inputs and their complements. Exactly n of them evaluate to 1.

Induction $(k > 0)$. By induction, COUNT$(d, k-1)$ evaluates to 1 for exactly one value of d, namely $d = \# P_{k-1}$. Thus the only term in the disjunction $\bigvee_{d=0}^{Z} \cdots$ that could possibly evaluate to 1 is $d = \# P_{k-1}$. For this term, each contingent complement gate CC$(g \mid d)$ counted by TH0(c, k, d) computes $\neg g$ by Lemma 13. Thus, TH1$(c, k) \wedge$ TH0(c, k, d) evaluates to 1 if and only if, on level $2k$, there are at least c original gates with value 1 and at least $(Z-c)$ original gates with value 0, and hence exactly c with value 1. Thus, COUNT(c, k) evaluates to 1 if and only if $c = \# P_k$. □

Construction Step 4. We complete the construction by defining the outputs of γ_n. For all original gates g that are outputs of β_n (hence on level $2D$), γ_n contains a gate COMP(g) that evaluates to 1 if and only if g evaluates to 0. COMP(g) is computed as follows:

$$\text{COMP}(g) = \bigvee_{c=0}^{Z} \text{AND}(\text{COUNT}(c, D-1), \text{CC}(g \mid c)).$$

Correctness is easily shown. By Lemma 14, COUNT$(c, D-1)$ evaluates to 1 if and only if $\# P_{D-1} = c$, and by Lemma 13, CC$(g \mid \# P_{D-1}) \equiv (\neg g)$. Hence COMP$(g) \equiv (\neg g)$.

Thus γ_n correctly computes the complement of the function computed by α_n.

Analysis. Next we will analyze the size and depth of γ_n.

Define the THRESHOLD-*depth* of a gate g of γ_n to be the maximum number of THRESHOLD gates, including g, along any path from g to an input vertex.

For each original gate g on level k, g is also at depth k in γ_n, and has THRESHOLD-depth 0. For all $0 \leqq c \leqq Z$, gate CC$(g \mid c)$ also has depth k, and THRESHOLD-depth 1, since the THRESHOLD gates among CC$(g \mid c)$ depend *only* on original gates, not on other contingent complement gates.

The gates TH1(c, k) have depth $2k+1$ and THRESHOLD-depth 1. The gates TH0(c, k, d) have depth $2k+1$ and THRESHOLD-depth 2.

The gates COUNT(c, k) have THRESHOLD-depth 2 and depth d_k, where

$$d_k = \begin{cases} 0 & \text{if } k = 0, \\ 2 + \max(d_{k-1}, 2k+1) & \text{if } k \geqq 1. \end{cases}$$

Thus $d_k \leqq 2k+3$.

Finally, the output gates COMP(g) have THRESHOLD-depth 2 and depth $2+\max(d_{D-1}, 2D) \leqq 2D+3$.

The size of the circuit is easily seen to be polynomial in Z and D. The dominant contributions are from the contingent complement subcircuits, which contain $O(DZ^2)$ OR and THRESHOLD gates, and from the COUNT subcircuits, which contain $O(DZ^2)$ AND and THRESHOLD gates. A careful analysis gives the exact bound of $4D(Z+1)^2+2DZ+2n-1$ claimed above.

To complete the proof of Theorem 12, we need to replace the THRESHOLD gates in γ_n by monotone SAC^1 threshold circuits, and analyze the size and depth of the resulting Boolean circuit $\overline{\alpha_n}$.

The existence of monotone SAC^1 threshold circuits is easily established. A simple divide-and-conquer technique gives n input monotone SAC^1 circuits of size $O(n^2)$ and depth at most $2\lceil \log_2 n \rceil$: the k-THRESHOLD of n bits can be computed as the OR over $0 \leqq j \leqq k$ of the AND of the j-THRESHOLD of the first half of the bits and the $(k-j)$-THRESHOLD of the last half of the bits:

$$k\text{-THRESHOLD}(x_1, \cdots, x_n)$$

$$= \bigvee_{j=0}^{k} \text{AND}(j\text{-THRESHOLD}(x_1, \cdots, x_{\lfloor n/2 \rfloor}),$$

$$(k-j)\text{-THRESHOLD}(x_{\lfloor n/2 \rfloor+1}, \cdots, x_{2n})).$$

Asymptotically, the smallest known monotone SAC^1 threshold circuits are actually NC^1 circuits: the $O(n \log n)$ size "AKS" sorting networks of Ajtai, Komlós, and Szemerédi [1]. Replacing each THRESHOLD gate in γ_n by one of these subcircuits, and noting that each THRESHOLD gate has at most Z inputs, would give an overall size for the Boolean circuit $\overline{\alpha_n}$ of $O(DZ^3 \log Z)$.

One observation reduces this substantially. Namely, a *single* n-input AKS network computes the c-threshold of its inputs for *all* $1 \leqq c \leqq n$. Thus, although an OR gate g gives rise to Z THRESHOLD gates CC$(g \mid c)$, $1 \leqq c \leqq Z$, all of these values can be computed by one AKS network. Similar combination is possible among the TH0$(c, -, -)$ and TH1$(c, -)$ gates, $1 \leqq c \leqq Z$. This reduces the size of $\overline{\alpha_n}$ to $O(DZ^2 \log Z)$, as claimed.

The depth of $\overline{\alpha_n}$ is the depth of γ_n plus twice the depth of the AKS network (since γ_n has THRESHOLD-depth 2), which is $O(D+\log Z)$, as claimed.

For the uniform case of the theorem, we observe that the transformation from α_n to $\overline{\alpha_n}$ is quite simple and regular. We leave it to the reader to verify that this transformation preserves uniformity. (The AKS networks are known to be uniform.) □

COROLLARY 15. *For all $k \geqq 1$, SAC^k and nonuniform SAC^k are closed under complementation.*

Cook [7] defined *CFL** as the set of functions each computable by a uniform family $\{\alpha_n\}$ of circuits, where α_n has n inputs, bounded fan-in AND, OR, and NOT gates, unbounded fan-in oracle gates for some context-free language, and $O(\log n)$ depth. An oracle gate with fan-in f is defined to contribute $\lceil \log_2 f \rceil$ to the depth of any path containing it.

COROLLARY 16. *Any function in CFL* can be computed by a uniform family of polynomial size, $O(\log n)$ depth, semi-unbounded fan-in circuits.*

Proof. Let α_n be a *CFL** circuit with oracle gates for some context-free language L. With a doubling of size and no increase in depth, α_n can be simulated by a *CFL**

circuit β_n whose NOT gates appear only at the inputs, provided β_n is allowed oracle gates for both L and $\bar{L}$. (See, for example, [13].) The result now follows from the fact that LOGCFL $= SAC^1$ [37], together with Corollary 15. □

Similarly, it is true that any 0-1 valued function in NL^* (see Cook [7]) is also in NL. This follows from Immerman's [17] and Szelepcsényi's theorems [33], and was noted independently by S. Buss (personal communication).

3.2. The weakness of role switches in pebbling. There are a number of ways in which we might define a "*LOGCFL* hierarchy." One consequence of Corollary 15 is that, for many reasonable ways of doing so, the resulting hierarchy collapses to *LOGCFL*. In this section we consider one such hierarchy that collapses. As a consequence, the "role switch" resource [38] in pebbling is shown to be much weaker than previously seemed plausible.

(In contrast, following a preliminary version of the present paper, Jenner and Kirsig [19] considered an alternative formulation of a hierarchy based on *LOGCFL*, showing that it coincides with the polynomial hierarchy and hence presumably does *not* collapse.)

We begin by presenting the hierarchy. Define a (z, d, k, f)-*circuit* as an unbounded fan-in circuit of size z and depth d, where negations appear only at the inputs, and the vertices can be partitioned into k "layers" that alternately have AND fan-in at most f and OR fan-in at most f. More precisely, the vertices can be partitioned into blocks $B_k, B_{k-1}, \cdots, B_1$ (B_1 containing the outputs) such that:

(1) Any wire (u, v) with $u \in B_i$ and $v \in B_j$ satisfies $i \geqq j$;

(2) All AND gates of odd-numbered blocks have fan-in at most f; and

(3) All OR gates of even-numbered blocks have fan-in at most f.

For any fixed k, let

$$\Sigma_k^{CFL} = \{L \mid L \text{ can be recognized by a (nonuniform) family of } (n^{O(1)}, O(\log n), k, O(1))\text{-circuits}\}.$$

For instance, Σ_1^{CFL} is nonuniform SAC^1. Corollary 18 demonstrates that this hierarchy collapses.

THEOREM 17. *Let $\{\alpha_n\}$ be a family of $(z, d, k, O(1))$-circuits, where z, d, and k may be functions of n. Then there is a family $\{\beta_n\}$ of $(O(dz^2 \log z), O(d + k \log z), 1, O(1))$-circuits such that β_n computes both the outputs of α_n and their negations.*

Proof. This is proved by induction on k, using Theorem 12 and De Morgan's laws in a straightforward way. □

COROLLARY 18. *For any fixed $k \geqq 1$, $\Sigma_k^{CFL} = \Sigma_1^{CFL}$.*

Proof. This follows from Theorem 17 by substituting $z = n^{O(1)}$ and $d = O(\log n)$. □

To see how much more general Theorem 17 is than Corollary 18, consider the case $z = n^{O(1)}$ and $d = O(\log^i n)$, for any $i \geqq 1$, which might be called the "SAC^i hierarchy." Not only does $k = \Theta(1)$ fail to add power to nonuniform SAC^i, but $k = \Theta(\log^{i-1} n)$ does as well.

The purpose of this section is to apply this result to a pebble game introduced by Venkateswaran and Tompa [38]. They defined a new resource called "role switches." It will follow from Theorems 19 and 20 that Σ_k^{CFL} is the set of languages recognized by polynomial-size circuits pebbleable with $O(1)$ pebbles, $O(\log n)$ rounds, and $k-1$ role switches. Hence, by Corollary 18, any such circuit can be simulated by one pebbleable with $O(1)$ pebbles, $O(\log n)$ rounds, and zero role switches. Similarly, any

polynomial-size circuit pebbleable with $O(1)$ pebbles, $O(\log^i n)$ rounds, and $O(\log^{i-1} n)$ role switches can be simulated by one pebbleable with $O(1)$ pebbles, $O(\log^i n)$ rounds, and zero role switches.

The pebble game introduced by Venkateswaran and Tompa is referred to as the *dual interpreted two-person pebble game.* This game is played by two players, called Player 0 and Player 1, on a circuit α_n together with values for its n inputs and their negations (referred to collectively as *literals*). There are two minor differences between Venkateswaran and Tompa's game and the one used in this section: for convenience, we assume that α_n is *nonuniform* and has *unbounded* fan-in. The latter condition does not affect the resources considered, provided the depth of α_n is $\Omega(\log n)$.

At any point, one of the players takes on the role of the Challenger and the other that of the Pebbler. The Challenger is responsible for selecting the "currently challenged vertex"; the Pebbler has a collection of pebbles that it can place on or remove from the vertices of α_n. The role of a player is automatically determined as part of the circuit information as follows. The vertices in α_n are partitioned into two sets, those of "challenge type" 0 and those of "challenge type" 1. If the currently challenged vertex is of challenge type 0 (challenge type 1), then Player 0 (Player 1) is the Challenger in the next round. A Boolean circuit augmented with this role information for each of its vertices will be referred to as an *augmented* circuit. For convenience, it is assumed that the output vertex has challenge type 0.

The objective of Player 0 (Player 1) is to establish that the output of the circuit evaluates to 0 (1). A pebble placement or challenge on a vertex v by Player 0 (Player 1) corresponds to asserting that v evaluates to 0 (1). A pebble placed by Player 0 (Player 1) will be referred to as a 0-pebble (1-pebble).

The initial challenge is on the output vertex. The game proceeds in rounds, with a round consisting of the following three parts. (a) If the game is not over at the currently challenged vertex u according to the conditions below, then Player 0 is the Challenger for this round if u is of challenge type 0 and the Pebbler otherwise. (b) In the *pebbling move*, the Pebbler picks up zero or more of its own pebbles from vertices already pebbled and places pebbles on any nonempty set of vertices. (c) In the *challenging move*, the Challenger either rechallenges the currently challenged vertex or challenges one of the vertices that acquired a pebble in the current round.

Player 1 wins the game if, immediately following the Challenger's move, the currently challenged vertex is a literal with value 1, or an OR gate at least one of whose immediate predecessors is 1-pebbled, or an AND gate all of whose immediate predecessors are 1-pebbled. Player 0 wins if, immediately following the Challenger's move, the currently challenged vertex is a literal with value 0, or an OR gate all of whose immediate predecessors are 0-pebbled, or an AND gate at least one of whose immediate predecessors is 0-pebbled. It is also possible to have a winner in an infinite play of the game, namely that player (if either) who is the Pebbler in only finitely many rounds. (The purpose of this last rule is to force each player to make progress as the Pebbler.)

These notions are defined more precisely below. Fix an augmented circuit α_n and its input x.

A *configuration* of the game is a tuple $(t, P_1, P_0, R_1, R_0, v)$, where
$t \in \{P, C\}$ indicates whether it is the Pebbler's or the Challenger's turn to move,
P_1 is the set of vertices with 1-pebbles on them from previous rounds,
P_0 is the set of vertices with 0-pebbles on them from previous rounds,
R_1 is the set of vertices 1-pebbled in the current round,
R_0 is the set of vertices 0-pebbled in the current round, and
v is the currently challenged vertex.

The initial configuration of the game is $(P, \varnothing, \varnothing, \varnothing, \varnothing, s)$, where s is the output of the circuit.

A configuration $(P, P_1, P_0, \varnothing, \varnothing, v)$ is *terminal* if v is a literal, or an OR (AND) gate with some (all) of its immediate predecessors in P_1 or all (some) of its immediate predecessors in P_0.

A *move* in the game is made in accordance with a binary relation $\vdash$ on configurations defined as follows (where P_0, P_1, S_0, and S_1 are arbitrary sets of vertices, and R_0 and R_1 are arbitrary nonempty sets of vertices):

$(P, P_1, P_0, \varnothing, \varnothing, v) \vdash (C, P_1 - S_1, P_0, R_1, \varnothing, v)$, for all configurations $(P, P_1, P_0, \varnothing, \varnothing, v)$ that are not terminal and where v is of challenge type 0,

$(P, P_1, P_0, \varnothing, \varnothing, v) \vdash (C, P_1, P_0 - S_0, \varnothing, R_0, v)$, for all configurations $(P, P_1, P_0, \varnothing, \varnothing, v)$ that are not terminal and where v is of challenge type 1,

$(C, P_1, P_0, R_1, \varnothing, v) \vdash (P, P_1 \cup R_1, P_0, \varnothing, \varnothing, v')$, for all $v' \in R_1 \cup \{v\}$,

$(C, P_1, P_0, \varnothing, R_0, v) \vdash (P, P_1, P_0 \cup R_0, \varnothing, \varnothing, v')$, for all $v' \in R_0 \cup \{v\}$.

The *game tree* T is a maximal rooted tree whose nodes are labeled by configurations of the game, whose root is labeled by the initial configuration, and whose edge relation is given by $\vdash$. Note that the leaves of the tree are labeled by terminal configurations.

A *finite play* of the game is a finite path in the game tree from the root to some leaf labeled by the terminal configuration $(P, P_1, P_0, \varnothing, \varnothing, v)$. It is a *winning finite play for Player* 1 if v is a literal with value 1, or if v is an OR gate at least one of whose immediate predecessors is in P_1, or if v is an AND gate all of whose immediate predecessors are in P_1; otherwise it is a *winning finite play for Player* 0. An infinite path Π in the game tree is a *winning infinite play* for the player (if either) that is the Pebbler in only finitely many configurations on Π.

A *winning strategy for Player* 1 (if it exists) is a subtree W of T such that:

(1) W contains the root of T;

(2) W contains exactly one child of every nonterminal node in W that is labeled by a configuration in which it is Player 1's turn to move;

(3) W contains all children of every nonterminal node in W that is labeled by a configuration in which it is Player 0's turn to move; and

(4) All paths in W are winning (finite or infinite) plays for Player 1.

A *winning strategy for Player* 0 is defined dually.

Let $\{\alpha_n\}$ accept the language L, where each member α_n of the family is an augmented circuit with n inputs. The game on α_n with input $x \in L \cap \{0, 1\}^n$ can be *played simultaneously in* $p(n)$ *space,* $r(n)$ *rounds, and* $s(n)$ *role switches* if and only if there is a winning strategy for Player 1 in which:

(1) Every pebbling configuration $(P, P_1, P_0, \varnothing, \varnothing, v)$ along every path satisfies $|P_1| \leqq p(n)$,

(2) On any path, there are at most $r(n)$ edges $(P, P_1, P_0, \varnothing, \varnothing, v) \vdash (C, P_1 - S_1, P_0, R_1, \varnothing, v)$ with v or challenge type 0, and

(3) On any path, there are at most $s(n)$ edges $(C, P_1, P_0, R_1, R_0, v) \vdash (P, P_1', P_0', \varnothing, \varnothing, v')$ having the challenge types of v and v' unequal.

Resources on inputs $x \notin L$ are defined dually, considering winning strategies for Player 0 in place of those for Player 1.

Finally, α_n is *pebbleable* simultaneously in $p(n)$ space, $r(n)$ rounds, and $s(n)$ role switches if and only if, for all x of length n, the game on α_n can be played simultaneously in $p(n)$ space, $r(n)$ rounds, and $s(n)$ role switches. Note that only the resources used by the winning player are counted.

Theorems 19 and 20 demonstrate the intimate relationship between layered circuits and pebbling.

THEOREM 19. *Suppose* $\{\alpha_n\}$ *is a family of* $(z, r, s+1, p)$*-circuits, where* z, r, s, *and* p *may be functions of* n. *Then there is an assignment of challenge types to the vertices of* α_n *such that* α_n *is pebbleable simultaneously in* p *space,* r *rounds, and* s *role switches.*

Proof. Any vertex in a layer with AND fan-in (OR fan-in) bounded by p is assigned challenge type 0 (respectively, 1). Suppose α_n outputs 1 on input x. A winning strategy for Player 1 follows from the claim below. The winning strategy for Player 0 when α_n outputs 0 is dual.

CLAIM. Let v be the currently challenged vertex of α_n. Suppose v evaluates to 1 on input x, no predecessor of v is 0-pebbled, and the subcircuit induced by v and its predecessors is a $(z', r', s'+1, p')$-circuit. Then Player 1 can win the game with p' pebbles, r' rounds, and s' role switches.

The claim is proved by induction on r'. The basis $r' = 0$ is immediate. The induction depends on the role of Player 1, as follows.

Case 1. The challenge type of v is 0; i.e., v is in a layer with bounded AND fan-in. Then Player 1 is the Pebbler, and begins by removing all 1-pebbles from the circuit.

Case 1.1. v is an OR gate. Then Player 1 pebbles any one immediate predecessor u of v that evaluates to 1. (Note that Player 1 does not lose immediately, as no predecessors of u are 0-pebbled.) Player 0 must move the challenge to u in order not to lose immediately. If the challenge type of u is 1, then a role switch occurs and s' decreases by at least 1. In any case, the claim now follows from the induction hypothesis.

Case 1.2. v is an AND gate with bounded fan-in. Then Player 1 pebbles every immediate predecessor of v. The claim follows as in Case 1.1.

Case 2. The challenge type of v is 1. Then Player 1 is the Challenger. If Player 0 never pebbles a predecessor of v that evaluates to 1, Player 1 retains its challenge on v and wins using no resources. Suppose Player 0 pebbles a predecessor of v that evaluates to 1. Consider the first such move. Let u be such a predecessor of minimum depth among the vertices that are pebbled. Player 1 moves its challenge to u. Notice that no predecessor of u is 0-pebbled. The depth of the challenged vertex is decreased without Player 1 using any resources (with the possible exception of a role switch), so the result again follows from the induction hypothesis. □

THEOREM 20. *Suppose a family* $\{\alpha_n\}$ *of augmented circuits of size* z *is pebbleable simultaneously in* p *space,* r *rounds, and* s *role switches, where* z, p, r, *and* s *may be functions of* n. *Then there is a family* $\{\beta_n\}$ *of* $((r+1)^2(s+1)z^{O(p)}, 4r+5, s+1, p+1)$*-circuits that recognizes the same language as* $\{\alpha_n\}$.

Proof. Construction. β_n has one vertex $g(A, r_1, r_0, \hat{s})$ for every $0 \leqq r_1, r_0 \leqq r$, every $0 \leqq \hat{s} \leqq s$, and every configuration $A = (t, P_1, P_0, R_1, R_0, v)$ with $\#(P_1 \cup R_1) \leqq p$ and $\#(P_0 \cup R_0) \leqq p$. $r_1(r_0)$ will be used to count the number of rounds in which Player 1 (respectively, 0) has been Pebbler, and $\hat{s}$ will be used to count the number of role switches that have occurred. If $A = (t, P_1, P_0, R_1, R_0, v)$ is terminal with v a literal x_j, then $g(A, r_1, r_0, \hat{s})$ is the literal x_j. If $A = (t, P_1, P_0, R_1, R_0, v)$ is terminal with appropriate immediate predecessors of the challenged gate v in P_i, then $g(A, r_1, r_0, \hat{s})$ is the constant i. Otherwise $g(A, r_1, r_0, \hat{s})$ is a gate of type OR (AND) if and only if in A it is the turn of Player 1 (respectively, 0). The output gate is $g(I, 0, 0, 0)$, where I is the initial configuration on α_n. Let $g(A, r_1, r_0, \hat{s})$ be a gate of β_n, where $A =$

$(t, P_1, P_0, R_1, R_0, v)$ and let $A' = (t', P_1', P_0', R_1', R_0', v')$ satisfy $A \vdash A'$. For $i \in \{0, 1\}$, let

$$r_i' = \begin{cases} r_i + 1, & \text{if } t = P \text{ and } v \text{ is of challenge type } 1 - i, \\ r_i, & \text{otherwise,} \end{cases}$$

$$\hat{s}' = \begin{cases} \hat{s} + 1, & \text{if the challenge types of } v \text{ and } v' \text{ are unequal,} \\ \hat{s}, & \text{otherwise.} \end{cases}$$

If $r_i' > r$ or $\#(P_i' \cup R_i') > p$, then there is a wire from the constant $1 - i$ to $g(A, r_1, r_0, \hat{s})$ (indicating that Player $1 - i$ must win from A', since the winner never exceeds its resources). If $\hat{s}' > s$, there is a wire from an arbitrary constant to $g(A, r_1, r_0, \hat{s})$. Otherwise, there is a wire from $g(A', r_1', r_0', \hat{s}')$ to $g(A, r_1, r_0, \hat{s})$.

Correctness. Suppose α_n evaluates to 1 on x. The fact that β_n evaluates to 1 on x is shown by arguing that there is an "accepting subcircuit" S of β_n, that is, a set of vertices including the output all of which evaluate to 1. (The proof when α_n evaluates to 0 is dual.)

Since α_n evaluates to 1 on x, there is a winning strategy W for Player 1 in which Player 1 uses at most p pebbles, r rounds, and s role switches. By construction, there is a corresponding subcircuit S of β_n that contains the output, and in which one immediate predecessor of each OR gate in S and all immediate predecessors of each AND gate in S are also in S. The major difference between W and S is that some plays (in particular, all infinite plays) in W are truncated in S due to the conditions $r_0 > r$ or $\#(P_0 \cup R_0) > p$. (Neither $r_1 > r$ nor $\#(P_1 \cup R_1) > p$ nor $\hat{s} > s$ ever occurs in S, since Player 1's strategy uses at most r rounds, at most p pebbles, and at most s role switches.) All literals or constants of S that correspond to leaves of W evaluate to 1, since Player 1 wins at all leaves of W. Furthermore, any literal or constant of S that does not correspond to a leaf of W is the constant 1 by construction, since it arises from a truncation due to $r_0 > r$ or $\#(P_0 \cup R_0) > p$. Thus, β_n outputs 1.

Analysis. The size bound of β_n follows from the fact that the number of configurations of the pebble game is $z^{O(p)}$. The depth bound of $4r + 5$ follows from the fact that either r_0 or r_1 increases each round, and there are two moves (one for each player) per round. The fan-in bound follows from the fact that, whenever it is the turn of Player 0 (1) who is the Challenger in A, the AND fan-in (respectively, OR fan-in) of $g(A, r_1, r_0, \hat{s})$ is at most $p + 1$, since only $p + 1$ configurations (corresponding to the possibilities for the next challenged vertex) follow from A by a move of the Challenger. Finally, there is one layer for each of the $s + 1$ values of $\hat{s}$, since the same player remains Challenger as long as $\hat{s}$ remains unchanged. □

Let *ROUNDS, SWITCHES*$(r(n), s(n))$ denote the set of languages each of which is accepted by a nonuniform family of polynomial-size circuits pebbleable simultaneously with $O(1)$ pebbles, $r(n)$ rounds, and $s(n)$ role switches.

COROLLARY 21.

$$\textit{ROUNDS, SWITCHES}(r(n), s(n)) \subseteq \textit{ROUNDS, SWITCHES}(O(r(n) + s(n) \log n), 0).$$

Proof. This follows from Theorems 17, 19, and 20, noting for the size bound that $r(n)$ and $s(n)$ are $n^{O(1)}$. □

The final corollary states an explicit threshold beyond which role switches appear to add power. Equations (1) and (3) are the nonuniform analogues of Theorems 4 and 5 in [38].

COROLLARY 22. *For any* $i \geqq 1$,

(1) $\quad$ *nonuniform* $SAC^i = ROUNDS, SWITCHES(O(\log^i n), 0)$

(2) $\quad = ROUNDS, SWITCHES(O(\log^i n), O(\log^{i-1} n))$;

(3) $\quad$ *nonuniform* $AC^i = ROUNDS, SWITCHES(O(\log^i n), O(\log^i n))$.

Proof. Equation (1) follows from Theorems 19 and 20, since nonuniform SAC^i is, by definition, the set of languages each of which is accepted by a family of $(n^{O(1)}, O(\log^i n), 1, O(1))$-circuits. Equation (3) follows by the same argument, since nonuniform AC^i is characterized similarly by $(n^{O(1)}, O(\log^i n), O(\log^i n), O(1))$-circuits. Equation (2) follows from Corollary 21. □

4. Open problems. Figure 1 indicates the relationships among space-bounded complexity classes between NC^1 and NC^2. Since it is still possible that $NC^1 = NC^2$ (or indeed $NC^1 = NP$), there are no proven proper inclusions or incomparability results among these classes.

In addition to the problems identified by Cook [7], we call attention to certain questions suggested by this paper:

(1) Is SL closed under complementation? If so, then the $\bigcup_k C\Sigma_k^{SL}$ hierarchy collapses to SL.

(2) Is RLP closed under complementation? If so, then $RLP = ZPLP$. If not, what is a new candidate for a language in RLP (or $BPLP$) that is not in $ZPLP$?

(3) Assuming that $RLP \neq NL = RL = ZPL$, we see that the expected polynomial-time bound is important in the case of errorless and one-sided error probabilistic $O(\log n)$ space computations, whereas $PL = PLP$ in the case of two-sided unbounded error. The case of two-sided bounded error remains open; that is, is $BPL = BPLP$? Is there a candidate for a language in $BPLP$ (or BPL) that is not in NL?

It seems surprising that the NC^1 AKS networks [1] provide the best known SAC^1 networks for Boolean sorting. An interesting question is whether we could exploit the availability of unbounded fan-in OR gates to get a simpler $O(n \log n)$ size monotone Boolean sorter, and/or one with a more favorable constant hidden in the big-O. Indeed, size $o(n \log n)$ is not out of the question for this model. See Friedman [11] and Valiant [35] for other recent approaches to threshold computation.

REFERENCES

[1] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *Sorting in c* log *n parallel steps*, Combinatorica, 3 (1983), pp. 1-19.

[2] R. ALELIUNAS, R. M. KARP, R. J. LIPTON, L. LOVÁSZ, AND C. RACKOFF, *Random walks, universal traversal sequences, and the complexity of maze problems*, in 20th Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico, October 1979, pp. 218-223.

[3] P. BILLINGSLEY, *Probability and Measure*, John Wiley, New York, 1979.

[4] S. R. BUSS, S. A. COOK, P. W. DYMOND, AND L. HAY, *The log space oracle hierarchy collapses*, Tech. Report CS103, Department of Computer Science and Engineering, University of California, San Diego, CA, 1987.

[5] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[6] S. A. COOK, *Characterizations of pushdown machines in terms of time-bounded computers*, J. Assoc. Comput. Mach., 18 (1971), pp. 4-18.

[7] ———, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2-22.

[8] P. W. DYMOND AND S. A. COOK, *Hardware complexity and parallel computation*, in 21st Annual IEEE Symposium on Foundations of Computer Science, Syracuse, NY, October 1980, pp. 360-372.

[9] R. W. FLOYD, *Algorithm 97: shortest path*, Comm. ACM, 5 (1962), p. 345.

[10] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proc. 10th Annual ACM Symposium on Theory of Computing, San Diego, CA, May 1978, pp. 114-118.

[11] J. FRIEDMAN, *Constructing $O(n \log n)$ size monotone formulae for the k-th elementary symmetric polynomial of n Boolean variables*, in 25th Annual IEEE Symposium on Foundations of Computer Science, Singer Island, FL, October 1984, pp. 506–515.

[12] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.

[13] L. M. GOLDSCHLAGER, *The monotone and planar circuit value problems are log space complete for P*, SIGACT News, 9 (1977), pp. 25–29.

[14] J. HARTMANIS, *The structural complexity column*, Bull. European Assoc. Theoret. Comput. Sci., 33 (1987), pp. 26–39.

[15] L. A. HEMACHANDRA, *The strong exponential hierarchy collapses*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, NY, May 1987, pp. 110–122.

[16] R. R. HOWELL, L. E. ROSIER, AND H.-C. YEN, *Unary minimum cost path problems, alternating logspace, and Ruzzo, Simon and Tompa's DL^{NL}*, Department of Computer Sciences TR-87-13, University of Texas, Austin, TX, May 1987.

[17] N. IMMERMAN, *Nondeterministic space is closed under complementation*, SIAM J. Comput., 17 (1988), pp. 935–938.

[18] ———, *Upper and lower bounds on first order expressibility*, J. Comput. System Sci., 25 (1982), pp. 76–98.

[19] B. JENNER AND B. KIRSIG, *Characterizing the polynomial hierarchy by alternating auxiliary pushdown automata*, University of Hamburg, Federal Republic of Germany, 1987.

[20] H. JUNG, *On probabilistic time and space*, in Automata, Languages, and Programming, Springer-Verlag, Berlin, 1985, pp. 310–317.

[21] S. C. KLEENE, *Representation of events in nerve nets and finite automata*, in Automata Studies, C. E. Shannon and M. McCarthy, eds., Princeton University Press, Princeton, NJ, 1956, pp. 3–40.

[22] K.-J. LANGE, B. JENNER, AND B. KIRSIG, *The logarithmic alternation hierarchy collapses: $A\Sigma_2^{\mathcal{L}} = A\Pi_2^{\mathcal{L}}$*, in Automata, Languages, and Programming, Springer-Verlag, Berlin, 1987, pp. 531–541.

[23] H. R. LEWIS AND C. H. PAPADIMITRIOU, *Symmetric space-bounded computation*, Theoret. Comput. Sci., 19 (1982), pp. 161–187.

[24] S. R. MAHANEY, *Sparse complete sets for* NP: *Solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[25] J. H. REIF, *On synchronous parallel computations with independent probabilistic choice*, SIAM J. Comput., 13 (1984), pp. 46–56.

[26] ———, *Symmetric complementation*, in Proc. 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, May 1982, pp. 201–214.

[27] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. System Sci., 21 (1980), pp. 218–235.

[28] W. L. RUZZO, J. SIMON, AND M. TOMPA, *Space-bounded hierarchies and probabilistic computations*, J. Comput. System Sci., 28 (1984), pp. 216–230.

[29] W. J. SAVITCH, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[30] U. SCHÖNING AND K. WAGNER, *Collapsing oracle hierarchies, census functions, and logarithmically many queries*, in 5th Annual Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, Berlin, 1988, pp. 91–97.

[31] J. SIMON, *Space-bounded probabilistic Turing machine complexity classes are closed under complement*, in Proc. 13th Annual ACM Symposium on Theory of Computing, Milwaukee, WI, May 1981, pp. 158–167.

[32] I. H. SUDBOROUGH, *On the tape complexity of deterministic context-free languages*, J. Assoc. Comput. Mach., 25 (1978), pp. 405–414.

[33] R. SZELEPCSÉNYI, *The method of forcing for nondeterministic automata*, Bull. European Assoc. Theoret. Comput. Sci., 33 (1987), pp. 96–100.

[34] S. TODA, *$\Sigma_2 SPACE(n)$ is closed under complement*, J. Comput. System Sci., 35 (1987), pp. 145–152.

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

[36] H. VENKATESWARAN, *Characterizations of parallel complexity classes*, Ph.D. thesis, University of Washington, Seattle, WA, August 1986; available as Department of Computer Science Tech. Report No. 86-08-06.

[37] ———, *Properties that characterize* LOGCFL, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, NY, May 1987, pp. 141–150.

[38] H. VENKATESWARAN AND M. TOMPA, *A new pebble game that characterizes parallel complexity classes*, SIAM J. Comput., this issue, pp. 533–549.

[39] S. WARSHALL, *A theorem on Boolean matrices*, J. Assoc. Comput. Mach., 9 (1962), pp. 11–12.

[40] D. J. A. WELSH, *Randomised algorithms*, Discrete Appl. Math., 5 (1983), pp. 133–145.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 579-583, June 1989

010

NOTE ON WEINTRAUB'S MINIMUM-COST CIRCULATION ALGORITHM*

FRANCISCO BARAHONA† AND ÉVA TARDOS‡

Abstract. In 1974 Weintraub [*Management Sci.*, 21 (1974), pp. 87-97] published an algorithm for the minimum-cost circulation problems with convex cost function. In this note Weintraub's algorithm is considered when applied to a minimum-cost circulation problem with linear objective function. It is shown that a minor variation of the algorithm runs in polynomial time. The resulting algorithm, although it is not strongly polynomial, does not rely on scaling. It is a generalization of the maximum flow algorithm due to Edmonds and Karp [*J. Assoc. Comput. Mach.*, 19 (1972), pp. 248-264] that augments along the fattest augmenting path in the residual graph.

The algorithm described here is slower than the fastest minimum-cost circulation algorithms known. The authors' interest in the algorithm is partially historical, a scaling free "almost polynomial time" algorithm was published in 1974, and partially due to the different ideas involved.

Key words. network flow, circulation, scaling, algorithms

C. R. subject classifications. F.2.2, G.2.2

1. Introduction. The circulation problem is one of the fundamental problems in the theory of algorithms. It is very often used in practice and has contributed considerably to our understanding of the efficiency of algorithms, especially of how the presence of numbers in a combinatorial problem affect the complexity of the problem.

An instance of the minimum-cost circulation problem is given by a directed graph $G=(V, E)$ and capacities $u: E \mapsto Z_+$ and costs $c: E \mapsto Z$ on the arcs. For notational convenience we assume that for every pair $v, w \in V$, there is at most one arc $e \in E$ with ends v and w. A *feasible circulation* is a vector $x \in R^E$ such that

$$0 \leqq x(e) \leqq u(e) \quad \text{for every } e \in E,$$

$$\sum_{w \in V} x(w, v) = \sum_{w \in V} x(v, w) \quad \text{for every node } v \in V.$$

The *cost of the circulation* x is

$$cx = \sum_{e \in E} c(e)x(e).$$

The *minimum-cost circulation problem* is to find a feasible circulation of minimum cost. We shall use n to denote the number of nodes, m to denote the number of arcs in G, and U and C to denote the maximum capacity and cost, respectively.

The problem was studied already as early as the 1940s. The first algorithm, the "out-of-kilter" method, was introduced in the early 60s (see Ford and Fulkerson [2]). The out-of-kilter method is a pseudo-polynomial algorithm and its running time is proportional to the unary size of the capacities.

* Received by the editors March 21, 1988; accepted for publication September 13, 1988. This research was partially supported by Air Force Office of Scientific Research contract AFOSR-86-0078.

† Department of Combinatorics and Optimization, University of Waterloo, Ontario, Ontario, Canada, N2L 3G1 and Institute for Operations Research, Bonn, Federal Republic of Germany.

‡ Computer Science Department, Eötvös University, Budapest, Hungary, and Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02319.

The first polynomial time algorithm for the problem was discovered by Edmonds and Karp in 1972 [1]. This is the first so-called scaling algorithm. Since then, scaling algorithms have been widely studied and used for a variety of combinatorial optimization problems.

A theoretical disadvantage of scaling algorithms is that they are not strongly polynomial (that is, the number of arithmetic operations used by the algorithm depends on the size of the numbers involved, and not only on the number of nodes and arcs in the graph). In fact, scaling algorithms have a non-strongly polynomial "best case running time," since the algorithm handles each bit in the binary description of the capacities separately.

The first strongly polynomial algorithm for the minimum cost circulation problem was published thirteen years later in 1985 [8]. The current most efficient strongly polynomial algorithm is due to Orlin [7]. Goldberg and Tarjan [3] discovered a very elegant strongly polynomial algorithm. This algorithm is similar to the one studied in this paper in the sense that both are based on the same basic idea of canceling cycles (Klein [5]).

In this note, we consider an algorithm due to Weintraub [10] that was published two years after the Edmonds-Karp paper. The algorithm is based on the idea due to Klein [5] of repeatedly canceling negative cost cycles in the residual graph of a feasible circulation. Although Weintraub considers the circulation problem with convex costs, here we shall restrict our attention to the minimum-cost circulation problem with linear cost function. In the following, we shall refer to this special case as Weintraub's algorithm. In the paper [10] Weintraub proves that his cycle selection rule results in a minimum-cost circulation after canceling a polynomial number of cycles, but his algorithm takes superpolynomial time to find the right cycles to cancel. We show that the required cycle can be found in polynomial time. To make this note self-contained we shall describe the special case of Weintraub's algorithm in detail.

Weintraub's method relies on a subroutine that solves the assignment problem, and therefore its running time is worse than the fastest known minimum-cost circulation algorithms ([4], [7]). We find it still interesting to see how Weintraub's approach, which was published in 1974, can yield a fairly simple polynomial time algorithm.

2. A modification of Weintraub's algorithm. Given a circulation problem and a feasible circulation x we define the associated *residual graph* $G_x=(V, E_x)$, where $(v, w)\in E_x$ if either $(v, w)\in E$ and $x(v, w)<u(v, w)$ or $(w, v)\in E$ and $x(w, v)>0$. In the first case the arc (v, w) has *residual capacity* $u_x(v, w)=u(v, w)-x(v, w)$ and cost $c(v, w)$. In the second case $u_x(v, w)=x(w, v)$ and $c(v, w)=-c(w, v)$.

The algorithm is based on the following well-known fact (see, e.g., [6]).

LEMMA 1. *A feasible circulation x is of minimum cost if and only if the residual graph G_x contains no negative cost residual cycles.*

The main step of the algorithm is canceling negative cost cycles. We define the characteristic vector of a cycle C in the residual graph of a circulation as

$$\chi_C(e)=\begin{cases} 1 & \text{if } e=(v, w)\in C,\\ -1 & \text{if } e=(v, w) \text{ and } (w, v)\in C,\\ 0 & \text{otherwise.}\end{cases}$$

The amount of flow we can push around the cycle is $u_x(C)=\min_{(v,w)\in C} u_x(v, w)$. *Canceling* C results in a new feasible circulation $x'=x+u_x(C)\chi_C$.

Note that canceling a cycle C improves the cost of the circulation by

$$cx-cx'=-u_x(C)c\chi_C. \tag{1}$$

LEMMA 2. *For every feasible circulation x there exists a cycle C in the residual graph of x, canceling of which results in a feasible circulation x' with*

$$cx' - cx_{\text{opt}} \leqq \frac{m-1}{m}(cx - cx_{\text{opt}}),$$

where x_{opt} denotes the minimum-cost circulation.

Proof. Consider the circulation $x_{\text{opt}} - x$. This circulation can be decomposed into at most m conforming cycles. That is, there exists cycles C_i and positive scalars λ_i for $i = 1, \cdots, k$, such that $k \leqq m$ and $\sum_{i=1}^{k} \lambda_i \chi_{C_i} = x_{\text{opt}} - x$ and further $(v, w) \in C_i$ implies that either $(v, w) \in E$ and $x_{\text{opt}}(v, w) > x(v, w)$ or $(w, v) \in E$ and $x_{\text{opt}}(w, v) < x(v, w)$. Now C_i is a cycle in G_x for every $i = 1, \cdots, k$. Further $u_x(C_i) \geqq \lambda_i$. Therefore, the cycle C_i that maximizes $u_x(C_i) c\chi_{C_i}$ proves the lemma. □

As a corollary we can note that canceling the residual cycle that gives the biggest improvement in the cost of the circulation would give a polynomial-time algorithm (see the proof of Theorem 5). However, finding this cycle is NP-hard; it contains the Hamiltonian cycle problem as a special case.

Remark. The maximum flow problem can be reduced to the minimum-cost circulation problem by introducing a new arc (t, s) connecting the sink back to the source with infinite (very high) capacity and cost -1 and assigning cost 0 to all other arcs. A minimum cost circulation in this graph is equal to a maximum flow in the original graph with the arc (t, s) carrying the value of the flow. The circulation $x = 0$ is feasible. All negative cost cycles in the residual graph of any feasible circulation have cost -1 and go through the arc (t, s). Negative cost residual cycles correspond to augmenting paths for the maximum flow problem. The residual cycle whose canceling gives the maximum improvement in the cost is the cycle corresponding to the augmenting path with the minimum residual capacity along the path maximal. Such a path can be found in polynomial time. The resulting maximum flow algorithm is the polynomial time fattest path algorithm discovered by Edmonds and Karp [1].

Weintraub's main idea is that instead of solving the NP-hard problem of finding the best single cycle to cancel, we can look for a set of node-disjoint cycles and cancel them simultaneously. The following algorithm finds a set of node-disjoint cycles, canceling of which results in an improvement of the cost at least as big as the canceling of any single cycle. The algorithm will use a subroutine that solves the assignment problem (i.e., the minimum-cost matching problem in a bipartite graph) to find the right set of cycles.

Given a graph $G = (V, E)$ with a cost function c on its arcs, let us define a corresponding bipartite graph $B = (V' \cup V'', E')$ as follows.

Let V' and V'' denote two disjoint copies of V. Let v' and v'' denote the nodes in V' and V'', respectively, corresponding to node $v \in V$. Let $E' = \{v'w'' \mid \text{if either } v = w \text{ or } (v, w) \in E\}$. Define the costs on the edges of B as $c(v'w'') = c(v, w)$ if $v \neq w$ and 0 otherwise.

LEMMA 3. *For any directed graph G and the corresponding bipartite graph B the perfect matchings (assignments) of B are in one-to-one correspondence with the sets of node-disjoint cycles in G, and the two have the same cost.*

This lemma is the basis of the algorithm whose main step is described in Fig. 1. The subroutine takes a feasible circulation x as its input, and outputs a set of node-disjoint cycles that should be canceled. For feasible circulation x and a real number $\lambda \in R_+$ we use the graph $G(x, \lambda) = (V, E(x, \lambda))$, where $E(x, \lambda)$ denotes the set of arcs with residual capacity at least λ.

Step 1 Let $\lambda_1 > \cdots > \lambda_k > 0$ denote the different values of the residual capacities in G_x.

Step 2 For $i = 1, \cdots, k$ find the minimum cost set of node-disjoint cycles in $G(x, \lambda_i)$. This can be done using Lemma 3 and a subroutine to solve the resulting assignment problem. Let ν_i be equal to $-\lambda_i$ times the sum of the costs of the arcs in the set of cycles found.

Step 3 Choose the index i for which ν_i is maximal and output the set of cycles found in $G(x, \lambda_i)$.

FIG. 1. FIND CYCLES *subroutine.*

THEOREM 4. *Given a feasible circulation x, canceling the cycles found by the* FIND CYCLES *subroutine in Fig.* 1 *results in a new feasible circulation x', such that the improvement in the cost is at least as big as the improvement caused by canceling any single cycle in* G_x.

Proof. Let C denote the best residual cycle in G_x. Further, let λ_i denote the minimum residual capacity of the arcs in C. C is a cycle in $G(x, \lambda_i)$ and therefore (1) implies that the improvement in the cost caused by canceling C is at most ν_i. On the other hand, the cost improvement caused by canceling the cycles found by the subroutine is at least ν_i. □

THEOREM 5. *Starting with any feasible integer circulation x and repeatedly canceling the cycles found by the* FIND CYCLES *subroutine results in a minimum-cost circulation after at most* $O(m \log mUC)$ *calls to the subroutine.*

Proof. Let x_{opt} denote a minimum-cost circulation. Note that every intermediate feasible circulation x' found by this algorithm is integral, and therefore $cx' - cx_{\text{opt}} < 1$ implies that x' is of minimum cost. Further, any initial feasible circulation x_0 satisfies $cx_0 - cx_{\text{opt}} \leqq \sum_{e \in E} c(e)u(e) \leqq mCU$. Lemma 2 and Theorem 4 imply that the circulation x_k found after canceling k set of cycles given by the subroutine satisfies

$$cx_k - cx_{\text{opt}} \leqq (1 - 1/m)^k (cx_0 - cx_{\text{opt}}) \leqq (1 - 1/m)^k mCU.$$

For $k = m \log mUC$ we have that $cx_k - cx_{\text{opt}} < 1$ (since $(1 - 1/m)^m < e^{-1}$) and consequently x_k is optimal. □

THEOREM 6. *The above version of Weintraub's algorithm can be implemented to run in* $O(m^2(m + n \log m) \log mUC)$ *time.*

Proof. Theorem 5 implies that the FIND CYCLES subroutine will be used at most $O(m \log mUC)$ times. The cycles found by the subroutine can be canceled in $O(n)$ time. The most time-consuming part of the subroutine is solving the m assignment problems. Therefore, the running time can be bounded by $O(m^2 \log mUC)$ times the time required for solving a single assignment problem. This can be improved by noticing that the m assignment problems solved in one application of the subroutine are closely related: only one new edge is added to the old problem to create the new one. Having already found a solution to the previous problem, the new problem can be solved by a single call to a shortest path algorithm with nonnegative arclengths. Consequently, the FIND CYCLES subroutine can be implemented in $O(m(m + n \log n))$ time (see, e.g., [9]), and this gives an $O(m^2(m + n \log n) \log mUC)$ time minimum-cost circulation algorithm. □

Remark. Let us point out in what way the algorithm described by Weintraub differs from the one given above. Weintraub's algorithm has the same overall strategy,

and Weintraub has proved that the cycle selection rule implemented by the FIND CYCLES subroutine gives (in the case of a linear objective function) a minimum-cost circulation after canceling a polynomial number of cycles. The only difference lies in the choice of the set of values λ_i in Step 1 of the subroutine (Fig. 1). Weintraub was concerned with convex cost functions, where the number of different values of λ to be considered is even bigger than U. He did not notice that in the linear case m values suffice.

A scaling-like choice for the set of values λ_i that would also make the algorithm run in polynomial time has been pointed out to us by Plotkin and Orlin (personal communication). Namely, we can take the values λ_i in each application of the subroutine to be the different powers of 2 up to the maximum capacity U. In this case Theorem 4 has to be relaxed. This version of the subroutine finds a set of cycles, whose canceling improves the cost by at least half of the improvement possible by canceling any single cycle.

Open problem. It would be interesting to know if the minimum-cost circulation algorithm described here (or a minor modification of this algorithm) is actually strongly polynomial. Even Edmonds and Karp's fattest path maximum flow algorithm (that is, the same as our implementation of Weintraub's algorithm when applied to solve the maximum flow problem) is not known to run in strongly polynomial time.

REFERENCES

[1] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[2] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[3] A. V. GOLDBERG AND R. E. TARJAN, *Finding minimum-cost circulations by canceling negative cycles*, in Proc. 20th ACM Symposium on Theory of Computing, 1988, pp. 388–397.

[4] ———, *Solving minimum-cost flow problems by successive approximation*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 7–18.

[5] M. KLEIN, *A primal method for minimal cost flows with applications to the assignment and transportation problems*, Management Sci., 14 (1967), pp. 205–220.

[6] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Reinhart, and Winston, New York, 1976.

[7] J. B. ORLIN, *A faster strongly polynomial minimum cost flow algorithm*, in Proc. 20th ACM Symposium on Theory of Computing, 1988, pp. 377–387.

[8] E. TARDOS, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica, 5 (1985), pp. 247–255.

[9] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[10] A. WEINTRAUB, *A primal algorithm to solve network flow problems with convex costs*, Management Sci., 21 (1974), pp. 87–97.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 584-593, June 1989

011

FAST FOURIER TRANSFORMS FOR METABELIAN GROUPS*

MICHAEL CLAUSEN†

Abstract. Let G be a finite group . Then $L_s(G)$, the minimal number of arithmetic operations to evaluate a Fourier transform corresponding to G, is smaller than $2 \cdot |G|^2$. The fast Fourier transform algorithms improve this trivial upper bound by showing that for a cyclic group G, $L_s(G) \leqq c \cdot |G| \cdot \log |G|$. This last result is extended to metabelian groups, and it is shown that these groups also have fast inverse Fourier transforms. In particular there are fast algorithms for the (inverse) Fourier transforms for dihedral and generalized quaternion groups, as well as for all groups of square-free order.

Key words. fast Fourier transform, group algebras, metabelian groups, dihedral groups, quaternion groups

AMS(MOS) subject classifications. 68Q40, 20C15

1. Introduction. This paper is concerned with fast Fourier transforms (FFTs). For applications of such fast algorithms to signal and picture processing, coding theory, cryptography, and algorithmic circuit design, the interested reader is referred to [2], [3], [11], [13]. We start with a brief review of the mathematical background.

The set $\mathbf{C}G := \{a \mid a : G \to \mathbf{C}\}$ of all complex-valued functions on the finite group G becomes a $\mathbf{C}$-space by pointwise addition and scalar multiplication. Its $\mathbf{C}$-dimension equals the order $|G|$ of G. A natural $\mathbf{C}$-basis is given by the indicator functions of the group elements. By identifying each group element with its indicator function, $\mathbf{C}G$ can be viewed as the space of all formal sums $\sum_{g \in G} a_g g$ with complex coefficients. The multiplication in G can be bilinearly extended to $\mathbf{C}G$:

$$\left(\sum_{g \in G} a_g g\right) \cdot \left(\sum_{h \in G} b_h h\right) = \sum_{k \in G} \left(\sum_{(g,h):gh=k} a_g b_h\right) k.$$

Thus $\mathbf{C}G$ becomes a $\mathbf{C}$-algebra, the so-called *group algebra* of G over $\mathbf{C}$. For instance, the group algebra of the cyclic group C_n of order n can be identified with the algebra $\mathbf{C}[X]/(X^n - 1)$ of polynomials of degree $< n$ with multiplication modulo $X^n - 1$. By Chinese remaindering we know that $\mathbf{C}[X]/(X^n - 1)$ is isomorphic to the algebra of n-square diagonal matrices. This isomorphism, known as the (cyclic) *discrete Fourier transform* (DFT), can be viewed as a structural transition from the *signal domain* $\mathbf{C}C_n$ into the *spectral domain* $\mathbf{C}^n$; in particular, the DFT is multiplicative and thus links the convolution in $\mathbf{C}[X]/(X^n - 1)$ and the multiplication of n-square diagonal matrices. A fast multiplication of polynomials results from an FFT and the multiplication of diagonal matrices followed by a fast inverse Fourier transform.

Wedderburn's theorem, which generalizes the DFT for cyclic groups to arbitrary finite groups, says that the group algebra $\mathbf{C}G$ of a finite group G is isomorphic to a suitable algebra of block diagonal matrices:

$$\mathbf{C}G \simeq \bigoplus_{i=1}^{h} \mathbf{C}^{d_i \times d_i}.$$

The number h of blocks equals the number of conjugacy classes of G. Every such isomorphism is called a (*generalized*) *Fourier transform* for $\mathbf{C}G$. For the symmetric group S_3 it turns out that $\mathbf{C}S_3 \simeq \mathbf{C} \oplus \mathbf{C} \oplus \mathbf{C}^{2\times 2}$. The first two blocks correspond to the

* Received by the editors August 15, 1987; accepted for publication (in revised form) August 29, 1988.
† Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Federal Republic of Germany.

representations ($S_3 \ni \pi \mapsto 1$) and ($S_3 \ni \pi \mapsto \operatorname{sgn}(\pi)$), whereas the last block represents S_3 as the symmetry group of a regular triangle. In general, the h blocks occurring in Wedderburn's theorem correspond to the h classes of inequivalent irreducible representations of $\mathbf{C}G$. A (*matrix*) *representation* D of $\mathbf{C}G$ of *degree* or *dimension* d is a $\mathbf{C}$-linear and multiplicative transformation $D: \mathbf{C}G \to \mathbf{C}^{d\times d}$ which maps 1 onto 1; it is *irreducible* if no proper linear subspace of $\mathbf{C}^d$ is invariant under the image of D. A change of bases in $\mathbf{C}^d$, described by an invertible d-square matrix T, leads to a new representation $D'(g) := T \circ D(g) \circ T^{-1}$, which, by definition, is *equivalent* to D. If $D_1, \cdots, D_h$ is a transversal of the equivalence classes of irreducible representations of $\mathbf{C}G$, then

$$\bigoplus_i D_i : a \mapsto \bigoplus_i D_i(a)$$

is a Fourier transform for $\mathbf{C}G$ and every algebra isomorphism $\mathbf{C}G \to \oplus_i \mathbf{C}^{d_i \times d_i}$ can be described in this way.

As a classical example we take the cyclic group $C_n = \langle x \rangle$ of order n. Let ω be a primitive nth root of unity. Then $\{D_k := (\sum_m a_m x^m \mapsto \sum_m a_m \omega^{mk}) \mid 0 \leqq k < n\}$ is the set of all irreducible $\mathbf{C}$-representations of $\mathbf{C}C_n$ and the Fourier transform for $\mathbf{C}C_n$ with respect to natural $\mathbf{C}$-bases is given by the n-square DFT matrix (ω^{mk}). Let us proceed with the general case. With respect to natural $\mathbf{C}$-bases in $\mathbf{C}G$ and $\oplus_{i \leqq h} \mathbf{C}^{d_i \times d_i}$, each Fourier transform can be viewed as a $|G|$-square complex matrix. The *linear complexity* $L_s(A)$ of a matrix $A \in \mathbf{C}^{r\times t}$ is the minimal number of $\mathbf{C}$-operations (= additions/subtractions/scalar multiplications) which are sufficient to compute $A \cdot x$ from x, where the input vector $x = (x_i)$ is a column vector of t indeterminates $x_1, \cdots, x_t$ over $\mathbf{C}$. Since a nonabelian group G has more than one Fourier transform, we define the *linear complexity* of the group G by $L_s(G) := \min L_s(W)$, where the minimum is taken over all possible Fourier transforms W for $\mathbf{C}G$. Trivially, $L_s(G) < 2 \cdot |G|^2$. For a cyclic 2-group G the Cooley-Tukey algorithm shows that $L_s(G) \leqq 3/2|G| \log |G| - |G| + 1$. (In this paper log will always mean $\log_2$.) Combining the variants of the FFT algorithms [1], [4], [5], [8], [12], [14], [15], [17], [18] we get for an arbitrary finite abelian group G

$$L_s(G) \leqq c \cdot |G| \log |G| - |G| + 1,$$

where $c \leqq 15$ depends on the prime divisors of $|G|$ and on the exponents of the Sylow subgroups of G (see the proof of Theorem 2.4).

The question is near at hand, whether an $O(|G| \log |G|)$ upper bound does also hold for the linear complexity of a wider class of groups. We answer this question in the affirmative by extending such an upper bound to all finite metabelian groups G (see Theorem 3.1). (Recall that G is metabelian if and only if G has an abelian normal subgroup A such that G/A is abelian. Dihedral and generalized quaternion groups are typical members of this class, as well as all p-groups G with $|G| \leqq p^5$. By a theorem of Hall, all groups of square-free order are also metabelian.) Furthermore we show that metabelian groups also have fast inverse Fourier transforms.

To prove this result we had to change the former strategies. Using a Fourier transform adapted to a composition series of G, Beth [2], [3] proved an $O(|G|^{3/2})$ upper bound for the linear complexity of soluble groups. This result can be essentially extended to arbitrary finite groups (see [6]) by choosing Fourier transforms adapted to a maximal chain of subgroups of G. In particular, for symmetric groups this technique yields the stronger estimate $L_s(S_n) = o(|S_n| \cdot \log^3 |S_n|)$ (see [6], [7]). To establish an FFT for metabelian groups we first fix a maximal abelian normal subgroup A of G containing the commutator subgroup G' of G. Then we evaluate the DFT corresponding

to A at $[G:A]$ suitable elements of $\mathbf{C}A$. By a result of Shoda [16] (see also Theorem 2.5), we know that every irreducible representation D of G is closely related to a one-dimensional representation (= linear character) of a normal subgroup B of G containing A. As a matter of fact, B depends only on the kernel of D. Choosing a DFT for $\mathbf{C}G$ that takes Shoda's theorem and the action of various character groups of type $X(B/A)$ into account, we get an FFT for $\mathbf{C}G$ along "global" FFTs for A and "local" FFTs for several groups B/A. Section 3 will make this statement more precise.

The major development of the paper is self-contained apart from some proof details of classical representation theory, which can be found in [9], [10].

2. Preliminaries. In the next section we will assume familiarity with some basic techniques of complexity and representation theory. The present section puts together the definitions and results we will need. The first lemma is concerned with the linear complexity of matrices. (When treating matrices the symbols $\oplus$ and $\otimes$ will always denote the direct sum and Kronecker product, respectively.)

LEMMA 2.1. *For $A \in \mathbf{C}^{a\times a}$, $B \in \mathbf{C}^{b\times b}$ and a-square permutation matrices P and Q we have*

(a) $L_s(A) = L_s(PAQ)$,

(b) $L_s(A \oplus B) \leqq L_s(A) + L_s(B)$,

(c) $L_s(A \cdot B) \leqq L_s(A) + L_s(B)$ *(assuming $a = b$)*,

(d) $L_s(A \otimes B) \leqq b \cdot L_s(A) + a \cdot L_s(B)$.

Proof. Claims (a), (b), and (c) are easy exercises. To prove (d), recall that the Kronecker product of matrices satisfies $A \otimes B = (A \otimes E_b) \cdot (E_a \otimes B)$, where E_a denotes the a-square unit matrix. Finally observe that $E_a \otimes B = B \oplus \cdots \oplus B$ (a times) and $A \otimes E_b = P(E_b \otimes A)Q$, for suitable permutation matrices P and Q. Claim (d) then follows from (a), (b), and (c). This proves Lemma 2.1.

The Kronecker product formula (d) yields an upper bound for the linear complexity of a direct product of finite groups.

LEMMA 2.2. *Let $G = G_1 \times \cdots \times G_r$ be the direct product of finite groups. Then*

$$L_s(G) \leqq \sum_{i=1}^{r} [G:G_i] L_s(G_i).$$

Proof. Let W_i be a Fourier transform for $\mathbf{C}G_i$ satisfying $L_s(G_i) = L_s(W_i)$. Then $W := W_1 \otimes \cdots \otimes W_r$ is a Fourier transform for $\mathbf{C}G$ (see [10, p. 516]). Hence Lemma 2.2 follows by induction from the Kronecker product formula, Lemma 2.1(d).

For $c > 0$ let FFT_c denote the class of all finite groups G that have a Fourier transform W satisfying $\max(L_s(W), L_s(W^{-1})) \leqq c \cdot |G| \cdot \log|G|$.

LEMMA 2.3. *FFT_c is closed under direct products.*

Proof. Let $G_1, \cdots, G_r \in FFT_c$, and let W_i be a Fourier transform for $\mathbf{C}G_i$ satisfying $\max(L_s(W_i), L_s(W_i^{-1})) \leqq c \cdot |G_i| \cdot \log|G_i|$. Then $W := W_1 \otimes \cdots \otimes W_r$ is a Fourier transform for $\mathbf{C}G$, $G := G_1 \times \cdots \times G_r$. By Lemma 2.2,

$$L_s(W) = L_s(W_1 \otimes \cdots \otimes W_r) \leqq \sum_i [G:G_i] L_s(W_i) \leqq c \cdot |G| \log|G|.$$

Since $W^{-1} = W_1^{-1} \otimes \cdots \otimes W_r^{-1}$, the same upper bound can be analogously derived for the linear complexity of W^{-1}. This proves Lemma 2.3.

Combining several variants of the classical FFT algorithms we get the following theorem.

THEOREM 2.4. (a) *$FFT_{3/2}$ contains all finite abelian 2-groups.*

(b) *$FFT_{2.2}$ contains all finite abelian 3-groups.*

(c) *FFT_{15} contains all finite abelian groups.*

Proof. (a) and (b). Every finite abelian group is a direct product of cyclic groups. Hence, by Lemma 2.3, it suffices to prove the statements for cyclic groups. Let C_n denote the cyclic group of order n, and let $\omega \in \mathbf{C}$ be a primitive nth root of unity. We have to estimate the linear complexity of the n-square DFT matrix $(\omega^{\mu\nu})_{0\leqq\mu,\nu<n}$ and its inverse

$$(\omega^{\mu\nu})^{-1} = \frac{1}{n}(\omega^{-\mu\nu}). \tag{1}$$

For every input vector $(a_\nu)_{0\leqq\nu<n}$ we thus have to compute

$$(A_\mu)_{0\leqq\mu<n} := (\omega^{\mu\nu})(a_\nu) \tag{2}$$

as well as

$$(B_\mu)_{0\leqq\mu<n} := \frac{1}{n}(\omega^{-\mu\nu})(a_\nu) \tag{3}$$

with at most $O(n \log n)$ **C**-operations. If $n = pq$, the indices μ and ν can be uniquely written as $\mu = kq + l (0 \leqq k < p,\ 0 \leqq l < q)$ and $\nu = ip + j (0 \leqq i < q,\ 0 \leqq j < p)$. Hence

$$\begin{aligned} A_{kq+l} &= \sum_{i<q} \sum_{j<p} a_{ip+j} \omega^{(ip+j)(kq+l)} \\ &= \sum_{j<p} \left(\sum_{i<q} a_{ip+j} (\omega^p)^{il} \right) \omega^{jl} (\omega^q)^{jk}. \end{aligned} \tag{4}$$

The last formula is the basis for the Cooley–Tukey algorithm [8] to compute the A_μ: first compute for all $j < p$ the expressions $b_{j,l} := \sum_{i<q} a_{ip+j}(\omega^p)^{il}$, $0 \leqq l < q$, by a cyclic DFT of order q. Then multiply the $b_{j,l}$ by the so-called twiddle factors ω^{jl} and finally compute for all $l < q$ the spectral coefficients $A_{kq+l} (0 \leqq k < p)$ by a cyclic DFT of order p. Altogether we get the recurrence relation $L_s(C_{pq}) \leqq p \cdot L_s(C_q) + q \cdot L_s(C_p) + pq - p - q + 1$. By induction, this yields for cyclic p-groups $L_s(C_{p^m}) \leqq (L_s(C_p) + p - 1)mp^{m-1} - p^m + 1$. Combining this with $L_s(C_2) = 2$ and $L_s(C_3) \leqq 8$ we get

$$G \simeq C_{2^m} \Rightarrow L_s(G) \leqq \tfrac{3}{2}|G| \log |G| - |G| + 1, \tag{5}$$

and

$$G \simeq C_{3^m} \Rightarrow L_s(G) \leqq 2.11|G| \log |G| - |G| + 1. \tag{6}$$

In order to get a fast algorithm that computes the B_μ (see (3)) we modify the right-hand side of equation (4) as follows. We replace ω by ω^{-1} and each twiddle factor ω^{jl} by ω^{-jl}/n. This proves (a) and (b).

(c) We partly follow the approach of Bluestein [4] (see also Büchi [5]) and show that for every n

$$L_s(C_n) \leqq 15n \log n - n + 1. \tag{7}$$

Furthermore we will prove that the same upper bound is valid for the inverse transformation. We are going to compute the spectral coefficients A_μ (see (2)) with a fast cyclic convolution of length a power of 2. First observe that $2\mu\nu = \mu^2 + \nu^2 - (\mu - \nu)^2$. Hence

$A_\mu = \omega^{\mu^2/2} \sum_{\nu<n} a_\nu \omega^{\nu^2/2} \omega^{-(\mu-\nu)^2/2}$ and with $u_j := a_j \omega^{j^2/2}$ and $v_k := \omega^{-k^2/2}$ we get

$$\omega^{-\mu^2/2} A_\mu = \sum_{0 \leqq \nu < n} u_\nu v_{(\mu-\nu) \bmod n}.$$

Let $C_N = \langle x \rangle$ denote the cyclic group of order $N := 2^{\lceil \log 2n \rceil}$. Then $U := \sum_{0 \leqq i < n} u_i (x^i + x^{N-n+i})$ as well as $V := \sum_{0 \leqq j < n} v_j x^j$ are elements of the group algebra $\mathbf{C}C_N$ and a straightforward computation shows that the first n coefficients of the product $U * V$ are equal to $\omega^{-\mu^2/2} A_\mu$, $(0 \leqq \mu < n)$. These coefficients can be computed along the formula $U * V = DFT_N^{-1}(DFT_N(U) \cdot DFT_N(V))$. Now observe that V is independent of the input. Thus we can precompute $DFT_N(V)$ and store it as a constant vector. To compute U we need at most $n-1$ operations. According to the special form of U we can save $2(N-2n)$ operations in the first level of the computation of $DFT_N(U)$; thus, according to (5), after at most $\frac{3}{2}N \log N - 3N + 5n$ steps we know $DFT_N(U)$. After the pointwise multiplication of the two transformed vectors we compute only the first n coefficients of the inverse of the resulting vector. Combining the final multiplications with $\omega^{\mu^2/2}$ and the factor N^{-1} corresponding to DFT_N^{-1}, this transformation needs at most $\frac{3}{2}N \log N - 2N + 2n + 1$ arithmetic operations. Now let $n \geqq 41$. (If $n \leqq 40$ then the trivial upper bound $2n^2 - 3n + 1$ does not exceed $15n \log n - n + 1$.) Since $N = cn$ for some $2 \leqq c \leqq 4$, we altogether get

$$L_s(G) \leqq 3cn \log n + 3cn \log c - 4cn + 7n + 1.$$

As $3c \log c - 4c$ is monotonically increasing in the relevant interval $2 \leqq c \leqq 4$, we get

$$L_s(G) \leqq 12n \log n + 16n - n + 1 \leqq 12n \log n + \frac{16}{\log 41} n \log 41 - n + 1$$

$$\leqq 15n \log n - n + 1.$$

Recalling that $(\omega^{\mu\nu})^{-1} = 1/n(\omega^{-\mu\nu})$ and multiplying in the final step by the precomputed values $\omega^{\mu^2/2} N^{-1} n^{-1}$ we see that the same upper bound is valid for the inverse transformation. This proves Theorem 2.4.

Now we begin our review of representation theory. In the introduction we already mentioned the notions of irreducibility and equivalence of representations of a group algebra. In the sequel it will sometimes be convenient for us to have a second concept at our disposal that is equivalent to the above concept. Let G be a finite group. A **C**-*representation* of G of *degree* or *dimension* d is a group morphism $D: G \to GL(d, \mathbf{C})$. By linear extension of D, we get a representation of the corresponding group algebra. By abuse of notation, this extension will also be denoted by D. D is *irreducible* if and only if its extension is. Two representations D and D' of G are *equivalent*, $D \sim D'$, if and only if the corresponding extensions are equivalent. By Maschke's theorem, every **C**-representation D of G is equivalent to a direct sum $D_1 \oplus \cdots \oplus D_r$ of irreducible **C**-representations D_i of G. This decomposition into irreducibles is essentially unique; in particular, the multiplicity $\langle D_i | D \rangle$, which is the number of j satisfying $D_j \sim D_i$, depends only on D and the equivalence class $[D_i]$ of D_i. By definition, D is *multiplicity-free* if and only if $\langle D_i | D \rangle \leqq 1$ for all i. Without explicitly knowing a decomposition of D into irreducibles, we can compute these multiplicities with the help of characters: Every representation D is associated with its *character* $\chi^D: G \to \mathbf{C}$, defined by $\chi^D(g) := \operatorname{trace}(D(g))$ for all $g \in G$. As a matter of fact, equivalent representations have the same character. The characters corresponding to the classes of irreducible representations are called *irreducible* characters. Characters are class functions on G, i.e., they are constant on the conjugacy classes of G. The set of all class functions on G becomes

a **C**-space under pointwise **C**-operations. It turns out that the irreducible characters form an orthonormal **C**-basis of this space under the inner product

$$\langle \phi \,|\, \psi \rangle := \frac{1}{|G|} \sum_{g \in G} \phi(g)\psi(g^{-1}).$$

According to this fact, the multiplicity $\langle D_i \,|\, D \rangle$ equals $\langle \chi^{D_i} \,|\, \chi^D \rangle$. If D is a d-dimensional representation of G, then G acts via D on $\mathbf{C}^d$. Let $\chi_1, \cdots, \chi_h$ be the irreducible characters of G. Then

$$\varepsilon_i := \frac{\chi_i(1)}{|G|} \sum_{g \in G} \chi_i(g^{-1})g$$

is a (primitive) idempotent of the center of $\mathbf{C}G$ and

$$\mathbf{C}^d = D(\varepsilon_1)\mathbf{C}^d \oplus \cdots \oplus D(\varepsilon_h)\mathbf{C}^d$$

is the decomposition of $\mathbf{C}^d$ into its *isotypic components*. If D_i is an irreducible representation of G with character χ_i then the action of G on $D(\varepsilon_i)\mathbf{C}^d$ affords a representation that is equivalent to $D_i \oplus \cdots \oplus D_i$ ($\langle D_i \,|\, D \rangle$ times). The simplest irreducible representations of G are those of degree 1. In this case the representation D is equal to its character χ^D. One-dimensional representations of G are usually called *linear characters* of G. They can be viewed as morphisms from G into the multiplicative group $\mathbf{C}^*$ of the complex field. Under pointwise multiplication, $(\chi \otimes \psi)(g) := \chi(g) \cdot \psi(g)$, the linear characters of G form an abelian group $X(G)$, the *character group* of G. If N is a normal subgroup of G, then $X(G/N)$ can be embedded into the character group $X(G)$ of G via the natural projection $G \to G/N$. Thus $X(G/N)$ "is" the set of all linear characters of G that have N in its kernel. For a subgroup U of G let $X(G)\downarrow U$ denote the subgroup of $X(U)$ consisting of all restrictions $\chi \downarrow U$, where χ is in $X(G)$. Using standard techniques it can be shown that

$$X(G)\downarrow U \simeq U/(G' \cap U). \tag{8}$$

In particular, $X(G)$ is isomorphic to the commutator factor group G/G'. Thus $X(G) \simeq G$ if and only if G is abelian. If G is not abelian, then G has at least one equivalence class of higher-dimensional irreducible representations. In case of a metabelian group G, every higher-dimensional irreducible representation D is closely related to a linear character of a subgroup U of G. To make this statement more precise, we need the concepts of induced and monomial representations. If U is a subgroup of the finite group G and F an f-dimensional **C**-representation of U, then the *induced representation* $F\uparrow G$ is a **C**-representation of G of degree $f \cdot [G : U]$. More precisely, let $u_1, \cdots, u_n$ be a transversal of the left cosets of U in G. Then, for all $g \in G$,

$$(F\uparrow G)(g) = [\dot{F}(u_i^{-1} g u_j)]_{1 \leqq i,j \leqq n} \tag{9}$$

where

$$\dot{F}(x) := \begin{cases} F(x) & \text{if } x \in U, \\ 0 & \text{if } x \in G \setminus U. \end{cases}$$

(The linear extension of $F\uparrow G$ to $\mathbf{C}G$ will sometimes be denoted by $F\uparrow \mathbf{C}G$.) Induction of representations is transitive up to permutational equivalence: If F is a representation of U and V is a subgroup of G containing U, then for some permutation matrix T and all $g \in G$ we have $T(F\uparrow G)(g)T^{-1} = ((F\uparrow V)\uparrow G)(g)$. Dual to induction is the concept of subduction (= restriction) of representations. Both concepts are linked by

the Frobenius Reciprocity Theorem (see, e.g., [10, p. 555]): If F and D are irreducible $\mathbf{C}$-representations of U and G, respectively, then the multiplicity of D in $F\uparrow G$ equals the multiplicity of F in the subduced representation $D\downarrow U$:

$$\langle D \mid F\uparrow G\rangle = \langle D\downarrow U \mid F\rangle.$$

If F is one-dimensional and $g\in G$ then, by (9), $(F\uparrow G)(g)$ is a *monomial* matrix, i.e., $(F\uparrow G)(g)$ has exactly one nonzero entry in each row and in each column. This fact justifies calling $F\uparrow G$ a *monomial* representation. A group G is called an *M-group* if and only if every irreducible representation of G is equivalent to a monomial one. The following result is essentially due to Shoda [16]. It shows that metabelian groups are M-groups and specifies the subgroups that are relevant for induction to get all equivalence classes of irreducible $\mathbf{C}$-representations of a metabelian group G represented by monomial representations.

THEOREM 2.5. *Let G be a finite metabelian group and let D be an irreducible $\mathbf{C}$-representation of G with kernel N. Let A be a subgroup of G containing $G'\cdot N$ such that A/N is a maximal abelian normal subgroup of G/N. Then D is equivalent to a monomial $\mathbf{C}$-representation $\alpha\uparrow G$ of G, where α is a linear character of A.*

Proof. Let D be an irreducible d-dimensional $\mathbf{C}$-representation of G with kernel N. If $d=1$ then $N\geqq G'$; thus $A=G$ and $\alpha=D$. Since all irreducible $\mathbf{C}$-representations of an abelian group G are one-dimensional, we are left with the case $G'\neq E$ and $d>1$. The rest of the proof is by induction on $|G|$.

Step 1. *Reduction to the case $N=E$.* If $N>E$, let ν be the projection $G\to G/N$. Then $D=D'\circ\nu$, for some *faithful* (i.e., injective) irreducible $\mathbf{C}$-representation D' of the metabelian group G/N. By induction, the theorem applies to D'. Thus there exists a subgroup A of G such that A/N is a maximal abelian normal subgroup of G/N containing $(G/N)'=G'N/N$. Furthermore, there is a linear character α' of A/N such that $D'\sim\alpha'\uparrow G/N$. Finally an easy computation shows that the linear character $\alpha:=\alpha'\circ(\nu\downarrow A)$ of A satisfies $D\sim\alpha\uparrow G$.

Step 2. Suppose $N=E$. Let A be a maximal abelian normal subgroup of G containing G'. Then, by Clifford's theorem (see, e.g., [10, p. 565]) G acts transitively on the isotypic constituents of $D\downarrow A$. Let $T\geqq A$ be the stabilizer (=inertia group) of such an isotypic constituent $\alpha\oplus\cdots\oplus\alpha$ of $D\downarrow A$. Again by Clifford's theorem, there exists an irreducible $\mathbf{C}$-representation τ of T such that $\tau\downarrow A=\alpha\oplus\cdots\oplus\alpha$ and $D\sim\tau\uparrow G$.

Case 1. $T=A$.

Then $\tau=\alpha$, $D\sim\alpha\uparrow G$, and $\deg(\alpha)=1$, since A is abelian. (We have to exclude the remaining cases.)

Case 2. $T=G$.

Since A is abelian and $D\downarrow A$ is isotypic, we have $D(a)=\alpha(a)\cdot E_d$ for all $a\in A$. Hence $D(A)$ is contained in the center of $D(G)$. As D is faithful, this implies that A is contained in the center $C(G)$ of G. Since $A\geqq G'$ is a maximimal abelian normal subgroup of G we get $G'\leqq A=C(G)<G$. Let $g\in G\backslash A$. Then the subgroup of G generated by A and g is an abelian normal subgroup of G, contradicting the maximality of A.

Case 3. $A<T<G$.

By induction, the theorem applies to the irreducible $\mathbf{C}$-representation τ of the metabelian group T. Thus for some subgroup B strictly containing A and for some one-dimensional representation β of B, we know that $\tau\sim\beta\uparrow T$. Since induction is transitive up to permutational equivalence, we get $D\sim\beta\uparrow G$. But then $N=\ker D=\bigcap_{g\in G} g\ker(\beta)g^{-1}\geqq B'>E$, contradicting the fact that D is a faithful representation of G. This proves Theorem 2.5.

3. FFT for metabelian groups. This section is devoted to the design and analysis of FFTs for metabelian groups. Our goal is to prove the following result.

THEOREM 3.1. (a) *$FFT_{3/2}$ contains all finite metabelian 2-groups.*

(b) *$FFT_{2.2}$ contains all finite metabelian 3-groups.*

(c) *FFT_{15} contains all finite metabelian groups.*

Proof. Let G be a finite metabelian group. By Theorem 2.4 we may assume that G is not abelian. Thus $G > G' > G'' = E$. Let A be a maximal abelian normal subgroup of G containing G'. We are going to specify a transversal $D_1, \cdots, D_h$ of irreducible $\mathbf{C}$-representations of G which will be the basis for a fast (inverse) Fourier transform. According to Theorem 2.5 we may assume that the D_i satisfy the following condition.

(i) For every i there exists a normal subgroup B_i of G containing A and a linear character β_i of B_i such that $D_i = \beta \uparrow G$.

To establish a second condition we let the character group $X(G/A)$ act on the set $\{[D_1], \cdots, [D_h]\}$ of equivalence classes of irreducible $\mathbf{C}$-representations of G by $\chi * [D_i] := [\chi \otimes D_i]$, where $(\chi \otimes D_i)(g) := \chi(gA) \cdot D_i(g)$ for all $\chi \in X(G/A)$ and all $g \in G$. Induction, subduction, and the tensor product of representations are related as follows:

$$\chi \otimes (\beta \uparrow G) = ((\chi \downarrow B) \otimes \beta) \uparrow G, \tag{10}$$

$\chi \in X(G/A)$, $A \leqq B \leqq G$, $\beta \in X(B)$. According to (8) and (10) we can postulate the following condition in addition to (i).

(ii) If $[D_i]$ and $[D_j]$ are in the same $X(G/A)$-orbit, then $B_i = B_j$ and $\beta_i = \psi \otimes \beta_j$, for some $\psi \in X(B_i/A)$.

If B is a normal subgroup of G then G acts via conjugation on $X(B)$ by $(g\beta)(b) := \beta(g^{-1}bg)$ ($g \in G$, $\beta \in X(B)$, $b \in B$). Since $g\beta = \beta$ for all $g \in B$, the G-orbits $G\beta$, $\beta \in X(B)$, have length at most $[G:B]$. The orbits of this length are of special interest for us: By (9) $\beta \uparrow G \downarrow B = \oplus_{gB \in G/B} (g\beta)$; applying the orthonormality of the irreducible characters and the Frobenius Reciprocity Theorem, we get for $\beta \in X(B)$

$$\begin{aligned} |G\beta| = [G:B] &\Leftrightarrow \beta \uparrow G \text{ is irreducible} \\ &\Leftrightarrow \beta \uparrow G \downarrow B \text{ is multiplicity-free.} \end{aligned} \tag{11}$$

Let $A \leqq b \leqq G$. We are going to combine the above group actions. To this end we define a semidirect product P of the normal subgroup $N := X(B/A) \trianglelefteq P$ and the subgroup $G < P$ by $(\psi, g)(\psi', g') := (\psi \otimes (g\psi'), gg')$. P acts on $X(B)$ by $(\psi, g)\beta := \psi \otimes (g\beta)$. According to (10) and (11), the set $\{\beta \in X(B) \mid \beta \uparrow G \text{ irreducible}\}$ is P-stable.

Now choose $\beta_1, \cdots, \beta_h$ according to (i) and (ii). Furthermore let P_i denote the semidirect product of $N_i := X(B_i/A)$ and G, as defined above. Then, by (10), (11), and condition (ii), the set of irreducible constituents of $\beta_1 \uparrow G \downarrow B_1, \cdots, \beta_h \uparrow G \downarrow B_h$ is the disjoint union of certain P_i-orbits; i.e., there is some index set $I \subseteq \{1, \cdots, h\}$ such that

$$\dot{\bigcup_{1 \leqq i \leqq h}} G\beta_i = \dot{\bigcup_{i \in I}} P_i\beta_i.$$

Every $P_i\beta_i$ further splits into N_i-orbits which are all of length $|N_i|$. More precisely, G acts on the set $\{N_i\gamma \mid \gamma \in P_i\beta_i\}$ by $g(N_i\gamma) := N_i(g\gamma)$. Let V_i be the stabilizer of the point $N_i\beta_i$ under this G-action. Then $V_i \geqq B_i$ and

$$P_i\beta_i = \dot{\bigcup_{u_iV_i \in G/V_i}} u_iN_i\beta_i = \dot{\bigcup_{u_iV_i \in G/V_i}} N_i(u_i\beta_i).$$

Our next goal is to evaluate (with a small number of **C**-operations) the Fourier transform $W := \oplus_i (\beta_i \uparrow \mathbf{C}G)$ of $\mathbf{C}G$ at an arbitrary element $y \in \mathbf{C}G$. To this end we first decompose G into the disjoint union of left cosets with respect to A: $G = \dot{\bigcup}_{1 \leqq k \leqq [G:A]} g_k A$. Then $y = \sum_k g_k a_k$, for suitable $a_k \in \mathbf{C}A$. Now let $B := B_i$ and $\beta := \beta_i$. If $G = \dot{\bigcup}_{1 \leqq j \leqq d} h_j B$ then, after renaming the g_k and a_k, we can achieve that $h_j = g_{j1}, \cdots, g_{jt} \in h_j B = \dot{\bigcup}_{1 \leqq l \leqq t} g_{jl} A$ with corresponding $a_{j1}, \cdots, a_{jt} \in \mathbf{C}A$. Thus $y = \sum_{j=1}^{d} \sum_{l=1}^{t} g_{jl} a_{jl}$. On the other hand there are $b_j \in \mathbf{C}B$ such that $y = \sum_{j=1}^{d} h_j b_j$. Combining the last three equations we have for all $j \leqq d$

$$b_j = \sum_{l=1}^{t} h_j^{-1} g_{jl} a_{jl}. \tag{12}$$

Altogether we get for the constituent $D = \beta \uparrow \mathbf{C}G$ of W evaluated at $y \in \mathbf{C}G$ the formula

$$D(y) = \sum_{j=1}^{d} D(h_j) D(b_j) = \sum_{j=1}^{d} D(h_j) \bigoplus_{gB \in G/B} (g\beta)(b_j). \tag{13}$$

Instead of evaluating D at $y \in \mathbf{C}G$ separately, it is of advantage to aim at a common evaluation of all representations in $\{\chi \otimes D \mid \chi \in X(G/A)\} \cap \{\beta_j \uparrow G \mid 1 \leqq j \leqq h\}$. With regard to (10), (12), and (13) we have to compute for any fixed j and for all $\chi \otimes \beta \in X(B/A) \otimes \beta$ the following expressions:

$$\begin{aligned}(\chi \otimes \beta)(b_j) &= \sum_{l=1}^{t} (\chi \otimes \beta)(h_j^{-1} g_{jl})(\chi \otimes \beta)(a_{jl}) \\ &= \sum_{l=1}^{t} \chi(h_j^{-1} g_{jl} A) \beta(h_j^{-1} g_{jl}) \beta(a_{jl}).\end{aligned}$$

Having computed for fixed j all products $b_{jl} := \beta(h_j^{-1} g_{jl}) \cdot \beta(a_{jl})$, we are left with the evaluation of

$$((\chi \otimes \beta)(b_j))_{\chi \in X(B/A)} = (\chi(h_j^{-1} g_{jl} A))_{\chi, l} (b_{jl})_{l \leqq t}. \tag{14}$$

The matrix $(\chi(h_j^{-1} g_{jl} A))$ is the character table of B/A; hence (14) is a DFT of the abelian group B/A of order t. Let W_A denote the Fourier transform of the abelian group A. After having computed all $W_A(a_k)$, the computation of all $(\chi \otimes \beta)(b_j)$, $\chi \in X(B/A)$, additionally costs $t-1$ multiplications (to get the b_{jl}) and $L_s(B/A)$ **C**-operations for the computation of *all* $(\chi \otimes \beta)(b_j)$. The proportion of **C**-operations to compute a single $(\chi \otimes \beta)(b_j)$ is thus $(L_s(B/A) + t - 1) t^{-1}$. (Note that according to our special choice of $D_1, \cdots, D_h$ and with regard to $P_i \beta_i = \dot{\bigcup}_{u_i} N_i(u_i \beta_i)$ *all* $(\chi \otimes \beta)(b_j)$ are really needed.) Thus , knowing all the $W_A(a_k)$, we can compute $D(b_1), \cdots, D(b_d)$ in $d^2(L_s(B/A) + t - 1) t^{-1}$ arithmetic steps, where $t = [B:A] = [G:A] \cdot d^{-1}$. Finally, knowing the $D(b_j)$, the computation of $D(y)$ via (13) additionally requires at most $(d-1) \cdot d$ multiplications. (To prove this last fact note that without loss of generality $h_1 = 1$, then use the monomiality of D, and finally observe that the concluding summation is free of costs since all the summands have their nonzero entries at pairwise disjoint positions: as D is irreducible we have $\dim D(\mathbf{C}G) = d^2$; on the other hand every $D(h_j) \cdot D(b_j)$ has its $\leqq d$ nonzero entries at the support of the monomial matrix $D(h_j)$.) Altogether the number of **C**-operations sufficient to compute $D(y)$ (knowing all the $W_A(a_k)$) is at most

$$d^2(L_s(B/A) + t - 1) t^{-1} + (d-1) \cdot d.$$

Let n_d denote the number of inequivalent irreducible representations of G of degree d. Furthermore, for every d let $t_d := [G:A] \cdot d^{-1}$. Let $m(t) := \max L_s(H)$, where the

maximum is over all abelian groups H of order t. Then, using (5), (6), (7), and the facts that $n_1 = [G:G']$ and $\sum_d n_d d^2 = |G|$ we get

$$L_s(W) \leqq [G:A]L_s(A) + \sum_d d^2 n_d \{(m(t_d) + t_d - 1)t_d^{-1} + 1 - d^{-1}\}$$

$$\leqq c|G| \log |G|,$$

with $c \leqq \frac{3}{2}$, $c \leqq 2.2$, and $c \leqq 15$ in cases (a), (b), and (c), respectively, of Theorem 3.1. This proves that metabelian groups have an FFT.

Our next goal is to show that the same upper bound does not hold for the linear complexity of W^{-1}. Given $W(y) = \oplus_i D_i(y)$, we thus have to recover $y \in \mathbf{C}G$. (The idea is to invert all local and global FFTs in the first half of the proof.) Again let $D := D_i$. As the summation over j in (13) is free of costs, the projection of $D(y)$ to the support of $D(h_j)$ equals $D(h_j) \oplus_{gB \in G/B} (g\beta)(b_j)$. By multiplying with the monomial matrix $D(h_j^{-1})$ we get $\oplus_{gB \in G/B} (g\beta)(b_j)$. Combining this information in an appropriate way, we know the left-hand side of (14). Inverting (14) with an inverse FFT we recover all the b_{jl}. Multiplying with $\beta(g_{jl}^{-1} h_j)^{-1}$ we get all the $\beta(a_{jl})$. Thus we know for all $\alpha \in X(A)$ the values $\alpha(a_k)$, $1 \leqq k \leqq [G:A]$. Finally with $[G:A]$ inverse FFTs we may recover all the a_k. Since $y = \sum_k g_k a_k$ and the above proof of the upper bound applies to the inverse of W as well, Theorem 3.1 is proved.

REFERENCES

[1] M. D. ATKINSON, *The complexity of group algebra computations*, Theoret. Comput. Sci., 5 (1977), pp. 205-209.

[2] T. BETH, *Verfahren der schnellen Fourier-Transformation*, Teubner, Stuttgart, 1984.

[3] ———, *On the computational complexity of the general discrete Fourier transform*, Theoret. Comput. Sci., 51 (1987), pp. 331-339.

[4] L. I. BLUESTEIN, *A linear filtering approach to the computation of the discrete Fourier transform*, IEEE Trans. Comput., AU-18 (1970), pp. 451-455.

[5] W. BÜCHI, *Die diskrete Fourier Transformation*, Diplomarbeit, Universität Zürich, 1979.

[6] M. CLAUSEN, *Fast generalized Fourier transforms*, Theoret. Comput. Sci., to appear.

[7] M. CLAUSEN AND D. GOLLMANN, *Spectral transforms for symmetric groups—Fast algorithms and VLSI architectures*, Proc. 3rd International Workshop on Spectral Techniques, University of Dortmund, Federal Republic of Germany, October 1988, to appear.

[8] J. W. COOLEY AND J. W. TUKEY, *An algorithm for the machine calculation of complex Fourier series*, Math. Comput., 19 (1965), pp. 297-301.

[9] C. W. CURTIS AND I. REINER, *Representation Theory of Finite Groups and Associative Algebras*, John Wiley, New York, 1962.

[10] B. HUPPERT, *Endliche Gruppen* I, Springer-Verlag, Berlin, New York, 1967.

[11] S. L. HURST, D. M. MILLER, AND J. C. MUZIO, *Spectral Techniques in Digital Logic*, Academic Press, New York, 1985.

[12] M. G. KARPOVSKY, *Fast Fourier transforms on finite non-abelian groups*, IEEE Trans. Comput., 26/10 (1977), pp. 1028-1030.

[13] M. G. KARPOVSKY, ED., *Spectral Techniques and Fault Detection*, Academic Press, New York, 1985.

[14] H. J. NUSSBAUMER, *Fast Fourier Transform and Convolution Algorithms*, Springer-Verlag, Berlin, New York, 1981.

[15] C. M. RADER, *Discrete Fourier transform when the number of data points is prime*, Proc. IEEE, 56 (1968), pp. 1107-1108.

[16] K. SHODA, *Über die monomialen Darstellungen einer endlichen Gruppe*, Proc. Phys. Math. Soc. Japan, 15 (1933), pp. 251-257.

[17] S. WINOGRAD, *On computing the discrete Fourier transform*, Proc. Nat. Acad. Sci. U.S.A., 73 (1976), pp. 1005-1006.

[18] S. WINOGRAD, *Arithmetic Complexity of Computations*, Society for Industrial and Applied Mathematics, Philadelphia, 1980.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 594–607, June 1989

012

OPTIMAL AND SUBLOGARITHMIC TIME RANDOMIZED PARALLEL SORTING ALGORITHMS*

SANGUTHEVAR RAJASEKARAN† AND JOHN H. REIF†

Abstract. This paper assumes a parallel RAM (random access machine) model which allows both concurrent reads and concurrent writes of a global memory.

The main result is an optimal randomized parallel algorithm for INTEGER_SORT (i.e., for sorting n integers in the range $[1, n]$). This algorithm costs only logarithmic time and is the first known that is *optimal*: the product of its time and processor bounds is upper bounded by a linear function of the input size. Also given is a deterministic sublogarithmic time algorithm for *prefix sum*. In addition this paper presents a sublogarithmic time algorithm for obtaining a random permutation of n elements in parallel. And finally, sublogarithmic time algorithms for GENERAL_SORT and INTEGER_SORT are presented. Our sublogarithmic GENERAL_SORT algorithm is also optimal.

Key words. randomized algorithms, parallel sorting, parallel random access machines, random permutations, radix sort, prefix sum, optimal algorithms

AMS(MOS) subject classification. 68Q25

1. Introduction.

1.1. Sequential sorting algorithms. Sorting is one of the most important problems not only of computer science but also of every other field of science. The importance of efficient sorting algorithms has been long realized by computer scientists. Many application programs like compilers, operating systems, etc., use sorting extensively to handle tables and lists. Due to both its practical value and theoretical interest, sorting has been an attractive area of research in computer science.

The problem of sorting a sequence of elements (also called *keys*) is to rearrange this sequence in either ascending order or descending order. When the keys to be sorted are *general*, i.e., when the keys have no known structure, a lower-bound result [1] states that any sequential algorithm (on the random access machine (RAM) and many other sequential models of interest) will require at least $\Omega(n \log n)$ time to sort a sequence of n keys. Many *optimal* algorithms like QUICK_SORT and HEAP_SORT, whose run times match this lower bound, can be found in the literature [1].

In computer science applications, more often, the keys to be sorted are from a finite set. In particular, the keys are integers of at most a polynomial (in the input size) magnitude. For keys with this special property, sorting becomes much simpler. If each one of the n elements in a sequence is an integer in the range $[1, n]$ we call these keys *integer keys*. The BUCKET_SORT algorithm [1] sorts n integer keys in $O(n)$ sequential steps. Notice that the run time of BUCKET_SORT matches the trivial $\Omega(n)$ lower bound for this problem.

In this paper we are concerned with randomized parallel algorithms for sorting both *general keys* and *integer keys*.

1.2. Known parallel sorting algorithms. The performance of a parallel algorithm can be specified by bounds on its principal resources viz., processors and time. If we

* Received by the editors January 17, 1987; accepted for publication (in revised form) August 19, 1987. A preliminary version of this paper appeared as "An optimal parallel algorithm for integer sorting" in the 18th IEEE Symposium of Foundations of Computer Science, Portland, Oregon, October 1985. This work was supported by National Science Foundation grant DCR-85-03251 and Office of Naval Research contract N00014-80-C-0647.

† Aiken Computation Laboratory, Harvard University, Cambridge, Massachusetts 02138.

let P denote the processor bound, and T denote the time bound of a parallel algorithm for a given problem, the product PT is, clearly, lower bounded by the minimum sequential time, T_s, required to solve this problem. We say a parallel algorithm is *optimal* if $PT = O(T_s)$. Discovering optimal parallel algorithms for sorting both *general* and *integer keys* remained an open problem for a long time.

Reischuk [25] proposed a randomized parallel algorithm that used n synchronous PRAM processors to sort n *general keys* in $O(\log n)$ time. This algorithm, however, is impractical owing to its large word-length requirements. Reif and Valiant [24] presented a randomized sorting algorithm that ran on a fixed-connection network called *cube-connected cycles* (CCC). This algorithm employed n processors to sort n *general keys* in time $O(\log n)$. Since $\Omega(n \log n)$ is a sequential lower bound for this problem, their algorithm is indeed optimal. Simultaneously, Atai, Komlós, and Szemerédi [4] discovered a deterministic parallel algorithm for sorting n *general keys* in time $O(\log n)$ using a sorting network of $O(n \log n)$ processors. Later, Leighton [17] showed that this algorithm could be modified to run in $O(\log n)$ time on an n-node fixed-connection network.

As in the sequential case, many parallel applications of interest need only to sort *integer keys.* Until now, no optimal parallel algorithm existed for sorting n *integer keys* with a run time of $O(\log n)$ or less.

1.3. Some definitions and notations. Given a sequence of keys $k_1, k_2, \cdots, k_n$ drawn from a set S having a linear order $<$, the problem of *sorting* this sequence is to find a permutation σ such that $k_{\sigma(1)} < k_{\sigma(2)} < \cdots < k_{\sigma(n)}$.

By *general keys* we mean a sequence of n elements drawn from a linearly ordered set S whose elements have no known structure. The only operation that can be used to gain information about the sequence is the comparison of two elements.

GENERAL_SORT is the problem of sorting a sequence of general keys, and INTEGER_SORT is the problem of sorting a sequence of integer keys.

Throughout this paper we let $[m]$ stand for $\{1, 2, \cdots, m\}$.

A sorting algorithm is said to be *stable* if equal elements remain in the same relative order in the sorted sequence as they were in originally. In more precise terms, a sorting algorithm is stable if on input $k_1, k_2, \cdots, k_n$, the algorithm outputs a sorting permutation σ of $(1, 2, \cdots, n)$ such that for all $i, j \in [n]$, if $k_i = k_j$ and $i < j$ then $\sigma(i) < \sigma(j)$. A sorting algorithm that is not guaranteed to output a stable sorted sequence is called *nonstable.*

Just as the big-O function serves to represent the complexity bounds of deterministic algorithms, we employ $\tilde{O}$ to represent complexity bounds of randomized algorithms. We say a randomized algorithm has resource (like time, space, etc.) bound $\tilde{O}(g(n))$ if there is a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geqq 1 - 1/n^{\alpha}$ for any $\alpha > 1$.

1.4. Our model of computation. We assume the CRCW PRAM (concurrent-read concurrent-write parallel RAM) model proposed by Shiloach and Vishkin [26]. In a PRAM model, a number (say P) of processors work synchronously communicating with each other with the help of a common block of memory. Each processor is a RAM. A single step of a processor is an arithmetic operation, a comparison, or a memory access. CRCW PRAM is a version of PRAM that allows both concurrent writes and concurrent reads of shared memory. Write conflicts are resolved by priority.

All the algorithms given in this paper, except the prefix sum algorithm, are randomized. Every processor, in addition to the operations allowed by the deterministic version of the model, is also capable of making independent (n-sided) coin flips. Our

stated resource bounds will hold for the worst-case input with overwhelming probability.

1.5. Contents of this paper. Our main contributions in this paper are

1) An optimal parallel algorithm for INTEGER_SORT. This algorithm uses $n/\log n$ processors and sorts n integer keys in time $\tilde{O}(\log n)$, and
2) Sublogarithmic time algorithms for GENERAL_SORT and INTEGER_SORT. GENERAL_SORT algorithm employs $n(\log n)^{\varepsilon}$ (for any $\varepsilon > 0$) processors, and INTEGER_SORT algorithm employs $n(\log\log n)^2/\log n$ processors. Both these algorithms run in time $\tilde{O}(\log n/\log\log n)$.

The problem of optimal parallel sorting of n integers in the range $[n^{O(1)}]$ still remains an open problem. Our sublogarithmic time algorithm for GENERAL_SORT is optimal as implied by a recent result of Alon and Azar [2].

In our sublogarithmic time sorting algorithms we reduce the problem of sorting to the problem of *prefix-sum computation.* We show in this paper that *prefix sum* can be computed in time $O(\log n/\log\log(P\log n/n))$ using $P \geqq n/\log n$ processors. We also present a sublogarithmic time algorithm for computing a random permutation of n given elements with a run time of $\tilde{O}(\log n/\log\log n)$ using $n(\log\log n)^2/\log n$ processors.

Some of the results of this paper appeared in preliminary form in [22], but are substantially simplified in this manuscript. In § 2 we present some relevant preliminary results. Section 3 contains our optimal INTEGER_SORT algorithm. In § 4 we describe our sublogarithmic time algorithms.

2. Preliminary results.

2.1. Prefix circuits. Let Σ be a domain and let $\circ$ be an associative operation that takes $O(1)$ sequential time over this domain. The *prefix-computation problem* is defined as follows

- **input** $(X(1), X(2), \cdots, X(n)) \in \Sigma^n$
- **output** $(X(1), X(1)\circ X(2), \cdots, X(1)\circ X(2)\circ\cdots\circ X(n))$.

The special case of prefix computation when Σ is the set of all natural numbers and $\circ$ is integer addition is called *prefix-sum computation.* Ladner and Fischer [18] show that prefix computation can be done by a circuit of depth $O(\log n)$ and size n. The processor bound of this algorithm can be improved as follows.

LEMMA 2.1. *Prefix computation can be done in time* $O(\log n)$ *using* $n/\log n$ PRAM *processors.*

Proof. Given $X(1), X(2), \cdots, X(n)$, each one of the $n/\log n$ processors gets $\log n$ successive keys. Every processor sequentially computes the prefix sum of the $\log n$ keys given to it in $\log n$ time. Let $S(i)$ be the sum of all the $\log n$ keys given to processor i (for $i = 1, \cdots, n/\log n$). Then, $n/\log n$ processors collectively compute the prefix sum of $S(1), S(2), \cdots, S(n/\log n)$, using Ladner and Fischer's [18] algorithm. Using this prefix sum, each processor sequentially computes $\log n$ prefixes of the original input sequence. □

The above idea of processor improvement was originally used by Brent in his algorithm for expression evaluation, and hence we attribute Lemma 2.1 to him. Recently Cole and Vishkin [9] have proved the following lemma.

LEMMA 2.2. *Prefix-sum computation of* n *integers* ($O(\log n)$ *bits each*) *can be performed in* $O(\log n/\log\log n)$ *time using* $n \log\log n/\log n$ CRCW PRAM *processors.*

2.2. An assignment problem. We are given a set $Q = \{1, 2, \cdots, n\}$ of n indices. Each index belongs to exactly one of m groups $G_1, G_2, \cdots, G_m$. Let g_i stand for the

number of indices belonging to group G_i, $i=1, \cdots, m$. We are given a sequence $N(1), N(2), \cdots, N(m)$ where $\sum_{i=1}^{m} N(i) = O(n)$ and $N(i)$ is an upper bound for g_i, $i=1, 2, \cdots, m$. The problem is to find in parallel a permutation of $(1, 2, \cdots, n)$ in which all the indices belonging to G_1 appear first, all the indices belonging to G_2 appear next, and so on. (Assume that given an index i, the group $G_{i'}$ to which i belongs can be found in $O(1)$ time.)

As an example, if $n=5$, $m=2$, $G_1=\{2, 5\}$, $G_2=\{1, 3, 4\}$, then $(5, 2, 1, 3, 4)$ and $(2, 5, 3, 1, 4)$ are (two of the) valid answers.

LEMMA 2.3. *The above assignment problem can be solved in* $\tilde{O}(\log n)$ *parallel time using* $n/\log n$ PRAM *processors.*

Proof. We present an algorithm. We use a shared memory of size $2\sum_{i=1}^{m} N(i)$ $(=L$, say). This memory is divided into m blocks, $B_1, B_2, \cdots, B_m$ the size of B_i being $2N(i)$. A unique assignment for the indices belonging to G_i will be found in the block B_i, for $i=1, 2, \cdots, m$.

Each one of the $P(=n/\log n)$ processors is given $\log n$ successive indices. Precisely, processor π is given the indices $(\pi-1)\log n+1, (\pi-1)\log n+2, \cdots, \pi \log n$, for $\pi=1, 2, \cdots, P$. There are three phases of the algorithm. In the first phase, boundaries of the m blocks are computed. In the second phase every processor sequentially finds unique assignments for the $\log n$ indices given to it in their **respective** blocks. In the third phase, a prefix-sum computation is done to eliminate the unused cells, and the position of each index in the output is read. Details follow.

Step 1.

P processors collectively do a prefix sum of $(N(1), N(2), \cdots, N(m))$ and hence compute the boundaries of blocks in the common memory.

Step 2.

Each processor π is given a total time of $d \log n$ (d being a constant to be fixed) to find assignments for all its indices sequentially.

π starts with its first index (call it) l. If $G_{l'}$ is the group to which l belongs, π chooses a random cell in $B_{l'}$ and tries to write its identification in it. If the chosen cell did not contain the identification of any other processor and π succeeds in writing, then that cell is assigned to l. The probability of success in one trial is $\geqq \frac{1}{2}$. If π has failed in this trial, then it tries as many times as it takes to find an assignment for l and then it takes up the next index.

After $d \log n$ steps, even if there is a single processor that has not found assignments for all its keys, the algorithm is aborted and started anew.

Step 3.

Each processor π writes a 1 in the cells that have been assigned to its indices. Unassigned cells in the common memory will have 0's. P processors perform a prefix sum computation on the contents of the memory cells $(1, 2, \cdots, L)$. Finally, every processor reads out from the prefix sum the position of each one of its indices in the output.

Analysis. Steps 1 and 3 can be completed in $O(\log n)$ time in accordance with Lemma 2.1.

In Step 2, the probability that a particular processor π successfully finds an assignment for one of its keys in a single trial is $\geqq \frac{1}{2}$. Let Y be the random variable equal to the number of successes of π in $d \log n$ trials. We require Y to be $\geqq \log n$

for every processor. Clearly Y is lower bounded by a binomial variable (see Appendix A for definitions) with parameters $(d \log n, \frac{1}{2})$. It follows from the Chernoff bounds (see Appendix A, equation (3)) that the probability that there will be at least a single processor that has not found assignments for all of its indices after $d \log n$ trials can be made $\leqq n^{-\alpha}$ for any $\alpha \geqq 1$, if we choose a proper constant d. Therefore the whole algorithm runs in time $\tilde{O}(\log n)$. This completes the proof of Lemma 2.3. □

It should be mentioned here that when the number of groups, m, is 1, the above algorithm outputs a random permutation of $(1, 2, \cdots, n)$. An algorithm for this special case was given by Miller and Reif [19].

2.3. Some known results. We state here the existence of optimal sequential algorithms for INTEGER_SORT and optimal parallel algorithms for GENERAL_SORT.

LEMMA 2.4. *Stable* INTEGER_SORT *of n keys can be done in time* $O(n)$ *by a deterministic sequential* RAM [1].

LEMMA 2.5. GENERAL_SORT *of n keys can be performed in time* $O(\log n)$ *using n* PRAM *processors* ([4] *and* [8]).

3. An optimal INTEGER_SORT algorithm. In this section we present an optimal algorithm for INTEGER_SORT. This algorithm employs $n/\log n$ processors and runs in time $\tilde{O}(\log n)$.

3.1. Summary of the algorithm. The main idea behind our algorithm is radix sorting [15]. As an example of radix sorting, consider the problem of sorting a sequence of two-bit decimal integers. One way of doing this is to sort the sequence with respect to the least significant bits (LSB) of the keys and then to sort the resultant sequence with respect to the most significant bits (MSB) of the keys. This will work provided, in the second sort keys with equal MSBs will remain in the same relative order as they were in originally. In other words, the second sort should be stable.

We have a sequence of keys $k_1, k_2, \cdots, k_n \in [n]$, where each key is a $\log n$-bit integer. We first (nonstable) sort this sequence with respect to the $(\log n - 3 \log\log n)$ LSBs of the keys. (Call this sort *Coarse_Sort.*) In the resultant sequence we apply a stable sort with respect to the $3 \log\log n$ MSBs of the keys. (Call this sort *Fine_Sort.*)

Even though the sequential time complexity of stable sort is no different from that of nonstable sort, it seems that parallel stable sort is inherently more complex than parallel nonstable sort. This is the reason we have divided the bits of the keys unevenly.

In Coarse_Sort we need to (nonstable) sort a sequence of n keys, each key being in the range $[1, n/\log^3 n]$ and, in Fine_Sort we have to (stable) sort n keys in the range $[1, \log^3 n]$. In terms of notation, our algorithm can be summarized as follows.

Let $D = n/\log^3 n$ and $k_i' = \lfloor k_i/D \rfloor$ and $k_i'' = k_i - k_i' * D$ for all $i \in [n]$.

Coarse_Sort. Sort $k_1'', k_2'', \cdots, k_n'' \in [D]$. Let σ be the resultant permutation.

Fine_Sort. Stable-sort $k'_{\sigma(1)}, k'_{\sigma(2)}, \cdots, k'_{\sigma(n)} \in [\log^3 n]$. Let ρ be the resultant permutation.

Output. The permutation $\rho \cdot \sigma$, the composition of ρ and σ.

In §§ 3.2 and 3.3 we describe Fine_Sort and Coarse_Sort, respectively.

3.2. Fine_Sort. We give a deterministic algorithm for Fine_Sort. First we will show how to stable-sort n keys in the range $[\log n]$ using $n/\log n$ processors in time $O(\log n)$ and then apply the idea of radix sorting to prove that we can stable-sort n keys in the range $[(\log n)^{O(1)}]$ within the same resource bounds.

LEMMA 3.1. *n keys $k_1, k_2, \cdots, k_n \in [\log n]$ can be stable-sorted in $O(\log n)$ time using $P = n/\log n$ processors.*

Proof. In Fine_Sort algorithm, each processor π is given $\log n$ successive keys. Each one of the P processors starts by sequentially stable-sorting the keys given to it. Then, collectively, the P processors group all the keys with equal values. (There are $\log n$ groups in all.) Finally, they output a rearrangement of the given sequence in which all the 1's (i.e., keys with a value 1) appear first, all the 2's appear next, and so on. Throughout the algorithm the relative order of equal keys is preserved. More details follow.

To each processor $\pi \in [P]$ we assign the key indices $J(\pi) = \{j \mid (\pi - 1) \log n < j \leqq \min(n, \pi \log n)\}$. There are three steps in the algorithm.

Step 1.

Each processor π sequentially stable-sorts the keys $\{k_j \mid j \in J(\pi)\}$ in time $O(\log n)$ (see Lemma 2.4), and hence constructs $\log n$ lists $J_{\pi,k} = \{j \in J(\pi) \mid k_j = k\}$ for $k \in [\log n]$. Elements in $J_{\pi,k}$ are ordered in the same relative order as in the input.

Step 2.

The P processors collectively perform the prefix sum of

$$\begin{aligned}(&|J_{1,1}|, |J_{2,1}|, \cdots, |J_{P,1}|,\\ &|J_{1,2}|, |J_{2,2}|, \cdots, |J_{P,2}|,\\ &\cdots\\ &|J_{1,q}|, |J_{2,q}|, \cdots, |J_{P,q}|)\end{aligned}$$

where $q = \log n$. Call this sum

$$\begin{aligned}(&S_{1,1}, S_{2,1}, \cdots, S_{P,1},\\ &S_{1,2}, S_{2,2}, \cdots, S_{P,2},\\ &\cdots\\ &S_{1,q}, S_{2,q}, \cdots, S_{P,q}).\end{aligned}$$

Step 3.

Each processor π sequentially computes the position of each one of its keys in the output using the prefix sum. The position of keys in the list $J_{\pi,l}$ will be $S_{\pi-1,l}+1, S_{\pi-1,l}+2, \cdots, S_{\pi,l}$.

Analysis. It is easy to see that Steps 1 and 3 can be performed within the stated resource bounds. Step 2 also can be completed within the stated resource bounds as given in Lemma 2.1. □

LEMMA 3.2. *If n keys in the range $[R]$ (for any $R = n^{O(1)}$) can be stable-sorted in $O(\log n)$ time using $P = n/\log n$ processors, then n keys $k_1, k_2, \cdots, k_n \in [R^2]$ can be stable-sorted in time $O(\log n)$ using the same number of processors.*

Proof. Let $k'_i = \lfloor k_i / R \rfloor$ and $k''_i = k_i - k'_i * R$ for every $i \in [n]$. First, stable-sort $k''_1, k''_2, \cdots, k''_n$ obtaining a permutation σ. Then stable-sort $k'_{\sigma(1)}, k'_{\sigma(2)}, \cdots, k'_{\sigma(n)}$ obtaining a permutation ρ. Output $\rho \cdot \sigma$. Clearly both these sorts can be completed in time $O(\log n)$ using P processors. □

Lemmas 3.1 and 3.2 immediately imply the following lemma.

LEMMA 3.3. *n integer keys in the range $[(\log n)^{O(1)}]$ can be stable-sorted in time $O(\log n)$ using $n/\log n$ processors.*

3.3. Coarse_Sort. In this section we fix a key domain $[D]$, where $D = n/\log^3 n$. We assume, without loss of generality, that $\log^3 n$ divides n. Let the input keys be $k_1, k_2, \cdots, k_n \in [D]$. Define the *index sequence* for each key $k \in [D]$ to be $I(k) = \{i \mid k_i = k\}$. The randomized algorithm for Coarse_Sort to be presented in this section employs $P = n/\log n$ processors and runs in time $\tilde{O}(\log n)$. The sorted sequence is nonstable.

The main idea is to calculate the cardinalities of the index sequences $I(k)$, $k \in [D]$ approximately, and then to use the assignment algorithm of § 2.2 to rearrange the given sequence in sorted order.

LEMMA 3.4. *Given as input $k_1, k_2, \cdots, k_n \in [D]$ we can compute $N(1)$, $N(2), \cdots, N(D)$ in $\tilde{O}(\log n)$ time using $P = n/\log n$ processors such that $\sum_{k \in [D]} N(i) = O(n)$ and furthermore, with very high likelihood $N(k) \geqq |I(k)|$ for each $k \in [D]$.*

Proof. The following sampling algorithm serves as a proof.

Step 1.

Each processor $\pi \in [D \log n]$ in parallel chooses a random index $s_\pi \in [n]$. Let S be the sequence $\{s_1, s_2, \cdots, s_{D \log n}\}$.

Step 2.

The P processors collectively sort the keys with the chosen indices. That is, they sort $k_{s_1}, k_{s_2}, \cdots, k_{s_{D \log n}}$ and compute index sequences $I_S(k) = \{i \in S \mid k_i = k\}$ (for each $k \in [D]$).

Step 3.

D of the P processors in parallel set $N(k) = d(\log^2 n) \max(|I_S(k)|, \log n)$ for $k \in [D]$, d being a constant to be fixed in the analysis.

Output $N(1), N(2), \cdots, N(D)$.

Analysis. Trivially, Steps 1 and 3 can be performed in $O(1)$ time. Step 2 can be performed using any of the optimal GENERAL_SORT algorithms in $O(\log n)$ time (see Lemma 2.5). (Notice that we have to sort only $n/\log^2 n$ keys in step 2.) It remains to be shown that $N(i)$'s computed by the sampling algorithm satisfy the conditions in Lemma 3.4.

If $|I(k)| \leqq d \log^3 n$, then always $N(k) \geqq d \log^3 n \geqq |I(k)|$. So suppose $|I(k)| > d \log^3 n$. Then it is easy to see that $|I_S(k)|$ is a binomial variable with parameters $(n/\log^2 n, |I(k)|/n)$. The Chernoff bounds (see Appendix A, (2)) imply that for all $\alpha \geqq 1$, there exists a c such that

$$\text{Probability}\,(|I_S(k)| \leqq c\alpha |I(k)|/\log^2 n) \leqq \frac{1}{n^\alpha}.$$

Therefore, if we choose $d = (c\alpha)^{-1}$ then $N(k) \leqq |I(k)|$ (for every $k \in [D]$) with probability $\geqq 1 - n^{-\alpha}$. The Chernoff bounds (3) also imply that for all $\alpha \geqq 1$ there exists an h such that $N(k) \geqq (h\alpha)|I(k)|$ (for every $k \in [D]$) with probability $\geqq 1 - n^{-\alpha}$.

The bound on $\sum_{k \in [D]} N(k)$ clearly holds since

$$\sum_{k \in [D]} N(k) \leqq \sum_{k \in [D]} d \log^2 n[|I_S(k)| + \log n] = d \log^3 nD + d \log^2 n \sum_{k \in [D]} |I_S(k)|$$

$$= dn + d \log^2 nD \log n = 2dn.$$

This concludes the proof of Lemma 3.4. □

Having obtained the approximate cardinalities of the index sets, we apply the assignment algorithm of § 2.2. The set Q is the set of key indices viz., $\{1, 2, \cdots, n\}$. An index i belongs to group $G_{i'}$ if the value of the key with index i is i'. Under this definition, group G_j is the same as index sequence $I(j)$, $j = 1, 2, \cdots, D$. Since we can find approximate cardinalities of these groups (Lemma 3.4), we can use the assignment algorithm of § 2.2 to rearrange the given sequence in sorted order. Thus we have the following lemma.

LEMMA 3.5. *n keys $k_1, k_2, \cdots, k_n \in [D]$ can be sorted in time $\tilde{O}(\log n)$ using $n/\log n$ processors.*

Lemmas 3.3 and 3.5 together with the algorithm summary in § 3.1 prove the following theorem.

THEOREM 3.1. INTEGER_SORT *of n keys can be performed in randomized $\tilde{O}(\log n)$ time using $n/\log n$* CRCW PRAM *processors.*

4. Sublogarithmic time algorithms. In the previous section we presented an optimal algorithm for INTEGER_SORT. In this section we will be presenting nonoptimal sublogarithmic time algorithms for (1) prefix sum computation, (2) finding a random permutation of n elements, (3) GENERAL_SORT, and (4) INTEGER_SORT. Algorithms 3 and 4 are direct consequences of algorithms 1 and 2. Our prefix algorithm employs $P \leqq n/\log n$ processors and runs in time $O(\log n/\log\log(P \log n/n))$. Algorithms 2, 3, and 4 run in time $\tilde{O}(\log n/\log\log n)$. GENERAL_SORT uses $n(\log n)^{\varepsilon}$ processors and algorithms 2 and 4 use $n(\log\log n)^2/\log n$ processors.

4.1. A sublogarithmic prefix algorithm. We have a sequence of integers $X(1), X(2), \cdots, X(n)$. We need to find the prefix sum of this sequence. This problem can be solved in sublogarithmic time if we use more than $n/\log n$ processors as is stated by the following lemma.

LEMMA 4.1. *Prefix-sum computation can be performed in time $O(\log n/\log(P \log n/n))$ using $P \leqq n/\log n$* CRCW PRAM *processors.*

Proof. The algorithm can be summarized as follows: (1) divide the given sequence into blocks of d (to be determined later) successive keys; (2) sequentially compute prefix sums in each block; (3) apply prefix to the final prefixes in each block; and (4) compute prefixes in each block by using the result from 3 for the previous block.

More details follow. Let $n_1 = n/d$.

Step 1.

In $O(d)$ time using $n_1 \leqq P$ processors compute $X'(i, m)$, $i \in [n_1]$, $m \in [d]$, where $X'(i, m) = \sum_{j=(i-1)d+1}^{(i-1)d+m} X'(j)$.

Step 2.

Compute the prefix sum of the total sum of each part, i.e., compute $Y'(1), Y'(2), \cdots, Y'(n_1)$, where $Y'(i) = \sum_{j=1}^{i} X(j, d)$, for $i = 1, \cdots, n_1$.

Step 3.

In time $O(d)$, using n_1 processors compute

$$(X'(1,1), X'(1,2), \cdots, X'(1,d),$$
$$Y'(1) \circ X'(2,1), Y'(1) \circ X'(2,2), \cdots, Y'(1) \circ X'(2,d),$$
$$\cdots$$
$$Y'(n_1-1) \circ X'(n_1,1), Y'(n_1-1) \circ X'(n_1,2), \cdots, Y'(n_1-1) \circ X'(n_1,d))$$

which is the required output.

Analysis. Clearly, Steps 1 and 3 can be performed with n_1 processors in time $O(d)$. It remains to show that Step 2 can be performed within the same time using P processors.

Let $C_{n,2}$ be a circuit of size n and in-degree 2 that computes the prefix sum of n elements in depth $O(\log n)$. Obtain an equivalent circuit $C_{n_2,b}$ of size $n_2 = n/b$ ($n_2 \geqq n_1$) and in-degree b in the obvious way (by collapsing subcircuits of height $\log b$ into single nodes starting from the bottom of the circuit [11]). We will simulate $C_{n_2,b}$.

Each input key is a $\log n$ bit integer. Each one of the keys is divided into d parts, each comprising $\log n/d$ successive bits. The simulation proceeds in d stages. In the first stage, we input the $\log n/d$ least significant bits of the keys to the circuit $C_{n_2,b}$. In the second stage, we input the next most $\log n/d$ significant bits of the input keys to the circuit. Similarly we pipeline all the parts of the keys one part per stage. The computation in the circuit proceeds in a pipeline fashion.

At any stage, every node v of $C_{n_2,b}$ has to compute the sum of b integers that arrive at this node from its children and the carry it stored from the previous stage. v also has to store the carry from this stage to be used in the next. Each one of these b integers and the carry can be of at the most $2 \log n/d = s$ bits. Therefore, the computation at v can be made to run in time $O(1)$ if we replace v by a constant depth circuit of size $b2^{s(b+1)}$. The depth of $C_{n_2,b}$ is $\log_b n_2$. Thus, the run time of the circuit (and hence the simulation time) will be $\log_b n_2 + O(d)$. The size of the circuit is $n_2 b 2^{s(b+1)}$.

We require $b2^{s(b+1)} \leqq P/n_2$, $s = 2 \log n/d$, $n_1 = n/d$, $n_1 \leqq n_2$ and $\log_b n_2 = O(d)$. It is easy to see that choosing $s = \log \log (P \log n/n)$ will satisfy all the above constraints. This concludes the proof of Lemma 4.1. □

4.2. A sublogarithmic permutation algorithm. The problem is to compute a random permutation of $(1, 2, \cdots, n)$ in sublogarithmic parallel time. The algorithm presented in this section is very similar to the assignment algorithm of § 2.2. It employs $P = n(\log \log n)^2/\log n$ processors and runs in time $\tilde{O}(\log n/\log \log n)$.

A shared memory of size $2n$ is used. The main idea is to find unique assignments (in the common memory) for each one of the indices $i \in [n]$ and then to eliminate unused cells of common memory using a prefix sum computation. Processors are partitioned into groups of size $(\log \log n)^2$. Each group of $(\log \log n)^2$ processors gets $\log n$ successive indices. Detailed algorithm follows.

Step 1.

The $\log n$ indices given to each group of processors are partitioned into groups of size $(\log \log n)^2$. Step 1 consists of $\log n/(\log \log n)^2$ phases. In the ith phase ($i = 1, 2, \cdots, \log n/(\log \log n)^2$) each processor is given a distinct index from the group i of indices. Each processor spends $d \log \log n$ time (for some constant d) to find an assignment for its index (as explained in Step 2 of § 2.2). After $d \log \log n$ time the ith phase ends.

Step 2.

P processors perform a prefix-sum computation to determine the number (call it N) of indices that do not yet have an assignment. Let $z = \lfloor P/N \rfloor$.

Step 3.

A distinct group of z processors in parallel work to find an assignment for every index j that remains without an assignment. A group succeeds even if a single processor in the group succeeds. Each group is given $C \log n/\log \log n$ time (for some constant C).

After $C \log n/\log\log n$ time, even if a single index remains without an assignment the whole algorithm is aborted and started anew. (Grouping of processors in this step can easily be done using the prefix sum of Step 2.)

Step 4.

Finally, P processors perform a prefix-sum computation to eliminate unused cells and read the positions of their indices in the output.

Analysis. Consider the ith phase of Step 1. The probability that a given processor π succeeds in finding an assignment for its index in a single trial is $\geqq \frac{1}{2}$. Let Y be a random variable equal to the number of processors failing in the jth trial of phase i. Then Y is upperbounded by a binomial random variable with parameters $(N_i^j, \frac{1}{2})$ (where N_i^j is the number of processors that have not succeeded until the beginning of the jth trial of phase i). (Note that $N_i^1 = P$.) The Chernoff bounds (3) imply that Y is at the most a constant (<1) fraction of N_i^j with probability $\geqq 1 - 2^{-\varepsilon N_i^j}$ (for some fixed $\varepsilon < 1$). Therefore the number of unsuccessful processors at the end of phase i is $\tilde{O}(P/\log n)$. The number of keys without assignments at the end of Step 1 is $\sum_{i=1}^{\log n/(\log\log n)^2} N_i^{d \log\log n}$. Using additive property of binomial distributions and the Chernoff bounds we conclude that the number of keys without assignments at the end of Step 1 is $O(n/\log n)$ (and hence $z = \Omega((\log\log n)^2)$) with probability $\geqq 1 - n^{-\beta}$ for any $\beta \geqq 1$.

Step 2 runs in time $O(\log n/\log\log n)$ (Lemma 2.2). In Step 3, probability that a particular group fails in one trial is $\leqq (\frac{1}{2})^{\Omega((\log\log n)^2)}$. This implies that the probability that there is at least one unsuccessful group at the end of Step 3 can be made $\leqq n^{-\alpha}$, for any $\alpha \geqq 1$, if we choose a proper C.

Thus we conclude that the whole algorithm will run successfully in time $\tilde{O}(\log n/\log\log n)$. Clearly, this algorithm can also be used to solve the assignment problem of § 2.2. Thus we have the following lemma.

LEMMA 4.2. *The problem of computing a random permutation of n elements (and hence the assignment problem of* § 2.2*) can be solved in time* $\tilde{O}(\log n/\log\log n)$ *using* $P = n(\log\log n)^2/\log n$ *processors.*

4.3. An optimal sub-logarithmic GENERAL_SORT algorithm. Given as input $k_1, k_2, \cdots, k_n$, Reischuk's algorithm [25] for GENERAL_SORT samples $\sqrt{n}$ keys at random. If $l_1, l_2, \cdots, l_{\sqrt{n}}$ are the sampled keys in sorted order, these keys divide the input keys into $p \leqq \sqrt{n}+1$ collections $S_1, S_2, \cdots, S_p$, where $S_1 = \{q \mid q \leqq l_1\}$, $S_i = \{q \mid l_{i-1} < q \leqq l_i\}$ for $i = 2, 3, \cdots, (p, -1)$, and $S_p = \{q \mid q > k_{p-1}\}$. With very high likelihood [25], each one of these collections will be of size $O(\sqrt{n}\log n)$. (Reif and Valiant [24] give an algorithm for sampling $\sqrt{n}$ keys that will ensure that each one of these collections will be of size $O(\sqrt{n})$.) Having identified these collections, his algorithm sorts each one of them recursively and merges the results trivially.

As such, the algorithm in [25] requires a computer of word length $\Omega(\sqrt{n}\log n)$. This problem can be circumvented using the assignment algorithm of § 2.2. Moreover, such a modified algorithm can be made sublogarithmic if $n(\log n)^{\varepsilon}$ processors are used. A detailed algorithm follows.

Procedure sublog$GS(\{k_1, k_2, \cdots, k_n\})$;

Step 1. If n is a constant sort trivially.

Step 2. $\sqrt{n}$ processors in parallel each sample a random key.

Step 3. Sort the $\sqrt{n}$ keys sampled in Step 2 by comparing every pair of keys and computing the rank of each key. This can be done in $O(\log n/\log$

$\log n$) time using n processors. Let the sorted sequence be $l_1, l_2, \cdots, l_{\sqrt{n}}$.

Step 4. Processors are partitioned into groups of size $(\log n)^\epsilon$. Each group gets an index $i \in [n]$. In parallel each group does a $(\log n)^\epsilon$-ary search on $l_1, l_2, \cdots, l_{\sqrt{n}}$ to find out the collection $S_{i'}$ to which k_i belongs.

Step 5. n processors collectively compute $N(1), N(2), \cdots, N(p)$ such that $\sum_{j=1}^{p} N(j) = O(n)$ and $N(j) \geqq |S_j|$ for every $j \in [p]$. (Recall that $p \leqq \sqrt{n}+1$).

Step 6. n processors use the sublogarithmic assignment algorithm of § 4.2 to rearrange $k_1, k_2, \cdots, k_n$ such that all the elements of S_1 will appear first, all the elements of S_2 will appear next, and so on.

Step 7. Recursively sort $S_1, S_2, \cdots, S_p$. Here $O(\sqrt{n}(\log n)^\epsilon)$ processors work on each subproblem. Finally output sublog$GS(S_1), \cdots,$ sublog$GS(S_p)$.

Analysis. If $T'(n)$ is the time sublogGS takes to sort n general keys, Step 1 and Step 2 take $O(1)$ time each. Step 3, Step 4, and Step 6 take $\tilde{O}(\log n/\log\log n)$ time each. Step 7 takes time $T'(c\sqrt{n})$ (for some constant c) with probability $\geqq 1-n^{-\alpha}$ (for any $\alpha \geqq 1$). This is because no collection will be of size more than $O(\sqrt{n})$ with the same probability (if we employ Reif and Valiant's [24] sampling algorithm). Computing $N(1), N(2), \cdots, N(p)$ (Step 5) can be done in time $\tilde{O}(\log n/\log\log n)$ using n processors that employ a sampling algorithm very similar to the one given in § 3.2. (For details see Appendix B.) Therefore, the recurrence relation for $\bar{T}'(n)$, the expected value of $T'(n)$ can be written as

$$\bar{T}'(n) \leqq \bar{T}'(c\sqrt{n}) + \tilde{O}(\log n/\log\log n) + \tilde{O}(n^{-\alpha})\,\bar{T}'(n-\sqrt{n}+1).$$

By induction we can show that $\bar{T}'(n) \leqq \tilde{O}(\log n/\log\log n)$. Thus we have the following theorem.

THEOREM 4.1. GENERAL_SORT *can be done in time* $\tilde{O}(\log n/\log\log n)$ *with* $n(\log n)^\epsilon$ CRCW PRAM *processors.*

4.4. A sublogarithmic algorithm for INTEGER_SORT. In § 3, we presented an INTEGER_SORT algorithm that used $n/\log n$ processors to sort n integer keys in time $\tilde{O}(\log n)$. The same algorithm can be used to sort in time $\tilde{O}(\log n/\log\log n)$ if the number of processors used is $P = n(\log\log n)^2/\log n$. Here we will indicate only the modifications that need to be made.

The P processors are partitioned into groups of size $(\log\log n)^2$ and each group is given $\log n$ successive indices. In Fine_Sort Step 1, each group of $(\log\log n)^2$ processors stable sorts the $\log n$ keys given to it using any of the parallel optimal stable GENERAL_SORT algorithms, in time $O(\log n/\log\log n)$. Step 2 runs in time $O(\log n/\log\log n)$. In Step 3, each group of processors computes the position of each one of its $\log n$ keys in the output using the prefix sum of Step 2. The time needed for Step 3 is $\log n/(\log\log n)^2$.

In Coarse_Sort, while computing the $N(i)$'s, Steps 1 and 3 run in time $O(1)$. In Step 2, we need to sort $n/\log^2 n$ keys. The sublogarithmic algorithm of § 4.2 for GENERAL_SORT can be used to run Step 2 in time $\tilde{O}(\log n/\log\log n)$ using $< n/\log n$ processors. After computing $N(i)$'s, rearranging of the keys can be done using P processors in time $\tilde{O}(\log n/\log\log n)$ (Lemma 4.2). Therefore, both Coarse_Sort and Fine_Sort run in time $\tilde{O}(\log n/\log\log n)$. Thus we have the following theorem.

THEOREM 4.2. INTEGER_SORT *can be performed in* $\tilde{O}(\log n/\log\log n)$ *time using* $P = n(\log\log n)^2/\log n$ CRCW PRAM *processors.*

5. Conclusions. All the sorting algorithms appearing in this paper are nonstable. It remains an open problem to obtain stable versions of these algorithms. If we have a stable algorithm for INTEGER_SORT then the definition of integer keys can be extended to include integers in the range $[n^{O(1)}]$. Any deterministic algorithm for INTEGER_SORT using a polynomial number of CRCW PRAM processors will take at least $\Omega(\log n/\log\log n)$ time, as has been shown by Beam and Hastad [6]. However it is an open question whether there exists a randomized CRCW PRAM algorithm that uses a polynomial number of processors and runs in time $o(\log n/\log\log n)$.

A recent result of Alon and Azar [2] implies that our sublogarithmic time GENERAL_SORT algorithm is optimal. Their lower bound result is for a more powerful comparison tree model of Valiant and hence readily holds for PRAMs as well. Alon and Azar's theorem is that if P is the number of processors used, then the average time, T, required for sorting n elements by any randomized algorithm is $\Theta(\log n/\log(1+P/n))$ for $P \geqq n$, and the average time is $\Theta(\log n/(P/n))$ for $P \leqq n$. In particular, if $P = n(\log n)^{\epsilon}$, then $T = \Theta(\log n/\log\log n)$. It remains an open problem to prove or disprove the optimality of our sublogarithmic INTEGER_SORT algorithm.

Appendix A. Probabilistic bounds. We say a random variable X *upper bounds* another random variable Y (equivalently, Y *lower bounds* X) if for all x such that $0 \leqq x \leqq 1$, Probability $(X \leqq x) \leqq$ Probability $(Y \leqq x)$.

A *Bernoulli trial* is an experiment with two possible outcomes viz., *success* and *failure*. The probability of success is p.

A *binomial variable* X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p.

The *distribution function* of X can easily be seen to be

$$\text{Probability}(X \leqq x) = \sum_{k=0}^{x} \binom{n}{k} p^n (1-p)^{n-k}.$$

Chernoff [7] and Angluin and Valiant [3] have found ways of approximating the tail ends of a binomial distribution. In particular, they have shown the following.

LEMMA A.1. *If X is binomial with parameters (n, p), and $m > np$ is an integer, then*

$$\text{Probability}(X \geqq m) \leqq \left(\frac{np}{m}\right)^m e^{m-np}. \tag{1}$$

Also,

$$\text{Probability}(X \leqq \lfloor (1-\varepsilon)pn \rfloor) \leqq \exp(-\varepsilon^2 np/2) \tag{2}$$

and

$$\text{Probability}(X \geqq \lceil (1+\varepsilon)np \rceil) \leqq \exp(-\varepsilon^2 np/3) \tag{3}$$

for all $0 < \varepsilon < 1$.

Appendix B. A sampling algorithm. We have an index set $Q = \{1, 2, \cdots, n\}$. Each index belongs to exactly one of $\sqrt{n}$ groups $G_1, G_2, \cdots, G_{\sqrt{n}}$. For any index i, in constant time we can find out the group $G_{i'}$ that i belongs to.

Problem. Compute $N(1), N(2), \cdots, N(\sqrt{n})$ such that $\sum_{i=1}^{\sqrt{n}} N(i) = O(n)$ and $N(i) \geqq |G_i|$ for each $i \in [\sqrt{n}]$, given that each $|G_i| \leqq \sqrt{n} \log n$.

LEMMA B.1. *The above problem can be solved in time $\tilde{O}(\log n/\log\log n)$ using n processors.*

Proof. We provide a sampling algorithm. A shared memory of size n is used. This shared memory is divided into $\sqrt{n}$ blocks $B_1, B_2, \cdots, B_{\sqrt{n}}$ each of size $\sqrt{n}$.

Step 1.

$n/\log n$ processors in parallel each choose a random index (in $[n]$).

Step 2.

Every processor $\pi \in [n/\log n]$ has to find an assignment for its index i in the block $B_{i'}$. It chooses a random cell in $B_{i'}$ and tries to write in it. If it succeeds, it increments the contents of that cell by 1. If it does not succeed in the first trial, it tries a second time to increment the **same** cell. It tries as many times as it takes.

A total of $h \log n/\log\log n$ (for some h to be determined) time is given.

Step 3.

n processors perform a prefix-sum computation on the contents of the shared memory and hence compute $L(1), L(2), \cdots, L(\sqrt{n})$ where $L(i)$ is the sum of the contents of block B_i, $i \in [\sqrt{n}]$.

Step 4.

$\sqrt{n}$ processors set in parallel $N(i) = d(\log n) \max(1, L(i))$ and output $N(i)$, $i \in [\sqrt{n}]$. d is a constant to be determined.

Analysis. Let $M(i)$, $i \in [\sqrt{n}]$ stand for the number of indices chosen in Step 1 that belong to G_i and let $R(i) = d(\log n) \max(1, M(i))$. Following the proof of Lemma 3.4, the $R(i)$'s satisfy the conditions $\sum_{i=1}^{\sqrt{n}} R(i) = O(n)$ and $R(i) \geqq |G_i|$, $i \in [\sqrt{n}]$. The proof will be complete if we can show that $L(i) = M(i)$ with very high probability.

Showing that $L(i) = M(i)$, $i \in [\sqrt{n}]$ is the same as showing that no cell in the common memory will be chosen by more than $h \log n/\log\log n$ processors in Step 1. Let Y be a random variable equal to the number of processors that have chosen a particular cell q. Following the proof of Lemma 3.4, no $M(i)$ will be greater than $c\beta\sqrt{n}$ with probability $\geqq 1 - n^{-\beta}$ for any $\beta \geqq 1$ and some fixed c. Therefore, Y is upperbounded by a binomial variable with parameters $(c\beta\sqrt{n}, 1/\sqrt{n})$. The Chernoff bounds (1) imply that $Y \leqq h \log n/\log\log n$ with probability $\geqq 1 - n^{-\alpha}$, for any $\alpha \geqq 1$ and a proper h. □

Acknowledgments. The authors thank Yijie Han, Sandeep Sen, and the referees for their insightful comments.

REFERENCES

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

[2] N. ALON AND Y. AZAR, *The average complexity of deterministic and randomized parallel comparison sorting algorithms*, Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 489–498.

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

[4] M. ATAI, J. KOMLÓS, AND E. SZEMERÉDI, *An $O(n \log n)$ sorting network*, Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 1–9.

[5] K. BATCHER, *Sorting networks and their applications*, Spring Joint Computer Conference 32, AFIPS Press, Montvale, NJ, 1968, pp. 307–314.

[6] P. BEAM AND J. HASTAD, *Optimal bounds for decision problems on the* CRCW PRAM, Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 83–93.

[7] H. CHERNOFF, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493-507.

[8] R. COLE, *Parallel merge sort*, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 511-516.

[9] R. COLE AND U. VISHKIN, *Approximate and exact parallel scheduling with applications to list, tree, and graph problems*, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 478-491.

[10] W. FELLER, *An introduction to probability theory and its applications*, Vol. 1, John Wiley, New York, 1950.

[11] F. E. FICH, *Two problems in concrete complexity cycle detection and parallel prefix computation*, Ph.D. thesis, Univ. of California, Berkeley, CA, 1982.

[12] W. HOEFFDING, *On the distribution of the number of successes in independent trials*, Ann. Math. Statistics, 27 (1956), pp. 713-721.

[13] J. E. HOPCROFT AND R. E. TARJAN, *Efficient algorithms for graph manipulation*, Comm. ACM, 16 (1973), pp. 372-378.

[14] N. J. JOHNSON AND S. KATZ, *Discrete Distributions*, Houghton-Miffin, Boston, MA, 1969.

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

[16] L. KUČERA, *Parallel computation and conflicts in memory access*, Inform. Process. Lett. 14 (1982), pp. 93-96.

[17] T. LEIGHTON, *Tight bounds on the complexity of parallel sorting*, Proc. 16th Annual ACM Symposium on Theory of Computing, Washington, DC, 1984, pp. 71-80.

[18] R. E. LADNER AND M. J. FISCHER, *Parallel Prefix Computation*, J. Assoc. Comput. Mach., 27(4) (1980), pp. 831-838.

[19] G. L. MILLER AND J. H. REIF, *Parallel tree contraction and its application*, Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 478-489.

[20] J. H. REIF, *Symmetric complementation*, J. Assoc. Comput. Mach., 31 (1984a), pp. 401-421.

[21] ———, *On the power of probabilistic choice in synchronous parallel computations*, SIAM J. Comput., 13 (1984), pp. 46-56.

[22] ———, *An optimal parallel algorithm for integer sorting*, Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 496-503.

[23] J. H. REIF AND J. D. TYGAR, *Efficient parallel pseudo-random number generation*, CRYPTO '85, Santa Barbara, CA, August 1985.

[24] J. H. REIF AND L. G. VALIANT, *A logarithmic time sort for linear size networks*, Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 10-16; J. Assoc. Comput. Mach., 34 (1987), pp. 60-76.

[25] R. REISCHUK, *A fast probabilistic sorting algorithm*, Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science, 1981, pp. 88-102.

[26] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging, and sorting in a parallel computation model*, J. Algorithms 2 (1981), pp. 212-219.

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

[28] U. VISHKIN, *Randomized speed-ups in parallel computation*, Proc. 16th Annual Symposium on Theory of Computing, 1984, pp. 230-239.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 608-624, June 1989

013

PRECISE ANALYSES OF THE RIGHT- AND LEFT-SHIFT GREATEST COMMON DIVISOR ALGORITHMS FOR $GF(q)[x]$*

G. H. NORTON†

Abstract. The precise average and worst-case running times of the right- and left-shift gcd algorithms for $GF(q)[x]$ are derived. A new approximate integer model for the binary greatest common divisor algorithm is obtained. The right-shift polynomial worst case differs markedly for $q=2$ and $q>2$. The method also yields an easy analysis of the Euclidean algorithm.

Key words. polynomial, finite field, greatest common divisor

AMS(MOS) subject classifications. primary 68Q25, 68Q40; secondary 11A05, 12E99

Introduction. Let $q \geqq 2$ be a prime power. This paper derives the precise running times of the right-shift greatest common divisor (gcd) algorithm for $GF(q)[x]$ of [N] and the left-shift gcd algorithm of [BK] (by way of comparison). The method used to obtain the averages for uniformly distributed polynomials is similar to the lattice-point analysis of the binary gcd algorithm [K, p. 330 ff.], except that exact "transition probabilities" are derived. The first part of this paper (§§ 1-4) treats the right-shift gcd algorithm. Sections 5, 6, and 7 treat the left-shift algorithm. The case $q=2$ suggests a new approximate model for the binary gcd algorithm for integers. The worst-case polynomial running times, however, are quite different from the integer case, and were obtained by first computing upper bounds and then the polynomials which realise the upper bounds. Three families were found: one for $q=2$ and one for $q>2$ in part one, whereas one family suffices for the second part.

A polynomial is said to be (right) *normalized* if it does not vanish at zero. (This is no restriction in computing gcd's, since the number of common powers of x may be accumulated beforehand.) The right-shift gcd algorithm of [N] for normalized polynomials will be called "Algorithm B2" and the left-shift algorithm of [BK] will be called "Algorithm A2." Both algorithms perform a number of steps or iterations of the following form: each *iteration* has a pair (u, v) of polynomials as input, with $u \neq 0$ and replaces (u, v) by another pair (u', v') such that $\gcd(u', v') = \gcd(u, v)$ and either (i) $\deg u' < \deg u$ (continuation) or (ii) $u' = 0$ (termination). See Algorithms 1.2 and 5.1 for precise details. It is not hard to see that Algorithm A2 and Algorithm B2 compute the gcd of (normalized for Algorithm B2) polynomials of degree m, n over a field K in at most $m+n+1$ iterations. This paper improves this worst-case result for Algorithm B2 when $K = GF(2)$; it also determines coefficients α_q and β_q explicitly so that the average number of iterations is $\alpha_q m + \beta_q n + 0(1)$ when $K = GF(q)$.

The principal results for Algorithm B2 are the following theorems.

THEOREM A. *For uniformly distributed normalized $GF(q)$ polynomials of degree m, n, the average number of iterations of Algorithm* B2 *is*

$$\frac{q-1}{q} m + \frac{q-1}{q+1} n + 0(1) \quad \text{if } m \geqq n \geqq 0.$$

THEOREM B. *Let $N \geqq 1$. For uniformly and independently distributed $GF(q)$ polynomials with degree in the range* 0 *to* $N-1$, *the average number of iterations of*

* Received by the editors August 18, 1987; accepted for publication (in revised form) February 26, 1988.
† Department of Electrical Engineering, University of Bristol, Bristol, United Kingdom. Present address, Laboratoire d'Analyse Numérique, Mathématiques, Université Paul Sabatier, 31062 Toulouse, France.

Algorithm B2 *is*

$$\frac{2q+1}{q+1}\,\frac{q-1}{q}\,N+0(1).$$

THEOREM C. *Let* $m \geqq n \geqq 0$. *For* $u, v \in GF(q)[x]$ *with degree* m, n *respectively, the maximum number of iterations of Algorithm* B2 *is* $m+\lfloor n/2 \rfloor + 1$ *if* $q=2$ *and* $m+n+1$ *if* $q>2$. *In both cases, the maxima are attained.*

The proof of Theorem A proceeds by first obtaining the recurrence relations satisfied by the averages and then by solving these relations. This method readily yields the probability that two normalized polynomials are coprime and the general solution will also be used in a future paper to analyse a related gcd algorithm.

The approximate integer model suggested by Theorem A is discussed after Theorem 2, § 3.

The corresponding principal results for Algorithm A2 are the following theorems.

THEOREM D. *Let* $m \geqq n \geqq 0$. *For uniformly distributed* $GF(q)$ *polynomials of degree* m, n, *the average number of iterations of Algorithm* A2 *is* $(m+n+1)(1-1/q)+1/q^{n+1}$.

THEOREM E. *Let* $N \geqq 1$. *For uniformly distributed* $GF(q)$ *polynomials with degree in the range* 0 *to* $N-1$, *the average number of iterations of Algorithm* A2 *is* $(2(q-1)/q+\varepsilon)N+0(1)$, *where* ε *tends to zero with* q^{-N}.

THEOREM F. *Let* $m \geqq n \geqq 0$. *For* $u, v \in GF(q)[x]$ *performs with degree* m, n, *the maximum number of iterations of Algorithm* A2 *is* $m+n+1$ *and this maximum is attained.*

In conclusion, it can be seen that Algorithm B2 always performs faster (on the average) than Algorithm A2 and that the most improvement occurs over $GF(2)$.

The averages for A2 also satisfy a recurrence relation, and Theorem D was obtained by solving it. Indeed, the average analysis of the classical Euclidean algorithm (see [K, p. 417, #4]) may also be found by calculating certain transition probabilities and then by solving the associated recurrence relation; the reader may verify that this recurrence has the same characteristic equation as Algorithm A2.

The results of this paper were first announced at AAECC-4 in Karlsruhe and in [No].

Conventions. K denotes a field, not necessarily finite, and $q \geqq 2$ denotes a prime power. The zero polynomial has degree -1, to simplify the transition probabilities. $K[x]$ denotes the ring of polynomials with coefficients in K, that is, finite power series

$$\sum_{i=0}^{m} f_i x^i \text{ with } f_i \in K.$$

1. The right-shift algorithm. This section states the right-shift gcd algorithm for $GF(q)[x]$ of [N] for normalized polynomials, shows that uniformity is preserved, and determines the transition probabilities, an essential step in the precise average analysis.

DEFINITION 1. For non-zero g in $K[x]$

(a) let $L(g)$ be the leading coefficient of g

(b) define the symbol $g/\!/x$ to be g/x^k if for some $k \geqq 0$, x^k divides g but x^{k+1} does not

(c) g_o denotes the constant term of g.

ALGORITHM 2. GCD by Right shifting.

Input. Normalized $f, g \in K[x]$.

Output. gcd (f, g).

procedure B2 $\left(\sum_{i=0}^{m} f_i x^i, \sum_{j=0}^{n} g_j x^j\right)$;

while true do begin

if $\deg(f) < \deg(g)$ then swap (f, g);
$f := f - f_o g_o^{-1} g$;
if $f = 0$ then exit;
$f := f /\!/ x$;
end;
return $L(g)^{-1} g$.

Note that after one iteration of the algorithm, (i) g is normalized and (ii) either $f = 0$ or f is normalized.

EXAMPLE 3. Over GF(2), the iterations are

$$(x^5+x^2+1, x^4+x+1) = (x^4+x^3+x+1, x^4+x+1)$$
$$= (1, x^4+x+1) = (x^3+1, 1) = (1, 1) = (0, 1)$$

and the gcd 1 is returned.

An example using $GF(13)$ coefficients is given in [N, Ex. 2]. It is clear that the computation may be stopped if $\deg(f) = 0$, returning a gcd of 1 in this case.

We now show that Algorithm B2 preserves uniformity.

PROPOSITION 4. *Let $i \geqq 1$, let u, v be uniformly distributed normalized $GF(q)$ polynomials of degree $m \geqq n \geqq 0$ and let $1 \leqq k \leqq m+1$, $0 \leqq l \leqq n$. Suppose that after i iterations (u, v) is replaced by (u', v') of degree $m-k$, $n-l$, respectively. Then u', v' are uniformly distributed in degree $m-k$, $n-l$.*

Proof. It suffices to prove this with $i = 1$, so that $v' = v$, and $\deg u' = m-k$. Then $u = u(0)v(0)^{-1}v(x) + x^k u'(x)$ and so for given polynomials u', v there is a unique u, v with $u(0) = 1$ yielding u', v in one step. Since u, v are uniformly distributed so are u', v as required.

DEFINITION 5. For $m \geqq n \geqq 0$ and $1 \leqq k \leqq m+1$, let $P[(m, n) \to (m-k, n)]$ denote the probability that one iteration of Algorithm B2 replaces uniformly distributed, normalized polynomials of degree m, n by normalized polynomials of degree $m-k$, n, or terminates (if $k = m+1$).

THEOREM 6. (a) *For $m > n \geqq 0$*

$$P[(m, n) \to (m-k, n)] = \begin{cases} (q-1)q^{-k} & \text{if } 1 \leqq k \leqq m-1, \\ q^{1-m} & \text{if } k = m. \end{cases}$$

(b) *For $n \geqq 0$,*

$$P[(n, n) \to (n-k, n)] = \begin{cases} \dfrac{k(q-1)-1}{q^k} & \text{if } 1 \leqq k \leqq n-1,\ n \geqq 2, \\ \dfrac{n(q-1)-1}{(q-1)q^{n-1}} & \text{if } k = n,\ n \geqq 1, \\ (q-1)^{-1}q^{1-n} & \text{if } k = n+1,\ n \geqq 1, \\ 1 & \text{if } k = n+1,\ n = 0. \end{cases}$$

Proof. (a) Suppose that $m > n$ and let u, v be normalized polynomials of degree m, n, respectively. We begin with the case $k = m$. For the algorithm to produce an output of degree zero, there are $m-1$ constraints ($u_i = 0$ for $n+1 \leqq i \leqq m-1$ and $v_o u_i - u_o v_i = 0$ if $n > 0$ and $1 \leqq i \leqq n$) each of which occurs with probability $1/q$. To obtain an output of degree $m-k$ for $1 \leqq k \leqq m-1$, the argument is similar: there will be $k-1$ equalities and one inequality yielding $q^{1-k} \cdot (q-1)q^{-1} = (q-1)q^{-k}$. This proves part (a).

(b) The case $n=0$ is trivial, so suppose now that u and v are normalized polynomials of degree $n>0$. Then $v=\lambda(u+w)$, where $\lambda \in GF(q)$ is non-zero, $-1 \leqq \deg(w) \leqq n-1$ and $w_o \neq -u_o$. The first iteration produces $w_o u - u_o w = w_o u_n x^n + \sum_{i=0}^{n-1} (w_o u_i - u_o w_i) x^i$ (up to a scalar) and $P[w_o = 0] = 1/(q-1)$, $P[w_o u_i - u_o w_i = 0] = 1/q$ if $n \geqq 2$ and $1 \leqq i \leqq n-1$. The cases $k = n+1$, $k = n$, and $1 \leqq k \leqq n-1$ (if $n \geqq 2$) correspond to no, one or at least two non-zero coefficients of $w_o u - u_o w$, respectively, giving the stated probabilities.

2. A general recurrence relation. The method used in [K, p. 330 ff.] to analyse the binary gcd algorithm is adapted to Algorithm B2 for $GF(q)[x]$ in this section. Using the *exact* transition probabilities of Theorem 6, § 1, we define a set of double recurrence relations $A_{mn}(a, c, A_{00})$, from which a single recurrence relation (Theorem 8) is derived. This is then solved using generating functions and partial fractions, and leads to the general solution (Theorem 13). Two applications of this solution are given in the next section, and it will be shown in a future paper that $A_{mn}(q/(q-1), 0, 1)$ yields the uniform average for the algorithm discussed in [K, p. 618, #6].

DEFINITION 1. Let a, c be constants and let $A_{mn} = A_{mn}(a, c, A_{00})$ be given by

$$A_{mn}(a, c, A_{00}) = \begin{cases} A_{00} & \text{if } m = n = 0, \\ a + \sum_{k=1}^{m-1} \dfrac{k(q-1)-1}{q^k} A_{m-k,m} + \dfrac{m(q-1)-1}{(q-1)q^{m-1}} A_{0m} & \text{if } m = n \geqq 1, \\ c + (q-1) \sum_{k=1}^{m-1} \dfrac{A_{m-k,n}}{q^k} + \dfrac{A_{0n}}{q^{m-1}} & \text{if } m > n \geqq 0, \\ A_{nm} & \text{if } n > m \geqq 0. \end{cases}$$

The calculation of A_{mn} for $n=0$ is straightforward.

PROPOSITION 2.

(a) $A_{m+k,n} = \dfrac{q-1}{q} ck + A_{mn}$ if $m > n \geqq 0$, $k \geqq 0$

(b) $A_{m0} = \dfrac{q-1}{q} mc + \dfrac{c}{q} + A_{00}$ if $m > 0$.

Proof.

$$\begin{aligned} \text{(a)} \quad A_{m+1,n} &= c + \frac{q-1}{q} A_{mn} + \sum_{k=2}^{m} \frac{q-1}{q^k} A_{m+1-k,n} + \frac{1}{q^m} A_{0n}, \\ &= c + \frac{q-1}{q} A_{mn} + \sum_{k=1}^{m-1} \frac{q-1}{q^{k+1}} A_{m-k,n} + \frac{1}{q^m} A_{0n}, \\ &= c + \frac{q-1}{q} A_{mn} + \frac{1}{q}(A_{mn} - c) = \frac{q-1}{q} c + A_{mn}, \end{aligned}$$

and part (a) follows from a trivial induction.

(b) By part (a),

$$\begin{aligned} A_{mo} &= \frac{q-1}{q} c(m-1) + A_{10}, \\ &= \frac{q-1}{1} c(m-1) + c + A_{00} \end{aligned}$$

if $m > 0$.

For simplicity let $A_m = A_{mm}$ for $m \geqq 0$. Our next step is to find the recurrence satisfied by A_m. A key role is played by the terms S_m in Definition 3 below; S_m is used to express $A_{m+1,m}$ in terms of A_m (Lemma 6) and the recurrence of Theorem 8 below is derived from a recurrence satisfied by S_m (Lemma 7).

DEFINITION 3. For $m \geqq 2$, let

$$S_m = \sum_{k=1}^{m-1} \frac{A_{m+1,m-k}}{q^{k+1}} + \frac{A_{m+1,0}}{q^m}.$$

LEMMA 4. *For* $m \geqq 2$,

$$A_{m+1,m} = \frac{q-1}{q} A_m + (q-1)S_m - \frac{q-2}{q^m} A_{00} - \frac{(q-2)(q-1)}{q^{m+1}} mc + \left(1 - \frac{1}{q} + \frac{1}{q^2} - \frac{q-2}{q^m}\right) c.$$

Proof.

$$\begin{aligned} A_{m+1,m} &= c + \sum_{k=1}^{m} \frac{q-1}{q^k} A_{m+1-k,m} + \frac{A_{0m}}{q^m}, \\ &= c + \sum_{k=1}^{m+1} \frac{q-1}{q^k} A_{m+1-k,m} - \frac{q-1}{q^{m+1}} A_{0m} + \frac{A_{0m}}{q^m}, \\ &= c + \frac{q-1}{q} A_m + \sum_{k=1}^{m} \frac{q-1}{q^{k+1}} A_{m-k,m} + \frac{A_{0m}}{q^{m+1}}, \\ &= c + \frac{q-1}{q} A_m + \sum_{k=1}^{m} \frac{q-1}{q^{k+1}} \left(A_{m-k,m+1} - \frac{q-1}{q} c\right) + \frac{A_{0m}}{q^{m+1}} \end{aligned}$$

by Proposition 2(a). Now

$$\sum_{k=1}^{m} \frac{A_{m+1,m-k}}{q^{k+1}} = S_m - \frac{q-1}{q^{m+1}} A_{m+1,0}$$

so that

$$A_{m+1,m} = c + \frac{q-1}{q} A_m + (q-1)S_m - (q-1)^2 \frac{A_{m+1,0}}{q^{m+1}} - (q-1)^2 c \sum_{k=1}^{m} \frac{1}{q^{k+2}} + \frac{A_{0m}}{q^{m+1}}.$$

Expanding $A_{m+1,0}$ and A_{m0} via Proposition 2(b) and collecting terms gives the stated formula.

LEMMA 5. *For* $m \geqq 2$,

$$A_{m+1} = \frac{q-1}{q} a + \frac{A_m}{q} + (q-1)S_m + (q-2)\frac{A_{m+1,m}}{q} - \frac{A_{m0}}{q^m} + \left[\frac{q-1}{q^2} - \frac{(q-1)^2+2}{q^{m+1}} + \frac{1}{q^m}\right] c.$$

Proof.

$$A_{m+1} = a + \frac{q-2}{q} A_{m,m+1} + \sum_{k=2}^{m} \frac{q-2+(k-1)(q-1)}{q^k} A_{m+1-k,m+1} + \left(A_{0m} + c\frac{q-1}{q}\right) q^{-m}\left(m + \frac{q-2}{q-1}\right)$$

and by Proposition 2(a), the summation is, after re-indexing,

$$\frac{1}{q}\sum_{k=1}^{m-1} \frac{q-2+(k-1)(q-1)}{q^k}\left(A_{m,m-k} + c\frac{q-1}{q}\right) + \frac{1}{q}\sum_{k=1}^{m-1} \frac{q-1}{q^k}\left(A_{m,m-k} + c\frac{q-1}{q}\right).$$

The first term contributes

$$\frac{1}{q}\left(A_m - a - \frac{A_{0m}}{q^{m-1}}\left((m-1) + \frac{q-2}{q-1}\right)\right)$$

and the second contributes

$$(q-1)\left(S_m - \frac{A_{m+1,0}}{q^m}\right).$$

Applying Proposition 2(a) to $A_{m+1,0}$ and collecting terms completes the proof.

LEMMA 6. *For* $m \geqq 2$,

(a) $$\frac{2(q-1)^2}{q} S_m = A_{m+1} - \frac{A_m}{q^2}(q^2-2q+2) - \frac{q-1}{q} a + \frac{2(q-1)(q-2)}{q^{m+1}} A_{00} + \left\{\frac{2(q-2)(q-1)^2}{q^{m+2}} - 1 + \frac{2}{q}\left(1 - \frac{1}{q} + \frac{1}{q^2}\right) + \frac{1}{q^{m+1}}(2q^2-6q+5)\right\} c$$

(b) $$A_{m+1,m} = \frac{qA_{m+1}}{2(q-1)} + \frac{q-2}{2(q-1)} A_m - \frac{a}{2} + \frac{q}{q-1}\left(\frac{1}{2} - \frac{1}{q} + \frac{1}{q^2} + \frac{1}{2q^{m+1}}\right) c.$$

Proof. Part (a) follows by substituting the value for $A_{m+1,m}$ obtained in Lemma 4 in the expression for A_{m+1} of Lemma 5, and part (b) follows by using part (a) to eliminate S_m from Lemma 4.

LEMMA 7. *For* $m \geqq 3$,

$$S_m = \frac{A_{m,m-1}}{q^2} + \frac{1}{q} S_{m-1} + \left(\frac{1}{q^2} + \frac{1}{q^m} - \frac{2}{q^{m+1}}\right) c.$$

Proof.

$$\begin{aligned} S_m &= \sum_{k=1}^{m-1} \frac{A_{m+1,m-k}}{q^{k+1}} + \frac{A_{m+1,0}}{q^m} \\ &= \frac{1}{q}\sum_{k=1}^{m-1}\left(A_{m,m-k} + \frac{q-1}{q} c\right) q^{-k} + \left(A_{m0} + \frac{q-1}{q} c\right) q^{-m} \\ &= \frac{A_{m,m-1}}{q^2} + \frac{1}{q}\left(\sum_{k=1}^{m-2} \frac{A_{m,m-k-1}}{q^{k+1}} + \frac{A_{m0}}{q^{m-1}}\right) + c\frac{q-1}{q^2}\sum_{k=1}^{m-1}\frac{1}{q^k} + \frac{q-1}{q^{m+1}} c \\ &= \frac{A_{m,m-1}}{q^2} + \frac{1}{q} S_{m-1} + \left(\frac{1}{q^2} + \frac{1}{q^m} - \frac{2}{q^{m+1}}\right) c. \end{aligned}$$

THEOREM 8. *For* $m \geqq 2$,

$$A_{m+1} = \left(1+\frac{1}{q^2}\right)A_m - \frac{1}{q^2}A_{m-1} + \left(\frac{q-1}{q}\right)^3 a + \left\{1 - \frac{3}{q^2} + \frac{2}{q^3} + \frac{q-1}{q^{m+2}}\right\} c.$$

Proof. Lemmas 6(a) and 7 give two expressions for S_m. In the latter, $A_{m,m-1}$ may be expanded using Lemma 6(b), and S_{m-1} may be expanded by Lemma 6(a). What results is an equation (with left-hand side involving A_{m+1} and A_m and right-hand side involving A_m and A_{m-1}) which simplifies to the above.

We now solve the recurrence relation using the method of generating functions and partial fractions.

Let G be the generating function $G(z) = \sum_{m=0}^{\infty} A_m z^m$.

LEMMA 9.

$$(z-1)(z-q^2)G(z) = q^2A_0 + \{q^2A_1 - (q^2+1)A_0\}z + \{q^2A_2 - (q^2+1)A_1 + A_0\}z^2 + \left\{\frac{(q-1)^3}{q}a + \left(q^2 - 3 + \frac{2}{q}\right)c\right\}\frac{z^3}{1-z} + \frac{q-1}{q}c\frac{z^3}{q-z}.$$

Proof. This is a straightforward consequence of Theorem 8.

PROPOSITION 10.

$$A_{10} = A_{00} + c \quad A_{11} = a + \frac{q-2}{q-1}(A_{00}+c) \quad A_{20} = A_{00} + \frac{2q-1}{q}c,$$

$$A_{21} = \frac{q-1}{q}(a + A_{00}) + \frac{2q-1}{q}c,$$

$$A_{22} = \left\{1 + \frac{(q-2)(q-1)}{q^2}\right\}a + \left\{\frac{(q-2)(q-1)}{q^2} + \frac{2q-3}{q(q-1)}\right\}A_{00} + \left\{\frac{(q-2)(2q-1)}{q^2} + \frac{(2q-3)(2q-1)}{(q-1)q^2}\right\}c.$$

Proof. These follow directly from the definition of A_{mn}.

LEMMA 11.

$$G(z) = \left(\frac{1-2q}{q}a + \frac{2q-1}{q-1}A_0 + \frac{c}{q(q-1)}\right) + \frac{qc}{q-1}\frac{1}{z-q} + \left\{\frac{(q-1)^2a}{q(q+1)} + \frac{q^2-3+2/q}{q^2-1}c\right\}\frac{1}{(z-1)^2} + \left\{\frac{q^3-4q^2-q+2}{q(q+1)^2}a - \frac{q}{q+1}A_0 + \frac{q^4+2q^3-4q^2+3}{q(q+1)^2(q-1)}c\right\}\frac{1}{z-1} + \left\{\frac{-a}{(q+1)^2} + \frac{A_0}{q^2-1} - \frac{c}{(q-1)(q+1)^2}\right\}\frac{2q^4}{z-q^2}.$$

Proof. This follows from Lemma 9 using the method of partial fractions (see, e.g., [BM, pp. 79-81]) and also uses the values for A_1, A_2 of the previous proposition.

PROPOSITION 12. $A_{mn} = ((q-1)/q)(m-n-1)c + A_{n+1,n}$ *if* $m > n \geqq 0$.

Proof. This follows from Proposition 2(b) by a simple change of subscripts.

Finally, the general solution is given in Theorem 13.

THEOREM 13. (a) *For* $m \geqq 3$

$$A_m = \left\{\frac{(q-1)^2}{q(q+1)}a + \frac{q^2-3+2/q}{q^2-1}c\right\}m + \frac{3q^2-1}{q(q+1)^2}a + \frac{q}{q+1}A_0 - \frac{q^3-q^2+q+1}{q(q-1)(q+1)^2}c - \frac{c}{(q-1)q^m} + \left\{\frac{a}{q+1} - \frac{A_0}{q-1} + \frac{c}{q^2-1}\right\}\frac{2}{q+1}q^{2-2m}.$$

(b) *For* $m-1 \geqq n \geqq 2$

$$A_{mn} = \frac{q-1}{q}mc + \left\{\frac{(q-1)^2}{q(q+1)}a + \frac{q-1}{q(q+1)}c\right\}n + \frac{2q^2-1}{q(q+1)^2}a + \frac{q}{q+1}A_0 + \frac{1}{q(q+1)}c + \left\{\frac{a}{q+1} - \frac{A_0}{q-1} + \frac{c}{q^2-1}\right\}\frac{q-1}{q+1}q^{1-2n}.$$

Proof. Part (a) is an immediate consequence of the definition of G and Lemma 11. Lemma 6(b) and part (a) give a formula for $A_{m+1,m}$, and an application of Proposition 12 completes the proof of part (b).

3. The average analysis of Algorithm B2. Two average running times for Algorithm B2 are calculated in this section: one for uniformly distributed normalized polynomials in degree $m \geqq n \geqq 0$ (Theorem 2), which is then used to determine the average (up to 0(1) terms) for uniformly distributed polynomials with degree in the range 0 to $N-1$, $N \geqq 1$ (Theorem 7); we also compare a new approximate model for the binary integer algorithm with the lattice point model, and determine the probability that normalized polynomials are coprime (Theorem 8).

Recall that $A_{mn}(a, c, A_{00})$ is defined in Definition 1, § 2.

PROPOSITION 1. $A_{mn}(1, 1, 1)$ *is the average number of iterations of Algorithm* B2 *for uniformly distributed normalized polynomials of degree* $m \geqq n \geqq 0$.

Proof. By virtue of Theorem 6, § 1, it suffices to show that the average is equal to $1 + \sum_{k=1}^{m} P[(m, n) \to (m-k, n)]A_{m-k,n}$. This is proved by induction on $m+n$, the case $m+n=0$ being clear. For non-negative integers i, r, s, let $\pi_{r,s}(i)$ be the probability that i iterations of Algorithm B2 are required to calculate the gcd of uniformly distributed normalized polynomials of degree r, s and set $\pi_{-1,s}(i) = \delta_{i0}$. Then for $m+n>0$ and $i>0$,

$$\pi_{mn}(i) = \sum_{k=1}^{m+1} P[(m, n) \to (m-k, n)]\pi_{m-k,n}(i-1)$$

and the average in question can be written as

$$\begin{aligned}\sum_{i \geqq 1} i\pi_{mn}(i) &= \sum_{i \geqq 1}\sum_{k=1}^{m+1} P[(m, n) \to (m-k, n)](1+(i-1))\pi_{m-k,n}(i-1) \\ &= \sum_{k=1}^{m+1} P[(m, n) \to (m-k, n)]\left(1 + \sum_{i \geqq 0} i\pi_{m-k,n}(i)\right) \\ &= 1 + \sum_{k=1}^{m+1} P[(m, n) \to (m-k, n)] \sum_{i \geqq 1} i\pi_{m-k,n}(i) \\ &= 1 + \sum_{k=1}^{m} P[(m, n) \to (m-k, n)]A_{m-k,n}\end{aligned}$$

by the inductive hypothesis and the definition of $\pi_{-1,n}$.

Theorem A of the Introduction is an immediate consequence of the following result.

THEOREM 2. *For uniformly distributed normalized $GF(q)$ polynomials of degree m, n, the average number of iterations of Algorithm* B2 *needed to compute* $\gcd(u, v)$ *is*

$$\begin{cases} 1 & \text{if } m = n = 0, \\ \dfrac{q-1}{q}\, m + \dfrac{1}{q} + 1 & \text{if } m > n = 0, \\ \dfrac{(q-1)(2q+1)}{(q+1)q}\, m + \dfrac{q^3+2q^2-3q-2}{(q-1)(q+2)^2} - \dfrac{q^{-m}}{q-1} - \dfrac{2q^{2-2m}}{(q-1)(q+1)^2} & \text{if } m = n \geqq 1, \\ \dfrac{q-1}{q}\, m + \dfrac{q-1}{q+1}\, n + \dfrac{q^2+3q+1}{(q+1)^2} - \dfrac{q^{1-2n}}{(q+1)^2} & \text{if } m > n \geqq 1. \end{cases}$$

Proof. This follows from the preceding proposition and the results of § 2: for $m = n \geqq 3$ or $m > n \geqq 2$ from Theorem 2.13 and the remaining cases from Proposition 10, § 2.

Recall that there is an integer analogue of Algorithm B2, which repeatedly subtracts (non-negative) *odd* integers rather than normalized $GF(2)$ polynomials. Thus for 17 and 5, the iterations would be

$$(17, 5) = (5, 3) = (3, 1) = (1, 1) = (0, 1) = 1.$$

Further, the lattice-point model [*K*, p. 330 ff.] is similar to the above model for $q = 2$. Indeed, the lattice-point model uses the same probabilities as Theorem 6(a), § 1, for the case $m > n$ and differs only in the transition probabilities when $m = n$ and $0 \leqq k \leqq n-1$. It is therefore natural to compare the solutions of these two models *in abstracto.*

Thus, ignoring $o(1)$ terms and setting $q = 2$ in Theorem 2 gives $m/2 + n/3 + 11/9 - \delta_{mn}/3$ as another estimate of the average number of iterations of the binary gcd algorithm for uniformly and independently distributed odd $(m+1)$- and $(n+1)$-bit integers where m, n are both nonzero. This model was compared to the lattice point model as follows: fix m, n with $1 \leqq n \leqq m \leqq 7$. Compute the actual average a_{mn} using all 2^{m+n-2} possible pairs of $(m+1)$- and $(n+1)$-bit odd integers. This gives 28 actual averages. Now compute the error terms $(a_{mn} - A_{mn})^2$, where $A_{mn} = m/2 + n/3 + 11/9 - \delta_{mn}/3$. The average of these squares (the mean square error of the model) is 0.144, whereas the corresponding value for the lattice point model is 0.213. It would seem therefore that the transition probabilities of Theorem 1.6 are a slightly better approximation to the actual ones.

We now deal with the case where the inputs are uniformly distributed with degree in the range 0 to $N-1$, $N \geqq 1$. The approach is completely analogous to that used in [K, p. 335].

LEMMA 3. *Let $N \geqq 1$ and let u, v be $GF(q)$ polynomials, uniformly distributed with degree in the range* 0 *to* $N-1$. *Then, the average number of iterations C of Algorithm* B2 *satisfies*

$$\begin{aligned} (q^N-1)^2 C = {} & (q-1)^2 N^2 C_{00} + 2(q-1)N \sum_{n=1}^{N-1} (q-1)^2 q^{n-1} (N-n) C_{n0} \\ & + 2 \sum_{1 \leqq n < m < N} (q-1)^4 q^{m+n-2} (N-m)(N-n) C_{mn} \\ & + \sum_{n=1}^{N-1} \{(q-1)^2 q^{n-1} (N-n)\}^2 C_{nn} \end{aligned}$$

where C_{mn} denotes the average of Theorem 2.

Proof. There are $(q-1)N$ polynomials with degree in the range 0 to $N-1$ which can be shifted right until they have degree 0, and there are $(q-1)^2q^{m-1}(N-m)$ polynomials with degree in the range m to $N-1$ which can be shifted right until they have degree $m>0$.

LEMMA 4. (a) *For* $m \geqq 2$,

$$\sum_{n=1}^{m} nx^n = \frac{x}{(x-1)^2}[mx^{m+1}-(m+1)x^m+1].$$

(b) *For* $m \geqq 2$,

$$\sum_{n=1}^{m} n(n-1)x^n = \frac{x^2}{(x-1)^3}\{m(m-1)x^{m+1}-2(m-1)(m+1)x^m+m(m+1)x^{m-1}-2\}.$$

Proof. Part (a) is obtained in the standard way by differentiating the geometric series with ratio x, and part (b) follows from the series of part (a) in the same way.

LEMMA 5. *If* $C_{mn} = \alpha m+\beta n+\gamma$, $m \geqq n$, *then*

$$\text{(a)} \quad \sum_{1\leqq n<m<N} (N-m)(N-n)q^{m+n-2}C_{mn} = q^{2N}\left(\frac{2q^2+q+1}{(q-1)^4(q+1)^3}(\alpha+\beta)N+0(1)\right)$$

$$\text{(b)} \quad \sum_{1\leqq n<N} (N-n)^2q^{2n-2}C_{nn} = q^{2N}\left(\frac{q^2+1}{(q^2-1)^3}(\alpha+\beta)N+0(1)\right).$$

Proof. Both parts follow from Lemma 4 by a routine calculation similar to [K, p. 597].

THEOREM 6. *Let* $N \geqq 1$. *For uniformly and independently distributed* $GF(q)$ *polynomials with degree in the range* 0 *to* $N-1$, *the average number of iterations of Algorithm* B2 *is*

$$\frac{(q-1)(2q+1)}{(q+1)q}N+0(1).$$

Proof. By Lemmas 3 and 5 and by Theorem 2, the coefficient of N in the uniform average C is

$$(q-1)^4\left\{\frac{2(2q^2+q+1)}{(q-1)^4(q+1)^3}\left(\frac{q-1}{q}+\frac{q-1}{q+1}\right)+\frac{q^2+1}{(q^2-1)^3}\left(\frac{q-1}{q+1}\cdot\frac{2q+1}{q}\right)\right\}$$

and this simplifies to the stated value.

We conclude this section with another application of the results of § 2.

THEOREM 7. *Let* f, g *be uniformly distributed normalized* $GF(q)$ *polynomials of degree* m, n, *respectively. Then the probability that* f *and* g *are coprime is*

$$\begin{cases} 1 & \text{if } m \geqq n = 0, \\ \dfrac{q}{q+1}-\dfrac{2}{(q^2-1)q^{2m-2}} & \text{if } m = n \geqq 1, \\ \dfrac{q}{q+1}-\dfrac{1}{(q+1)q^{2n-1}} & \text{if } m > n \geqq 1. \end{cases}$$

Proof. As in the proof of Proposition 1, the required probability is clearly $A_{mn}(0, 0, 1)$, and, so for $m = n \geqq 3$ or $m > n \geqq 2$, the result follows from Theorem 13, § 2. The reader may check that in the remaining cases the stated formulae agree with Proposition 10 when $a = c = 0$ and $A_{00} = 1$.

Theorem 7 is also a simple consequence of the following result, which is due to an anonymous referee. First some notation is given:

for $m, n \geqq 0$, let
$P_m = \{u \in GF(q)[x]: \deg u = m, u \text{ monic and normalized}\}$
$R_{mn} = \{(u, v) \in P_m \times P_n : \gcd(u, v) = 1\}$
$r_{mn} = |R_{mn}|$.

THEOREM 8.

$$r_{mn} = \begin{cases} 1 & \text{if } m = n = 0, \\ q^{m-1}(q-1) & \text{if } m > n = 0, \\ \dfrac{q-1}{q+1}(q^{2n} - q^{2n-1} - 2) & \text{if } m = n > 0, \\ \dfrac{(q-1)^2}{q+1} q^{m-n-1}(q^{2n} - 1) & \text{if } m > n > 0. \end{cases}$$

Proof. The first two cases are trivial since $|P_0| = 1$ and for $m > 0$, $|P_m| = q^{m-1}(q-1)$. Fix $m \geqq n \geqq 1$, and for $1 \leqq k \leqq n$, define $\alpha_k : R_{m-k,n-k} \times P_k \to P_m \times P_n$ by $\alpha_k((u, v), w) = (uw, vw)$. To see that α_k is one-to-one onto its image, suppose that $(s, t) \in P_m \times P_n$ are *not* relatively prime. Put $g = \gcd(s, t)$, where $d = \deg g \geqq 1$. It is elementary that

$$\alpha_k^{-1}(s, t) = \begin{cases} \varphi & \text{if } k \neq d \\ ((s/g, t/g), g) & \text{if } k = d \end{cases}$$

so that each α_k is one-to-one and $P_m \times P_n = R_{mn} \cup \cup_{k=1}^{n} im(\alpha_k)$ is a disjoint union. Since $|P_m| = q^{m-1}(q-1)$, this gives the double recurrence relation

$$r_{mn} = q^{m+n-2}(q-1)^2 - \sum_{k=1}^{n} r_{m-k,n-k} q^{k-1}(q-1).$$

The remaining two formulae for r_{mn} are now straightforward verifications based on the first two cases.

We note in conclusion that the same argument shows that the probability that *unnormalized* polynomials (of positive degree) are relatively prime is $(q-1)/q$: (cf. [K, p. 417, #5]). To wit, let

$$Q_n = \{u \in GF(q)[x]: \deg u = n, u \text{ monic}\},$$
$$S_{mn} = \{(u, v) \in Q_m \times Q_n : \gcd(u, v) = 1\},$$
$$s_{mn} = |S_{mn}|.$$

THEOREM 9.

$$s_{mn} = \begin{cases} 1 & \text{if } m = n = 0, \\ q^m & \text{if } m > n = 0, \\ q^{m+n+1}(q-1) & \text{if } m \geqq n > 0. \end{cases}$$

Proof. As for Theorem 8, but using the fact that $|Q_m| = q^m$ for $m \geqq 0$.

4. The worst-case analysis of Algorithm B2. The worst-case analysis of Algorithm B2 for $GF(q)[x]$ differs when $q = 2$ and when $q > 2$. A more elaborate polynomial

construction, as well as a more elaborate argument, is needed in the former case. Both constructions incorporate a type of polynomial Fibonacci term.

PROPOSITION 1. *Let* $m \geqq n \geqq 0$. *If* $u, v \in K[x]$ *have degree* m, n, *respectively, then Algorithm* B2 *requires at most* $m+n+1$ *iterations to compute* $\gcd(u, v)$.

Proof. This is completely analogous to the proof of the integer case given in [N, § A] and is omitted.

If K is $GF(2)$, it is easy to check that the upper bound of Proposition 1 is the best possible if either polynomial has degree 0 or 1, but Algorithm B2 applied to $u(x) = x^2 + x + 1$ and $v(x) = x + 1$ requires three iterations, less than the expected upper bound of 4.

PROPOSITION 2. *If* $u, v \in GF(2)[x]$ *have degree* m, n, *respectively, where* $m \geqq n \geqq 0$, *then Algorithm* B2 *requires at most* $m + \lfloor n/2 \rfloor + 1$ *iterations to compute* $\gcd(u, v)$.

Proof. Without loss of generality, we may assume that u and v are normalized. We use induction on $m+n$. The result is clearly true for $m = n = 0$. Suppose that $m+n > 0$ and the first iteration yields polynomials of degree $m-k$, n for some k, $1 \leqq k \leqq m+1$. If $m = n$, at most $m + \lfloor (m-k)/2 \rfloor + 1$ further steps are required where $k \geqq 2$, since $u, v \in GF(2)[x]$. If $k = m+1$, the algorithm terminates and there is nothing to prove. If $m > n$, either $m - k + \lfloor n/2 \rfloor + 1$ or $n + \lfloor (m-k)/2 \rfloor + 1$ further steps are required, according as $m - k \geqq n$ or $m - k \leqq n$. In all cases, the inductive hypothesis implies that at most $m + \lfloor n/2 \rfloor + 1$ steps are needed.

Note that the preceding proof works because $m = n \Rightarrow k \geqq 2$ where $q = 2$. This fails for $q \geqq 3$ since there is no longer automatic cancellation of *leading* terms.

It will be shown that the following construction realises the previous upper bound; note that for m even, u_{mm} is defined in a way reminiscent of Fibonacci numbers.

DEFINITION 3. Define $u_{mn}, v_{mn} \in GF(2)[x]$ by $u_{00}(x) = v_{00}(x) = 1$ and

$$v_{m+1,n}(x) = \begin{cases} v_{mn}(x) & \text{if } n \text{ is even and } 0 \leqq n \leqq m, \\ x v_{m+1,n-1}(x) & \text{if } n \text{ is odd and } 1 \leqq n \leqq m+1; \end{cases}$$

$$u_{m+1,n}(x) = \begin{cases} x u_{mn}(x) + v_{m+1,n}(x) & \text{if } n \text{ is even and } 0 \leqq n \leqq m, \\ u_{m+1,n-1}(x) & \text{if } n \text{ is odd and } 1 \leqq n \leqq m+1; \end{cases}$$

$$v_{m+1,m+1}(x) = u_{m+1,m}(x) \quad \text{if } m \text{ is odd},$$

$$u_{m+1,m+1}(x) = u_{m+1,m}(x) + v_{m+1,m}(x) \quad \text{if } m \text{ is odd}.$$

The reader may check that $\deg(u_{mn}) = m$, $\deg(v_{mn}) = n$ by inducting on $m+n$.

LEMMA 4. (a) *For all* $m \geqq n \geqq 0$, $u_{mn}(0) \neq 0$,

(b) *If* n *is even*, $v_{nn}(0) \neq 0$.

Proof. This is trivial.

LEMMA 5. *Let* $m \geqq n \geqq 0$ *and let* $P(m, n)$ *be the proposition that Algorithm* B2 *requires* $m + \lfloor n/2 \rfloor + 1$ *steps to compute* $\gcd(u_{mn}, v_{mn})$.

(a) $P(n+1, n) \Rightarrow P(n+1, n+1)$

(b) if n is even, $P(m, n) \Rightarrow P(m+1, n)$

(c) if n is odd, $P(m, n-1) \Rightarrow P(m, n)$.

Proof. This follows from Definition 3 and from Lemma 4 using the parities of m and n, and is omitted.

PROPOSITION 6. *For* $m \geqq n \geqq 0$, *Algorithm* B2 *computes* $\gcd(u_{mn}, v_{mn})$ *in* $m + \lfloor n/2 \rfloor + 1$ *steps.*

Proof. The proof is by induction on $m+n$. $P(0, 0)$ is true. The case $m = n$ follows from part (a) of Lemma 5, and the case $m > n$ follows from part (b) or part (c) depending on whether n is even or odd.

The last topic is the worst case for $q>2$. This is easier than the case $q=2$, being independent of parity and attaining the first upper bound.

DEFINITION 7. For $q>2$ and $m>n\geqq 0$ define u_{mn}, $v_{mn}\in GF(q)[x]$ by

$$u_{00}(x)=v_{00}(x)=1,$$

$$u_{mn}(x)=xu_{m-1,n}(x)+v_{nn}(x),$$

$$v_{mn}(x)=v_{nn}(x),$$

$$v_{n+1,n+1}(x)=u_{n+1,n}(x),$$

$$u_{n+1,n+1}(x)=M(xv_{nn}(x)+tv_{n+1,n+1}(x)),$$

where $M(f)=L(f)^{-1}f$ makes $f\in GF(q)[x]$ monic, and $t\neq 0,-1$.

The condition on t easily implies that (a) $\deg(u_{mn})=m$ and $\deg(v_{mn})=n$ and (b) $u_{n+1,n}(0)\neq 0$, $v_{nn}(0)\neq 0$ for all $n\geqq 0$. This latter property and a simple induction establish the last result.

PROPOSITION 8. *For $q>2$, Algorithm* B2 *requires $m+n+1$ iterations to compute* $\gcd(u_{mn}, v_{mn})$.

It follows that for $q>2$, the upper bound $m+n+1$ of Proposition 1 is tight.

5. The left-shift algorithm. This section states the left-shift gcd algorithm for $GF(q)[x]$ and determines the transition probabilities. These in turn yield the recurrences to be solved for the average analyses. Recall that for non-zero $g\in K[x]$, $L[g]$ is the leading coefficient of g.

ALGORITHM 1. GCD by Left shifting.
Input. Polynomials f, $g\in K[x]$ with g non-zero.
Output. $\gcd(f, g)$.

procedure A2 $\left(\sum_{i=0}^{m} f_i x^i, \sum_{j=0}^{n} g_j x^j\right)$;

```
while f ≠ 0 do begin
    if deg(f) < deg(g) then swap (f, g);
    f := L(g)f − L(f)x^(deg(f)−deg(g)) g;
    end;
return L(g)^(−1) g.
```

EXAMPLE 2. Over $GF(2)$, the iterations are

$$(x^5+x^2+1, x^4+x+1)=(x^4+x+1, x+1)=(x^3+x+1, x+1)$$

$$=(x^2+x+1, x+1)=(x+1, 1)=(1, 1)=(0, 1)=1.$$

We now show that Algorithm A2 preserves uniformity.

PROPOSITION 3. *Let $i\geqq 1$, let u, v be uniformly distributed $GF(q)$ polynomials of degree $m\geqq n\geqq 0$, and let $1\leqq k\leqq m+1$, $0\leqq l\leqq n$. Suppose that after i iterations, (u, v) is replaced by (u', v') of degree $m-k$, $n-l$, respectively. Then u', v' are uniformly distributed in degree $m-k$, $n-l$.*

Proof. This is similar to Proposition 4, § 1, and is omitted.

DEFINITION 4. For $m\geqq n\geqq 0$ and $1\leqq k\leqq m+1$, let $P[(m, n)\to(m-k, n)]$ denote the probability that one iteration of Algorithm A2 replaces polynomials of degree m, n by polynomials of degree $m-k$, n, or terminates (if $k=m+1$).

PROPOSITION 5. *For* $m \geqq n \geqq 0$

$$P[(m, n) \to (m-k, n)] = \begin{cases} \dfrac{q-1}{q^k} & \text{if } 1 \leqq k \leqq m \\ \dfrac{1}{q^m} & \text{if } k = m+1. \end{cases}$$

Proof. Suppose that $m > n$ and let u, v be of degree m, n, respectively. The degree $m-k$ of the result falls into one of three cases: $m-n \leqq m-k \leqq m-1$, $0 \leqq m-k \leqq m-n-1$, and $m-k=-1$. The first case requires $v_n u_j = u_m v_{j+n-m}$ for $m-k+1 \leqq j \leqq m-1$ and $v_n u_{m-k} \neq u_m v_{n-k}$. The second requires $v_n u_j = u_m v_{j+n-m}$ for $m-n \leqq j \leqq m-1$, $u_j = 0$ for $m-k+1 \leqq j \leqq m-n-1$, and $u_{m-k} \neq 0$, whereas the last requires that $v_n u_j = u_m v_{j+n-m}$ for $m-n \leqq j \leqq m-1$ and $u_j = 0$ for $0 \leqq j \leqq m-n-1$. Each equality occurs with probability $1/q$, and each of the first two cases has $k-1$ equalities and one inequality, whereas the last case has m equalities.

The case $m = n$ is similar (requiring the first and last case only) without the zero conditions, and is omitted.

6. The average analysis of Algorithm A2. Two average running times for Algorithm A2 are calculated in this section: one for uniformly distributed polynomials in degree $m \geqq n \geqq 0$ (Corollary 5), which is then used to determine the average (up to 0(1) terms) for uniformly distributed polynomials with degree in the range 0 to $N-1$, $N \geqq 1$ (Theorem 8).

DEFINITION 1. Let A_{mn} be given by

$$A_{mn} = \begin{cases} 1 & \text{if } m = n = 0, \\ 1 + \sum_{k=1}^{m} \dfrac{q-1}{q^k} A_{m-k,n} & \text{if } m \geqq n \geqq 0, (m, n) \neq (0, 0), \\ A_{nm} & \text{if } n \geqq m \geqq 0, (m, n) \neq (0, 0). \end{cases}$$

As in Proposition 1, § 3, A_{mn} is the required average number of iterations of Algorithm A2. The following results are used to determine the recurrence satisfied by A_{nn} and to express A_{mn} in terms of A_{nn}.

PROPOSITION 2. *For* $m \geqq n \geqq 0$ *and* $k \geqq 0$,

$$A_{m+k,n} = A_{m,n} + k\left(\frac{q-1}{q}\right).$$

Proof.

$$\begin{aligned} A_{m+1,n} &= 1 + \sum_{k=1}^{m+1} \frac{q-1}{q^k} A_{m+1-k,n}, \\ &= 1 + \frac{q-1}{q} A_{mn} + \sum_{k=2}^{m+1} \frac{q-1}{q^k} A_{m+1-k,n}, \\ &= 1 + \frac{q-1}{q} A_{mn} + \frac{1}{q} \sum_{k=1}^{m} \frac{q-1}{q^k} A_{m-k,n}, \\ &= 1 + \frac{q-1}{q} A_{mn} + \frac{1}{q}(A_{mn} - 1), \\ &= A_{mn} + \frac{q-1}{q}. \end{aligned}$$

The general result now follows by induction on k.

For simplicity, let $A_m = A_{mm}$ for $m \geqq 0$.

PROPOSITION 3. *For* $m \geqq 0$,

$$A_{m+1} = A_m + \left(\frac{q-1}{q}\right)\left(2 - \frac{1}{q^{m+1}}\right).$$

Proof.

$$\begin{aligned}
A_{m+1} &= 1 + \sum_{k=1}^{m+1} \frac{q-1}{q^k} A_{m+1-k,m+1}, \\
&= 1 + \frac{q-1}{q} A_{m,m+1} + \sum_{k=2}^{m+1} \frac{q-1}{q^k} A_{m+1-k,m+1}, \\
&= 1 + \frac{q-1}{q}\left(A_m + \frac{q-1}{q}\right) + \frac{1}{q}\sum_{k=1}^{m} \frac{q-1}{q^k} A_{m-k,m+1}, \\
&= 1 + \frac{q-1}{q} A_m + \left(\frac{q-1}{q}\right)^2 + \frac{1}{q}\sum_{k=1}^{m}\left(\frac{q-1}{q^k} A_{m-k,m} + \frac{q-1}{q}\right), \\
&= 1 + \frac{q-1}{q} A_m + \frac{1}{q}(A_m - 1) + \left(\frac{q-1}{q}\right)^2 \sum_{k=1}^{m} \frac{1}{q^k}, \\
&= A_m + \frac{q-1}{q} + \left(\frac{q-1}{q}\right)^2 \frac{1-q^{m+1}}{q^{m+1}} \frac{q}{1-q}, \\
&= A_m + \frac{q-1}{q} - \frac{q-1}{q}\left(\frac{1}{q^{m+1}} - 1\right), \\
&= A_m + \left(\frac{q-1}{q}\right)\left(2 - \frac{1}{q^{m+1}}\right).
\end{aligned}$$

The recurrence of Proposition 3 is readily solved using the generating function (Proposition 4).

PROPOSITION 4. *For* $m \geqq 0$,

$$A_{m+1} = (2(m+1)+1)\left(\frac{q-1}{q}\right) + \frac{1}{q^{m+2}}.$$

Proof. Let $G(z) = \sum_{m=0}^{\infty} A_m z^m$. Proposition 3 implies that

$$G(z) - 1 - zG(z) = 2\left(\frac{q-1}{q}\right) z \sum_{m=0}^{\infty} z^m - \frac{q-1}{q}\frac{z}{q} \sum_{m=0}^{\infty} \left(\frac{z}{q}\right)^m,$$

and the method of partial fractions (see, e.g., [BM, pp. 79–81]) yields

$$G(z) = \sum_{m \geqq 0} z^m + 2\left(\frac{q-1}{q}\right) \sum_{m \geqq 0} (m+1) z^{m+1} - \frac{z}{q} \sum_{m \geqq 0} z^m + \frac{z}{q^2} \sum_{m \geqq 0} \left(\frac{z}{q}\right)^m.$$

Collecting terms on the right-hand side yields the formula for A_{m+1}.

COROLLARY 5. *Let* $m \geqq n \geqq 0$. *For uniformly distributed* $u, v \in GF(q)[x]$ *of degree* m, n, *respectively, the average number of iterations required to compute* $\gcd(u, v)$ *using Algorithm* A2 *is* $(m+n+1)(1-1/q)+1/q^{n+1}$.

Proof. By Proposition 2, $A_{mn} = (m-n)(1-1/q) + A_n$. Now $A_{00} = 1$ and if $n > 0$, A_n is given by Proposition 4. Combining these formulae yields the result.

The remainder of this section deals with the case where the inputs are uniformly distributed with degree in the range 0 to $N-1$, $N \geqq 1$.

LEMMA 6. *Let $N \geqq 1$ and let u, v be $GF(q)$ polynomials, independently and uniformly distributed with degree in the range 0 to $N-1$. Then C, the average number of iterations of Algorithm* A2 *satisfies*

$$(q^N-1)^2 C = \sum_{0 \leqq m,n < N} q^{m+n}(q-1)^2 C_{mn}$$

where C_{mn} denotes the average of Corollary 5.

Proof. There are $(q-1)q^t$ $GF(q)$ polynomials of degree $t \geqq 0$.

LEMMA 7. *If $m \geqq n \geqq 0$ and $C_{mn} = \alpha m + \beta n + \gamma$, then*

$$\text{(a)} \quad \sum_{0 \leqq n < m < N} q^{m+n}(q-1)^2 C_{mn} = q^{2N}\left(\frac{1}{q+1}(\alpha+\beta+\varepsilon)N + 0(1)\right)$$

$$\text{(b)} \quad \sum_{0 \leqq n < N} q^{2n}(q-1)^2 C_{nn} = q^{2N}\left(\frac{q-1}{q+1}(\alpha+\beta+\varepsilon)N + 0(1)\right)$$

where ε tends to zero with q^{-N}.

Proof. These are routine summations using the change of indices $r = N-m$, $s = N-n$.

THEOREM 8. *Let $N \geqq 1$. For uniformly and independently distributed $GF(q)$ polynomials with degree in the range 0 to $N-1$, the average number of iterations of Algorithm* A2 *is $(2(1-1/q)+\varepsilon)N + 0(1)$, where ε tends to zero with q^{-N}.*

Proof. By Lemmas 6 and 7 and by Corollary 5, the coefficient of N in the uniform average is

$$\left\{\frac{2}{q+1} + \frac{q-1}{q+1}\right\}\left(2\frac{q-1}{q} + \varepsilon\right)$$

as required.

7. The worst-case analysis of Algorithm A2. This short section gives a simple polynomial construction to realise the maximum number of iterations for Algorithm A2.

PROPOSITION 1. *Let $m \geqq n \geqq 0$. If u, $v \in K[x]$ have degree m, n, respectively, then Algorithm* A2 *requires at most $m+n+1$ iterations to compute* gcd (u, v).

Proof. This is a trivial induction on $m+n \geqq 0$.

DEFINITION 2. Define u_{mn}, $v_{mn} \in GF(q)[x]$ by

$$u_{00}(x) = v_{00}(x) = 1,$$

$$v_{mn}(x) = u_{n,n-1}(x) \quad \text{if } m \geqq n \geqq 1,$$

$$u_{nn}(x) = v_{n-1,n-1}(x) + u_{n,n-1}(x) \quad \text{if } n \geqq 1,$$

$$u_{mn}(x) = x^{m-n}v_{nn}(x) + u_{m-1,n}(x) \quad \text{if } m > n \geqq 0.$$

It is trivial to verify that deg $(u_{mn}) = m$, deg $(v_{mn}) = n$ and that each u_{mn}, v_{mn} is monic.

PROPOSITION 3. *For $q \geqq 2$ and $m \geqq n \geqq 0$, Algorithm* A2 *requires $m+n+1$ iterations to compute* gcd (u_{mn}, v_{mn}).

Proof. The result is true for $m = n = 0$. The general result follows by induction on $m+n$ since one iteration replaces (u_{mn}, v_{mn}) by $(u_{m-1,n}, v_{m-1,n})$ if $m > n$ and (u_{nn}, v_{nn}) by $(u_{n,n-1}, v_{n-1,n-1})$ otherwise.

It is not hard to see that the Euclidean algorithm requires $n+1$ steps to compute gcd (u_{mn}, v_{mn}).

Acknowledgments. I am indebted to the referees for their valuable comments and especially to "referee A" for his report on a first version of this paper. The first version

discussed the average number of subtraction steps as well as the average number of shifts, but for the case $q = 2$ only, and conjectured Theorem 3.7. Referee A kindly supplied a proof of this (see Theorem 3.8). His first report thus led to the current solution. The second reports of the referees also suggested many improvements. The time that they have spent is much appreciated.

REFERENCES

[BK] R. P. BRENT AND H. T. KUNG, *Systolic* VLSI *arrays for polynomial* GCD *Computation*, IEEE Trans. Comput., C-33 (1984), pp. 731–736.

[BM] G. BIRKHOFF AND S. MACLANE, *A Survey of Modern Algebra*, Third edition, MacMillan, London, 1965.

[K] D. E. KNUTH, *The Art of Computer Programming*, Vol. 2, *Seminumerical Algorithms*, Second edition, Addison Wesley, Reading, MA, 1981.

[N] G. H. NORTON, *Extending the binary* GCD *algorithm*, in Algebraic Algorithms and Error Correcting Codes, Springer Lecture Notes in Computer Science, Vol. 229, (1986), pp. 363–372.

[No] G. H. NORTON, *A unified design and analysis of some* GCD *algorithms*, 1986, September, preprint.

SIAM J. COMPUT.
Vol. 18, No. 3, pp. 625–638, June 1989

014

EXPRESSIBILITY AND PARALLEL COMPLEXITY*

NEIL IMMERMAN†

Abstract. It is shown that the time needed by a concurrent-read, concurrent-write parallel random access machine (CRAM) to check if an input has a certain property is the same as the minimal depth of a first-order inductive definition of the property. This in turn is equal to the number of "iterations" of a first-order sentence needed to express the property.

The second contribution of this paper is the introduction of a purely syntactic uniformity notion for circuits. It is shown that an equivalent definition for the uniform circuit classes AC^i, $i \geq 1$ is given by first-order sentences "iterated" $\log^i n$ times. Similarly, uniform AC^0 is defined to be the first-order expressible properties (which in turn is equal to constant time on a CRAM by our main theorem). A corollary of our main result is a new characterization of the Polynomial-Time Hierarchy (PH): PH is equal to the set of languages accepted by a CRAM using exponentially many processors and constant time.

Key words. computational complexity, parallel complexity, first-order expressibility, polynomial-time hierarchy

AMS(MOS) subject classification. 68025

1. Introduction.

Parallel time on a random access machine has a surprisingly simple mathematical definition involving well-studied objects of mathematical logic. We show that the time needed by a concurrent-read, concurrent-write parallel random access machine (CRAM) to check if an input has a certain property is the same as the minimal depth of a first-order inductive definition of the property. This in turn is equal to the number of "iterations" of a first-order sentence needed to express the property.

We now state our main result. (See §2 for relevant definitions. In particular, the iteration of a first-order sentence is defined in §2.2, and the CRAM is defined in §2.3. The definition of the CRAM differs from the standard definition of the CRCW PRAM in [17] only in that a processor may shift a word of local memory by any polynomial number of bits in unit time. It follows from our results that for parallel time greater than or equal to $\log n$ there is no distinction between the models with and without the Shift instruction.)

THEOREM 1.1. *Let S be a set of structures of some vocabulary τ. For example, S is a set of boolean strings, or a set of graphs, etc. For all polynomially bounded, parallel time constructible $t(n)$, the following are equivalent:*

1. *S is recognizable by a* CRAM *in parallel time $t(n)$, using polynomially many processors.*

2. *There exists a first-order sentence φ such that the property S for structures of size at most n is expressed by φ iterated $t(n)$ times.*

3. *S is definable as a uniform first-order induction whose depth, for structures of size n, is at most $t(n)$.*

For $t(n) \geq \log n$, the equivalence of (1) and (2) in Theorem 1.1 may also be obtained by combining a result of Ruzzo and Tompa relating CRAMs to alternating Turing machines [17, Thm. 3], together with a result of ours relating alternating Turing machines to first-order expressibility [9, Thm. B.4]. In order to prove the

* Received by the editors June 13, 1987; accepted for publication (in revised form) August 11, 1988. This research was supported in part by the Mathematical Sciences Research Institute, Berkeley, California, and by National Science Foundation grants DCR-8603346 and CCR-8806308.

† Computer Science Department, Yale University, New Haven, Connecticut 06520.

theorem for $t(n) < \log n$, we were forced to modify the models slightly, adding the Shift operation to the CRAMs and adding BIT as a new logical relation to our first-order language (see §2). We believe that the naturalness of Theorem 1.1 justifies these modifications.

This paper is organized as follows. In §2 we give all relevant definitions. In §3 we prove our main result. In §4 we give a more detailed analysis of the bounds in Theorem 1.1. We show that the number of distinct variables in a first-order inductive definition is closely tied to the number of processors in the corresponding CRAM.

Until now, a principal unaesthetic feature of the theory of complexity via boolean circuits was that one had resorted to Turing machines to define the uniformity conditions for circuits [15]. As a corollary to Theorem 1.1, we obtain a purely syntactic uniformity notion for circuits. In §5 we describe this result as well as other relations between circuits and first-order complexity.

As another corollary to Theorem 1.1, we present in §6 a new characterization of the Polynomial-Time Hierarchy (PH): PH is equal to the set of languages recognized by a CRAM using exponentially many processors and constant time. In §7 we give some suggestions for future work in this area.

2. Background and definitions.

2.1. First-order logic. We begin this section by making some precise definitions concerning first-order logic. For more information see [4].

A *vocabulary* $\tau = \langle \underline{R}_1^{a_1}, \cdots, \underline{R}_k^{a_k}, \underline{c}_1, \cdots, \underline{c}_r \rangle$ is a tuple of relation symbols and constant symbols. $\underline{R}_i^{a_i}$ is a relation symbol of arity a_i. In the sequel we will usually omit the superscripts and the underlines to improve readability. A finite *structure* of vocabulary τ is a tuple, $\mathcal{A} = \langle \{0, 1, \cdots, n-1\}, R_1^{\mathcal{A}}, \cdots, R_k^{\mathcal{A}}, c_1^{\mathcal{A}}, \cdots, c_r^{\mathcal{A}} \rangle$, consisting of a universe $|\mathcal{A}| = n = \{0, \cdots, n-1\}$ and relations $R_1^{\mathcal{A}}, \cdots, R_k^{\mathcal{A}}$ of arities $a_1, \cdots, a_k$ on $|\mathcal{A}|$ corresponding to the relation symbols $\underline{R}_1^{a_1}, \cdots, \underline{R}_k^{a_k}$ of τ, and constants $c_1^{\mathcal{A}}, \cdots, c_r^{\mathcal{A}}$ from $|\mathcal{A}|$ corresponding to the constant symbols $\underline{c}_1, \cdots, \underline{c}_r$ from τ.

For example, a graph on n vertices, $G = \langle \{0 \cdots n-1\}, E \rangle$, is a structure whose vocabulary $\tau_0 = \langle \underline{E}^2 \rangle$ has a single binary relation symbol. Similarly, a binary string of length n is a structure $S = \langle \{0 \cdots n-1\}, M \rangle$, whose vocabulary $\tau_1 = \langle \underline{M}^1 \rangle$ consists of a single unary relation symbol. Here the ith bit of S is 1 if and only if $S \models M(i)$.

Let the symbol "$\leq$" denote the usual ordering on the natural numbers. We will include $\leq$ as a logical relation in our first-order languages. This seems necessary in order to simulate machines whose inputs are structures given in some order. It is convenient to include logical constant symbols, $0, 1, \cdots$, referring to the zeroth, first, etc., elements of the universe, respectively. (If the universe is smaller than a given constant, then interpret that constant as 0.) We also include the logical predicate BIT, where $\mathrm{BIT}(x, y)$ holds if and only if the xth bit in the binary expansion of y is a one.[1]

We now define the *first-order language* $\mathcal{L}(\tau)$ to be the set of formulas built up from the relation and constant symbols of τ and the logical relation and constant symbols, $=, \leq, \mathrm{BIT}, 0, 1, \cdots$, using logical connectives, $\wedge, \vee, \neg$, variables, $x, y, z, \cdots$, and quantifiers, $\forall, \exists$.

We will think of a *problem* as a set of structures of some vocabulary τ. It suffices to consider only problems on binary strings, but it is more interesting to be able to talk about other vocabularies, e.g., graph problems, as well. For definiteness, we will

[1] The relation BIT is crucial for the truth of Theorem 1.1, when $t(n) < \log n$, and for the plausibility of Definition 5.5.

fix a scheme for coding an input structure as a binary string. If $\mathcal{A} = \langle \{0, 1, \cdots, n-1\}, R_1^{\mathcal{A}} \cdots R_k^{\mathcal{A}}, c_1^{\mathcal{A}} \cdots c_r^{\mathcal{A}} \rangle$, is a structure of type τ, then $\mathcal{A}$ will be encoded as a binary string $\text{bin}(\mathcal{A})$ of length $I(n) = n^{a_1} + \cdots + n^{a_k} + r\lceil \log n \rceil$, consisting of one bit for each a_i-tuple, potentially in the relation R_i, and $\lceil \log n \rceil$ bits to name each constant, c_j. Thus we reserve n to indicate the size of the universe of the input structure. $I(n)$, the length of $\text{bin}(\mathcal{A})$, is polynomially related to n, and in the case where τ consists of a single unary relation – i.e., inputs are binary strings – $I(n) = n$.

Define the complexity class FO to be the set of all first-order expressible problems. We will see in §5 that FO is a uniform version of the circuit class AC^0. (See also [1], where it is shown that FO is equal to deterministic log time uniform AC^0.)

EXAMPLE 2.1. An example of a first-order expressible property is addition.[2] In order to turn addition into a yes/no question, we can let our input have the vocabulary $\tau_a = \langle A, B, k \rangle$ consisting of two unary relations and a constant symbol. In a structure $\mathcal{A}$ of vocabulary τ_a, the relations A and B are binary strings of length $n = |\mathcal{A}|$. We will say that $\mathcal{A}$ satisfies the addition property if the kth bit of the sum of A and B is one.

In order to express addition, we will first express the carry bit,

$$\text{CARRY}(x) \equiv (\exists y < x)[A(y) \wedge B(y) \wedge (\forall z.y < z < x)A(z) \vee B(z)] .$$

Then with $\oplus$ standing for exclusive or, we can express PLUS,

$$\text{PLUS}(x) \equiv A(x) \oplus B(x) \oplus \text{CARRY}(x) .$$

Thus the sentence expressing the addition property is $\text{PLUS}(k)$.

2.2. Iterating first-order sentences. To describe properties that are not in AC^0, we need languages that are more expressive than FO. We now recall the definition of the complexity classes $\text{FO}[t(n)]$[3]. Intuitively, $\text{FO}[t(n)]$ consists of those problems that may be described by a first-order sentence "iterated $t(n)$ times."

Let x be a variable and M a quantifier-free formula. We will use the notation $(\forall x.M)\psi$ – read, "for all x such that M, ψ," – to abbreviate $(\forall x)(M \rightarrow \psi)$. Similarly we will write $(\exists x.M)\psi$ – read, "there exists an x such that M, ψ," – to abbreviate $(\exists x)(M \wedge \psi)$. We will call the expressions $(\forall x.M)$ and $(\exists x.M)$ *restricted quantifiers.* Let a *quantifier block* be a finite sequence of restricted quantifiers: $\text{QB} = (Q_1 x_1.M_1) \cdots (Q_k x_k.M_k)$. We will use the notation $[\text{QB}]^t$ to denote the quantifier block QB repeated t times. I mean this literally:

$$[\text{QB}]^t = \underbrace{\text{QB}\,\text{QB}\,\text{QB} \cdots \text{QB}}_{t \text{ times}} .$$

Note that for any quantifier-free formulas $M_0, M_1, \cdots, M_k \in \mathcal{L}(\tau)$, and any $i \in \mathbf{N}$, the expression $[\text{QB}]^i M_0$ is a well-formed formula in $\mathcal{L}(\tau)$.

DEFINITION 2.2. Let $t : \mathbf{N} \rightarrow \mathbf{N}$ be any function, and let τ be any vocabulary. A set C of structures of vocabulary τ is a member of $FO[t(n)]$ if and only if there exists a quantifier block QB and a quantifier-free formula M_0 from $\mathcal{L}(\tau)$, such that if we let $\varphi_n = [\text{QB}]^{t(n)} M_0$, for $n = 1, 2, \cdots$, then for all structures G of vocabulary τ with $|G| = n$,

$$G \in C \Leftrightarrow G \models \varphi_n .$$

[2] This is a standard construction, see e.g., [17].

[3] The notation $\text{FO}[t(n)]$ was first used in [12]; however, the same classes were defined in [10] using the notation $\text{IQ}[t(n)]$, standing for "iterated queries." See [13] for a survey of descriptive complexity.

A more traditional way to iterate formulas is by making inductive definitions, [14], [10]. Let IND-DEPTH[$t(n)$] be the set of problems expressible as a uniform induction that requires depth of recursion at most $t(n)$ for structures of size n. In [7], Harel and Kozen introduce a programming language called IND, which is closely tied to inductive definitions. They prove that the execution time for their IND programs is equal to the depth of the inductive definitions that describe the programs' input output behavior. Let IND-TIME[$t(n)$] be the set of languages accepted by IND programs using $O[t(n)]$ steps for inputs of size n. Then:

FACT 2.3 ([7]). *For all $t(n)$,*

$$\text{IND-TIME}[t(n)] = \text{IND-DEPTH}[t(n)] .$$

This fact, together with Theorem 1.1, shows that there is a simple, high-level programming language for which time corresponds exactly to time on a CRAM. In the remainder of this paper we write IND[$t(n)$] to signify IND-TIME[$t(n)$] as well as IND-DEPTH[$t(n)$].

The following fact relates IND[$t(n)$] to FO[$t(n)$]. This fact follows easily from Moschovakis' Canonical Form for Positive Formulas, [14].

FACT 2.4 ([10], [14]). *For all $t(n)$,*

$$\text{IND}[t(n)] \subseteq \text{FO}[t(n)] .$$

(*In particular, a property in* IND[$t(n)$] *is expressible as a* FO[$t(n)$] *property in which* $M_0 \equiv$ *false, cf. Definition* 2.2.)

EXAMPLE 2.5. We show how to transfer a $\log n$ depth inductive definition of the transitive closure of a graph to an equivalent FO[$\log n$] definition.

Let E be the edge predicate for a graph G with n vertices. We can inductively define E^*, the reflexive, transitive closure of G, as follows:

$$E^*(x,y) \equiv x = y \vee E(x,y) \vee (\exists z)(E^*(x,z) \wedge E^*(z,y)) .$$

Let $P_n(x,y)$ mean that there is a path of length at most n from x to y. Then we can rewrite the above definition of E^* as:

$$P_n(x,y) \equiv x = y \vee E(x,y) \vee (\exists z)(P_{n/2}(x,z) \wedge P_{n/2}(z,y)) .$$

This can be rewritten:

$$P_n(x,y) \equiv (\forall z.M_1)(\exists z)(P_{n/2}(x,z) \wedge P_{n/2}(z,y)) ,$$

where $M_1 \equiv \neg(x = y \vee E(x,y))$. Note that there is no free occurrence of the variable z after the $\forall z$ quantifier. Thus, in this case $(\forall z.M_1)\alpha$ is equivalent to $(M_1 \rightarrow \alpha)$. Next,

$$P_n(x,y) \equiv (\forall z.M_1)(\exists z)(\forall uv.M_2)(P_{n/2}(u,v)) ,$$

where $M_2 \equiv (u = x \wedge v = z) \vee (u = z \wedge v = y)$. Now,

$$P_n(x,y) \equiv (\forall z.M_1)(\exists z)(\forall uv.M_2)(\forall xy.M_3)(P_{n/2}(x,y)) ,$$

where $M_3 \equiv (x = u \wedge y = v)$. Thus,

$$P_n(x,y) \equiv [\mathrm{QB}]^{\lceil \log n \rceil}(P_1(x,y)) ,$$

where $\mathrm{QB} = (\forall z.M_1)(\exists z)(\forall uv.M_2)(\forall xy.M_3)$. Note that

$$P_1(x,y) \equiv [\mathrm{QB}](\mathit{false}) .$$

It follows that

$$P_n(x,y) \equiv [\mathrm{QB}]^{\lceil 1+\log n \rceil}(\mathit{false}) ,$$

and thus $E^* \in \mathrm{FO}[\log n]$ as claimed.

2.3. Concurrent random access machines. We define the concurrent random access machine (CRAM) to be essentially the concurrent-read, concurrent-write parallel random access machine (CRCW PRAM) described in [17]. A CRAM is a synchronous parallel machine such that any number of processors may read or write into any word of global memory at any step. If several processors try to write into the same word at the same time, then the lowest-numbered processor succeeds.[4] In addition to assignments, the CRAM instruction set includes addition, subtraction, and branch on less than. Each processor also has a local register containing its processor number.

The difference between the CRAM and the CRCW PRAM described in [17] is that we also include a Shift instruction. Shift(x,y) causes the word x to be shifted y bits to the right. Without Shift, CRAM$[t(n)]$ would be too weak to simulate FO$[t(n)]$ for $t(n) < \log n$. The reason behind the Shift operation for CRAMs and the corresponding BIT predicate for first-order logic is that each bit of global memory should be available to every processor in constant time.

Let CRAM$[t(n)]$ be the set of problems accepted by a CRAM using polynomially many processors and time $O[t(n)]$. Recall that we encode an input structure $\mathcal{A} = \langle \{0,1,\cdots,n-1\}, R_1^{\mathcal{A}},\cdots,R_k^{\mathcal{A}}, c_1^{\mathcal{A}},\cdots,c_r^{\mathcal{A}} \rangle$, as the binary string $\mathrm{bin}(\mathcal{A})$ of length $I(n) = n^{a_1} + \cdots + n^{a_k} + r\lceil \log n \rceil$, Where a_i is the arity of the ith input relation. The input string is placed one bit at a time in the first $I(n)$ global memory locations.[5]

3. Proof of the main theorem. Theorem 1.1 follows immediately from three containments: Fact 2.4, and the following two lemmas.

LEMMA 3.1. *For any polynomially bounded $t(n)$ we have,*

$$\mathrm{CRAM}[t(n)] \subseteq \mathrm{IND}[t(n)] .$$

Proof. We want to simulate the computation of a CRAM M. On input $\mathcal{A}$, a structure of size n, M runs in $t(n)$ synchronous steps, using $p(n)$ processors, for some polynomial $p(n)$. Since the number of processors, the time, and the memory word size are all polynomially bounded, we need only a constant number of variables

[4] This is the "priority write" model. Our results remain true if instead we use the "common write" model, in which the program guarantees that different values will never be written to the same location at the same time. See Corollary 3.4.

[5] We show in Corollary 3.4 that if placement of the input is varied, e.g., if the first $I(n)/\log n$ words of memory contain $\log n$ bits each of the input, or even if all $I(n)$ bits are placed in the first word, then all our results remain unchanged. Note that this is not true of the models used in [2], for example. There processors are assumed to have unlimited power and thus the partition of the inputs is crucial.

$x_1, \cdots, x_k$, each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation. We can thus define the contents of all the relevant registers for any processor of M, by induction on the time step.

We now specify the CRAM model more precisely. We may assume that each processor has a finite set of registers including the following: Processor, containing the number between 1 and $p(n)$ of the processor; Address, containing an address of global memory; Contents, containing a word to be written into or read from global memory; and Program_Counter, containing the line number of the instruction to be executed next. The instructions to be simulated are limited to the following:

- READ: Read the word of global memory specified by Address into Contents.
- WRITE: Write the Contents register into the global memory location specified by Address.
- OP $R_a\, R_b$: Perform OP on R_a and R_b, leaving the result in R_b. Here OP may be Add, Subtract, or Shift.
- MOVE $R_a\, R_b$: Move R_a to R_b.
- BLT $R\, L$: Branch to line L if the contents of R is less than zero.

It is straightforward to write a first-order inductive definition for the relation VALUE$(\overline{p}, \overline{t}, \overline{x}, r, b)$ meaning that bit $\overline{x}$ in register r of processor $\overline{p}$ just after step $\overline{t}$ is equal to b. Note that since the number of processors, the time, and the word size are all polynomially bounded, a constant number of variables ranging from 0 to $n-1$ suffice to specify each of these values.

The inductive definition of the relation VALUE$(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of $\overline{p}$'s program counter at time $\overline{t}-1$. The most interesting case is when the instruction to be executed is READ. Here we simply find the most recent time $\overline{t'} < \overline{t}$ at which the word specified by $\overline{p}$'s Address register at time $\overline{t}$ was written into, and the lowest-numbered processor $\overline{p'}$ that wrote into this address at time $\overline{t'}$. In this way we can access the answer, namely, the $\overline{x}$th bit of $\overline{p'}$s Contents register at time $\overline{t'}$.

It remains to check that Addition, Subtraction, BLT, and Shift are first-order expressible, and that we can express the fact that each processor begins with its own processor number in its Processor register. Addition was done in Example 2.1. In a similar way we can express Subtraction, and Less Than. The main place we need the BIT relation is to express the fact that the initial contents of each processor's Processor register is its processor number. The relation BIT allows us to translate between variable numbers and words in memory. Using BIT we can also express addition on variable numbers and thus express the Shift operation.

Thus we have described an inductive definition of the relation VALUE, coding M's entire computation. Furthermore, one iteration of the definition occurs for each step of M. □

LEMMA 3.2. *For polynomially bounded, and parallel time constructible $t(n)$,*

$$\mathrm{FO}[t(n)] \subseteq \mathrm{CRAM}[t(n)] .$$

Proof. Let the FO$[t(n)]$ problem be determined by the following quantifier-free formulas and quantifier block,

$$M_0, M_1, \cdots, M_k, \quad \mathrm{QB} = (Q_1 x_1.M_1) \cdots (Q_k x_k.M_k) .$$

Our CRAM must test whether an input structure $\mathcal{A}$ satisfies the sentence,

$$\varphi_n \equiv [\mathrm{QB}]^{t(n)} M_0 .$$

The CRAM will use n^k processors and n^{k-1} bits of global memory. Note that each processor has a number $a_1 \cdots a_k$ with $0 \leq a_i < n$. Using the Shift operation it can retrieve each of the a_is in constant time.[6]

The CRAM will evaluate φ_n from right to left, simultaneously for all values of the variables $x_1, \cdots, x_k$. For $0 \leq r \leq t(n) \cdot k$, let

$$\varphi_n^r \equiv (Q_i x_i . M_i) \cdots (Q_k x_k . M_k)[\mathrm{QB}]^q M_0 ,$$

where $r = k \cdot (q+1) + 1 - i$. Let $x_1 \cdots \hat{x}_i \cdots x_k$ be the $k-1$-tuple resulting from $x_1 \cdots x_k$ by removing x_i. We will now give a program for the CRAM which is broken into rounds, each consisting of three processor steps such that:

(*) Just after the rth round, the contents of memory location $a_1 \cdots \hat{a}_i \cdots a_k$ is 1 or 0 according to whether $\mathcal{A} \models \varphi_n^r(a_1, \cdots, a_k)$.

Note that x_i is not free in φ_n^r! At the rth round, processor number $a_1 \cdots a_k$ executes the following three instructions according to whether $Q_i = \exists$ or $Q_i = \forall$:

$\{Q_i = \exists\}$

1. $b \leftarrow \mathrm{loc}(a_1 \cdots \hat{a}_{i+1} \cdots a_k)$;
2. $\mathrm{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 0$;
3. if $M_i(a_1, \cdots, a_k)$ and b then $\mathrm{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 1$;

$\{Q_i = \forall\}$

1. $b \leftarrow \mathrm{loc}(a_1 \cdots \hat{a}_{i+1} \cdots a_k)$;
2. $\mathrm{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 1$;
3. if $M_i(a_1, \cdots, a_k)$ and $\neg b$ then $\mathrm{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 0$;

It is not hard to prove by induction that (*) holds, and thus that the CRAM simulates the formula. □

REMARK 3.3. The proof of Lemma 3.2 provides a very simple network for simulating a FO[$t(n)$] property. The network has n^{k-1} bits of global memory and kn^k gates, where k is the number of distinct variables in the quantifier block. Each gate of the network is connected to two bits of global memory in a simple connection pattern. The blowup of processors going from CRAM to FO to CRAM seems large (cf. Corollary 4.1); however, it is plausible to build first-order networks with billions of processing elements, i.e., gates, thus accommodating fairly large n and moderately large k.

An immediate corollary of Theorem 1.1 is that the complexity class CRAM[$t(n)$] is not affected by minor changes in how the input is arranged, nor in the global memory word size, nor even by a change in the convention on how write conflicts are resolved.

COROLLARY 3.4. *For any function $t(n)$, the complexity class* CRAM[$t(n)$] *is not changed if we modify the definition of a* CRAM *in any consistent combination of the following ways. (By consistent we mean that we don't allow input words larger than the global word size, nor larger than the allowable length of applications of Shift.)*

1. *Change the input distribution so that either* (a) *the entire input is placed in the first word of global memory, or* (b) *the $I(n)$ bits of input are placed* $\log n$ *bits at a time in the first $I(n)/\log n$ words of global memory.*

2. *Change the global memory word size so that either* (a) *the global word size is one, i.e., words are single bits (local registers do not have this restriction so that*

[6] This is obvious if n is a power of 2. If not, we can just let each processor break its processor number into k $\lceil \log n \rceil$-tuples of bits. If any of these is greater than or equal to n, then the processor should do nothing during the entire computation.

the processor's number may be stored and manipulated); or (b) *the global word size is bounded by* $O[\log n]$.

3. *Modify the Shift operation so that shifts are limited to the maximum of the input word size and of the log base 2 of the number of processors.*

4. *Remove the polynomial bound on the number of memory locations, thus allowing an unbounded global memory.*

5. *Instead of the priority rule for the resolution of write conflicts, adopt the common write rule in which different processors never write different values into the same memory location at a given time step.*

Proof. The proof is that Lemmas 3.1 and 3.2 still hold with any consistent set of these modifications. This is immediate for Lemma 3.1. For Lemma 3.2, we must only show that processor number $a_1 \cdots a_k$ still has the power in constant time to evaluate the quantifier-free formula $M_i(a_1, \cdots, a_k)$, and to name the global memory location $a_1 \cdots \hat{a}_i \cdots a_k$, for $1 \leq i \leq k$. Recall that we are assuming that the input structure $\mathcal{A} = \langle \{0, 1, \cdots, n-1\}, R_1^{\mathcal{A}}, \cdots, R_p^{\mathcal{A}}, c_1^{\mathcal{A}}, \cdots, c_q^{\mathcal{A}} \rangle$ is coded as a bit string of length $I(n) = n^{r_1} + \cdots + n^{r_p} + q\lceil \log n \rceil$. It is clear that all of the consistent modifications above allow processor $a_1 \cdots a_k$ to test in constant time whether or not the relation $R(t_1, \cdots, t_r)$ holds, where R is an input or logical relation, and $t_j \in \{a_1, \cdots, a_k\} \cup \{c_j | 1 \leq j \leq q\}$. □

4. On the efficiency of the simulations. In this section we analyze the proof of Theorem 1.1 in more detail in order to give the following bounds for translating between CRAM and IND. After we prove Corollary 4.1, we discuss the cost of the simulation, and how these bounds can be improved. The proofs in this section involve counting how many variables are needed in various first-order formulas. This whole section should be omitted by the casual reader.

COROLLARY 4.1. *Let* CRAM$[t(n)]$-PROC$[p(n)]$ *be the complexity class* CRAM $[t(n)]$ *restricted to machines using at most* $O[p(n)]$ *processors. Let* IND$[t(n)]$-VAR$[v(n)]$ *be the complexity class* IND$[t(n)]$ *restricted to inductive definitions using at most* $v(n)$ *distinct variables. Assume for simplicity that the maximum size of a register word, and* $t(n)$ *are both* $o[\sqrt{n}]$, *and that* $\pi \geq 1$ *is a natural number. Then,*

$$\begin{aligned} &\text{CRAM}[t(n)]\text{-PROC}[n^{\pi}] \\ &\qquad \subseteq \quad \text{IND}[t(n)]\text{-VAR}[2\pi + 2] \\ &\qquad\qquad \subseteq \quad \text{CRAM}[t(n)]\text{-PROC}[n^{2\pi+2}]. \end{aligned}$$

Proof. We prove these bounds using the following two lemmas.

LEMMA 4.2. *If the maximum size of a register word, and* $t(n)$ *are both* $o[\sqrt{n}]$, *and if* M *is a* CRAM$[t(n)]$-PROC$[n^{\pi}]$ *machine, then the inductive definition of* VALUE *may be written using* $2\pi + 2$ *variables.*

Proof. We write out the inductive definition of VALUE in enough detail to count the number of variables used:

$$\text{VALUE}(\bar{p}, t, x, r, b) \equiv Z \vee W \vee S \vee R \vee M \vee B \vee A\,,$$

where the disjuncts have the following intuitive meanings:

Z: $t = 0$ and the initial value of r is correct.

W: $t \neq 0$ and the instruction just executed is WRITE, and the value of r is correct, i.e., unchanged unless r is Program_Counter.

S, R, M, B, A: Similarly for SHIFT, READ, MOVE, BLT, and, ADD or SUBTRACT, respectively.

It suffices to show that each disjunct can be written using the number of variables claimed. First we consider the disjunct Z. The only interesting part of Z is the case where r is "Processor". In this case we use the relation BIT to say that $b = 1$ if and only if the xth bit of $\overline{p}$ is 1. No extra variables are needed. Note that the number of free variables in the relation is $\pi + 1$ because we may combine the values t, x, r, and b into a single variable.

Next we consider the case of Addition. Recall that the main work is to express the carry bit:

$$C[A,B](x) \equiv (\exists y < x)[A(y) \wedge B(y) \wedge (\forall z.y < z < x)A(z) \vee B(z)] .$$

This definition uses two extra variables. Thus $\pi + 3 \leq 2\pi + 2$ variables certainly suffice. The cases S, M, and B are simpler.

The last, and most interesting case is R. Here we must say,

1. The instruction just executed is READ,
2. Register r is the Contents register,
3. There exists a processor $\overline{p'}$ and a time t' such that:
   (a) $t' < t$,
   (b) Address$(\overline{p'}, t')$ =Address$(\overline{p}, t)$,
   (c) VALUE$(\overline{p'}, t', x, r, b)$,
   (d) Processor $\overline{p'}$ wrote at time t',
   (e) For all $\overline{p''} < \overline{p'}$, if $\overline{p''}$ wrote at time t', then Address$(\overline{p''}, t') \neq$Address$(\overline{p'}, t')$,
   (f) For all t'' such that $t' < t'' < t$ and for all $\overline{p''}$, if $\overline{p''}$ wrote at time t'', then Address$(\overline{p''}, t'') \neq$Address$(\overline{p'}, t')$.

Let's count variables. On its face this formula uses three $\overline{p}$'s and three t's. However, two copies of each suffice. Observe that where we quantify $\overline{p''}$ in lines 3e and 3f, we no longer need $\overline{p}$, so we may use these variables instead. Similarly, when we quantify t'' on line 3f, we don't need $\overline{p''}$ so we may temporarily use one of its variables for t''. Finally, we would seem to need an extra variable to say "Address$(\overline{p''}, t'') \neq$Address$(\overline{p'}, t')$," in 3f. Here we use the fact that t is $o[\sqrt{n}]$, so t' and t'' can be coded into a single variable. Then with one more variable we can say that there exists a bit on which Address$(\overline{p''}, t'')$ and Address$(\overline{p'}, t')$ disagree. Thus $2\pi + 2$ variables suffice as claimed. □

The second lemma we need (Lemma 4.3) is a refinement of Lemma 3.2.

LEMMA 4.3. *Let $\varphi(R, \overline{x})$ be an inductive definition of depth $d(n)$. Let k be the number of distinct variables including $\overline{x}$ occurring in φ. Then the relation defined by φ is also computable in* CRAM$[d(n)]$-PROC$[O[n^k]]$.

Proof. This is very similar to the proof of Lemma 3.2. Let T be the parse tree of φ. The CRAM will have $n^k|T|$ processors: one for each value of the k variables and each node in T. Let δ be the depth of T. In rounds consisting of 3δ steps, the CRAM will evaluate an iteration of φ. Let $r =$ arity$(R) =$ the number of variables in $\overline{x}$; so $r \leq k$. The CRAM will have n^r bits of global memory to hold the truth value of $R_t = \varphi^t(\emptyset)$. It will use an additional $n^k|T|$ bits of memory to store the truth values corresponding to nodes of T. Thus $R_{d(n)}$, the least fixed point of φ, is computed in time $O[d(n)]$, using $O[n^k]$ processors, as claimed. □

This completes the proof of Corollary 4.1. □

The above proofs give us some information concerning the efficiency of our simulation of CRAMs with first-order inductive definitions. The main questions is, "Why is the number of variables needed to express a computation of n^π processors $2\pi + 2$,

instead of π?" We discuss the multiplicative factor of two, and the additional two variables, respectively in the next two paragraphs.

We need the 2π term for two reasons: we must specify $\overline{p}$ and $\overline{p'}$ at the same time in order to say that their Address registers are equal; and we need to say that no lower-numbered processor $\overline{p''}$ wrote into the same address as $\overline{p'}$. This term points out a difference between the CRAM model and the network described in Remark 3.3 that was used to simulate a FO$[t(n)]$ property. The factor of two would be eliminated if we adopted a weaker parallel machine model allowing only common writes[7], and such that the memory location accessed by a processor could be determined by a very simple computation on the processor number.

The additional two variables arise for various bookkeeping reasons. This term can be significantly reduced if we make the following two changes:

1. Rather than keeping track of all previous times, we can assume that every bit of global memory is written into at least every T time steps for some constant T.

2. The register size can be restricted to $O[\log n]$ so we need only $O[\log \log n]$ bits to name a bit of a word.

REMARK 4.4. The above observations show that the relation between the number of variables needed to give an inductive definition of a relation, and the logarithm to the base n of the number of processors needed to quickly compute the relation are nearly identical. The cost of programming with first-order inductive definitions rather than CRAMs is theoretically very small. More work and even some experimentation must be done before one can say whether or not this will turn out to be a practical approach.

5. NC versus FO. In this section we relate the uniform NC circuit classes to FO$[t(n)]$, and we derive a completely syntactic definition for circuit uniformity. We show that our definition is equivalent to the usual Turing machine-based definition in the range where the latter exists.

Let NC^i (respectively, AC^i) be the set of problems recognizable by a uniform sequence of polynomial size, bounded fan-in (respectively, unbounded fan-in) boolean circuits of depth $\log^i n$. Let $\mathrm{NC} = \mathrm{AC} = \bigcup_i \mathrm{NC}^i$. Ruzzo characterized these uniform circuit classes in terms of alternating Turing machines:

FACT 5.1 ([15]). *For $i \geq 1$,*

$$\begin{aligned} \mathrm{NC}^i &= \mathit{ASPACE\text{-}TIME}[\log n, \log^i n] , \\ \mathrm{AC}^i &= \mathit{ASPACE\text{-}ALT}[\log n, \log^i n] . \end{aligned}$$

Ruzzo and Tompa proved the following relationship between the uniform AC classes and the CRAM :

FACT 5.2 ([17]). *For $i \geq 1, \mathrm{AC}^i = \mathrm{CRAM}[\log^i n]$.*

The following corollary of Theorem 1.1 and Fact 5.2 shows that the uniformity condition for the AC^i circuit classes can be described in a syntactic way. A first-order sentence iterated $t(n)$ times is also an AC circuit "iterated" $t(n)$ times. Thus we no longer need to mention machines when discussing uniform circuit complexity.

COROLLARY 5.3. *For $i \geq 1$, $\mathrm{AC}^i = \mathrm{FO}[\log^i n]$.*

Before now there was no satisfactory definition for uniform AC^0. It is easy to see that a first-order sentence corresponds to a particularly simple sequence of AC^0

[7] See [6] for an earlier proof that a common write machine can simulate a CRAM with a linear increase in time and a squaring of the number of processors.

circuits. Each quantifier $\exists x$ (respectively, $\forall x$) is just an n-ary "or" (respectively, "and"). In [11] we showed that an appropriate way to make first-order sentences nonuniform is to add an arbitrary new logical relation. The following fact says that nonuniform AC^0 is equal to nonuniform FO.[8]

FACT 5.4 ([11]). *Given a problem S and an integer $d > 1$ the following are equivalent:*

1. *S is recognized by a sequence of depth $d+1$, polynomial-size circuits, with bounded fan-in at the bottom level.*

2. *There exists a new logical relation $R \subset \mathbf{N}^a$ and a first-order formula φ in which R occurs such that φ expresses S. The formula φ contains d alternating blocks of quantifiers.*

In view of the above results, we propose the following:

DEFINITION 5.5. Let (uniform) $AC^0 \stackrel{\text{def}}{=} FO[1] = CRAM[1]$.

Since we first made this suggestion, much evidence concerning the appropriateness of Definition 5.5 has appeared. In particular, see [1] for a study of low-level uniformity. It is shown there that FO is equal to deterministic log time uniform AC^0.

In [11] we introduced the notion of first-order translations. These reductions consist of a fixed first-order formula translating all instances of one problem to instances of another. (First-order translations are interpretations between theories, cf. [4], that are also reductions in the complexity theoretic sense.) It follows from Definition 5.5 that first-order translations are exactly uniform AC^0 reductions.

One way to evaluate the appropriateness of Definition 5.5 is to examine examples of AC^0 reductions in the literature and see whether or not they can be made uniform. Of those we have considered, the answer is yes, with the following interesting exception. (The UGAP problem is the set of undirected graphs for which there exists a path from vertex 0 to vertex $n-1$.)

FACT 5.6 ([3]). UGAP *is nonuniform* AC^0 *reducible to* UNDIR-CYCLE.

Now UNDIR-CYCLE is in DSPACE[$\log n$] [8], but UGAP is not known to be in DSPACE[$\log n$]. Of course,

REMARK 5.7. If UGAP is uniform AC^0 reducible to UNDIR-CYCLE, then UGAP is in DSPACE[$\log n$].

We mention one more interesting justification of Definition 5.5. In [3] it is shown that the obvious bounds,

$$\text{nonuniform } NC^i \subseteq \text{nonuniform } AC^i$$

can be improved to

FACT 5.8 ([3]).

$$\text{nonuniform } NC^i \subseteq \text{nonuniform AC-DEPTH}[\log^i n/\log\log n].$$

When $i = 1$ this bound is optimal because nonuniform AC-DEPTH[$\log n/\log\log n$] is necessary for Parity [18]. We next show that the same bound holds in the uniform case:

THEOREM 5.9. *For $t(n) \geq \log n$,*

$$\text{ASPACE}[\log n] - \text{TIME}[t(n)] \subseteq \text{FO}[t(n)/\log\log n].$$

[8] In [17] Stockmeyer and Vishkin showed that nonuniform AC^0 is equal to constant time on a nonuniform CRAM. This, together with Fact 5.4, gives a nonuniform version of Theorem 1.1.

Proof. This is a $\log\log n$ factor improvement of Theorem B.3 in [9]. There we showed how to code a log space Turing machine configuration using a constant number of variables, as well as how to write the predicate $M_1(\bar{x},\bar{y})$, meaning that $\langle \bar{x},\bar{y}\rangle$ is a valid move of the given alternating Turing machine. We could then inductively define the predicate $\text{Accept}_t(\bar{x})$, meaning that the configuration $\bar{x}$ leads to acceptance in the sense of alternating Turing machines in t steps:

$$\text{Accept}_t(\bar{x}) \equiv (\exists \bar{y}.M_1(\bar{x},\bar{y}))(\forall \bar{z}.M_2)\text{Accept}_{t-1}(\bar{z}) ,$$

where $M_2 \equiv (\bar{z}=\bar{y}) \vee (\text{“}\bar{x} \text{ is universal”} \wedge M_1(\bar{x},\bar{z}))$.

To improve this simulation by a $\log\log n$ factor, observe that a list of which existential moves to make in the event of each possible sequence of $(\log\log n)/2$ universal moves can be given in $\log n$ bits. Thus we can write

$$\text{Accept}_t(\bar{x}) \equiv (\exists e \forall u)(\exists \bar{z})(R \wedge \text{Accept}_{t-\log\log n}(\bar{z})) , \tag{1}$$

where R says that $\bar{z}$ follows from $\bar{x}$ in the $\log\log n$ moves determined by e and u.

Now it is easy to write an inductive definition of R whose depth is $\log\log n$. This definition uses the BIT predicate to decode from e and u which of the possible two moves the Turing machine makes at each of the $\log\log n$ steps. The simultaneous inductive definition of Accept is given in Equation 1. Obviously its depth is $\log n/\log\log n$. □

COROLLARY 5.10. *For $i \geq 1$, $\text{NC}^i \subseteq \text{FO}[\log^i n/\log\log n]$.*

6. The polynomial-time hierarchy. In second-order logic we have first-order logic, plus new relation variables over which we may quantify. Let A_i^j be a j-ary relation variable. Then $(\forall A_i^j)\varphi$ means that for all choices of j-ary relation A_i^j, φ holds. It is well known that second-order formulas may be transformed into prenex form, with all second-order quantifiers in front. Let SO be the set of second-order expressible properties, and let (SO $\exists$) be the set of second-order properties that may be written in prenex form with no universal second-order quantifiers. Fagin gave the following interesting characterization of nondeterministic polynomial-time (NP) in terms of logical expressibility:

FACT 6.1 ([5]). (SO$\exists$) = NP .

A few years later, when he defined the polynomial-time hierarchy (PH), Stockmeyer showed that it coincided with the set of second-order expressible properties:

FACT 6.2 ([16]). PH = SO.

As a corollary to Fact 6.2 and Theorem 1.1, we obtain the following characterization of PH as a parallel complexity class:

COROLLARY 6.3. PH *is equal to the set of properties checkable by a* CRAM *using exponentially many processors and constant time*[9]*:*

$$\text{PH} = \bigcup_{k=1}^{\infty} \text{CRAM[1]-PROC}[2^{n^k}] .$$

Proof. The inclusion $\text{SO} \subseteq \text{CRAM[1]-PROC}[2^{n^{O(1)}}]$ follows just as in the proof of Lemma 3.2. A processor number is now large enough to give values to all the

[9] Up to this point we had been assuming for notational simplicity that a CRAM has at most polynomially many processors. However, the class CRAM[$t(n)$]-PROC[$p(n)$] still makes sense for numbers of processors $p(n)$ that are not polynomially bounded.

relational variables as well as to all the first-order variables. Thus, as in Lemma 3.2, the CRAM can evaluate each first or second-order quantifier in three steps.

The inclusion CRAM[1]-PROC[$2^{n^{O(1)}}$] $\subseteq$ SO follows just as in the proof of Lemma 3.1. The only difference is that we use second-order variables to specify the processor number. □

7. Conclusions. To recapitulate, we have shown that parallel time has a simple mathematical definition: the minimal parallel time needed to compute a property using at most polynomially many processors is equal to the minimum depth of a first-order inductive definition of the property. Furthermore, the number of processors needed by the CRAM is closely tied to the number of variables needed in the inductive definition. We have also given purely syntactic definitions for uniformity of the circuit complexity classes AC^i, $i \geq 0$. Finally, we have given a striking, new characterization of the polynomial-time hierarchy. We believe that these results help to explain the nature of parallel complexity and will lead to an improved understanding of the subject.

There is much work to be done. The following general directions suggest themselves:

1. This paper provides a new way to think about parallel programming. The programmer provides efficient inductive definitions of the problem to be solved. Our simulation results then automatically give an efficient implementation on a CRAM. Much work is needed to explore whether or not this approach is practical.

2. We have given characterizations of parallel time and number of processors in terms of the depth and number of variables in inductive definitions. One should now develop upper and lower bounds on these parameters for all sorts of problems. We also feel that the analysis of the simulation in §4 can and should be improved.

3. There are many fascinating questions concerning uniformity and the power of precomputation. We hope that the notion of syntactic uniformity of circuits will help researchers determine when precomputation/nonuniformity can help; or, to prove lower bounds on what can be done by uniform circuits and formulas.

Acknowledgments. Thanks to Steve Cook, Steven Lindell, Ruben Michel, and Larry Ruzzo, who contributed comments and corrections to previous drafts of this paper.

REFERENCES

[1] D. M. BARRINGTON, N. IMMERMAN, AND H. STRAUBING, *On uniformity within* NC^1, Proc. 3rd Annual Symposium on Structure in Complexity Theory (1988), pp. 47-59.

[2] P. BEAME, *Limits on the power of concurrent-write parallel machines,* Proc. 18th ACM Symposium on Theory of Computing (1986), pp. 169-176.

[3] A. CHANDRA, L. STOCKMEYER, AND U. VISHKIN, *Constant depth reducibility,* SIAM J. Comput. 13 (1984), pp. 423-439.

[4] H. ENDERTON, *A Mathematical Introduction to Logic,* Academic Press, New York, 1972.

[5] R. FAGIN, *Generalized first-order spectra and polynomial-time recognizable sets,* in Complexity of Computation, R. Karp, ed., SIAM-AMS Proc., 7 (1974), pp. 27-41.

[6] F. FICH, P. RAGDE, AND A. WIGDERSON, *Relations between concurrent-write models of parallel computation,* Proc. 3rd Annual ACM Symposium on Principles of Distributed Computing (1984), pp. 179-189.

[7] D. HAREL AND D. KOZEN, *A programming language for the inductive sets, and applications,* Proc. 9th International Colloquium on Automata Languages, and Programming, Springer-Verlag Lecture Notes in Computer Science, 140 (1982), pp. 313-329.

[8] J.-W. HONG, *On some deterministic space complexity problems,* SIAM J. Comput., 11 (1982), pp. 591-601.
[9] N. IMMERMAN, *Upper and lower bounds for first-order expressibility,* J. Comput. System. Sci., 25 (1982), pp. 76-98.
[10] ———, *Relational queries computable in polynomial time,* Inform. and Control, 68 (1986), pp. 86-104. A preliminary version of this paper appeared in Proc. 14th ACM Symposium on Theory of Computing (1982), pp. 147-152.
[11] ———, *Languages that capture complexity classes,* SIAM J. Comput., 16 (1987), pp. 760-778. A preliminary version of this paper appeared in Proc. 15th ACM Symposium on Theory of Computing (1983),pp. 347-354.
[12] ———, *Expressibility as a complexity measure: Results and directions,* Proc. 2nd Annual Symposium on Structure in Complexity Theory (1987), pp. 194-202.
[13] ———, *Descriptive and computational complexity,* Proc. 1988 AMS Short Course in Computational Complexity Theory, to appear.
[14] Y. N. MOSCHOVAKIS, *Elementary Induction on Abstract Structures,* North Holland, Amsterdam, 1974.
[15] L. RUZZO, *On uniform circuit complexity,* J. Comput. System. Sci., 21 (1981), pp. 365-383.
[16] L. STOCKMEYER, *The polynomial-time hierarchy,* Theoretical Comput. Sci., 3 (1977), pp. 1-22.
[17] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits,* SIAM J. of Comput. 13 (1984), pp. 409-422.
[18] A. C.-C. YAO, *Separating the polynomial-time hierarchy by oracles,* Proc. 26th Annual IEEE Symposium on Foundations of Comput. Sci. (1985), pp. 1-10.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 639-657, August 1989

MATRIX PADÉ FRACTIONS AND THEIR COMPUTATION*

GEORGE LABAHN† AND STAN CABAY‡

Abstract. For matrix power series with coefficients over a field, the notion of a matrix power series remainder sequence and its corresponding cofactor sequence are introduced and developed. An algorithm for constructing these sequences is presented.

It is shown that the cofactor sequence yields directly a sequence of Padé fractions for a matrix power series represented as a quotient $B(z)^{-1}A(z)$. When $B(z)^{-1}A(z)$ is normal, the complexity of the algorithm for computing a Padé fraction of type (m, n) is $O(p^3(m+n)^2)$, where p is the order of the matrices $A(z)$ and $B(z)$.

For a power series that are abnormal for a given (m, n), Padé fractions may not exist. However, it is shown that a generalized notion of Padé fraction, the Padé form, which is introduced in this paper, does always exist and can be computed by the algorithm. In the abnormal case, the algorithm can reach a complexity of $O(p^3(m+n)^3)$, depending on the nature of the abnormalities. In the special case of a scalar power series, however, the algorithm complexity is $O((m+n)^2)$, even in the abnormal case.

Key words. matrix Padé fraction, matrix power series, matrix Padé form

AMS(MOS) subject classifications. 41A21, 41A63, 68Q40

1. Introduction.

Let

$$A(z) = \sum_{i=0}^{\infty} a_i z^i, \tag{1.1}$$

where a_i, $i = 0, \cdots$, is a $p \times p$ matrix with coefficients from a field K, be a formal power series. Loosely speaking, a matrix Padé approximant of $A(z)$ is an expression of the form $U(z) \cdot V(z)^{-1}$, or $V(z)^{-1} \cdot U(z)$, where $U(z)$ and $V(z)$ are matrix polynomials of degree at most m and n, respectively, whose expansion agrees with $A(z)$ up to and including the term z^{m+n}.

The definition of a Padé approximant can be made more formal in a variety of ways. For example, Rissanen [17] restricts $V(z)$ to be a scalar polynomial and allows $U(z)$ to be a $p \times q$ matrix. Typically, however, $U(z)$ and $V(z)$ are $p \times p$ polynomial matrices, and $V(z)$ is further restricted by the condition that the constant term, $V(0)$, is invertible (cf., Bose and Basu [2], Bultheel [5], and Starkand [19]). In this paper, we call such approximants matrix Padé fractions, which is consistent with the scalar $(p = 1)$ case (cf., Gragg [12]).

For a particular m and n, however, matrix Padé fractions need not exist. Therefore, in this paper, we introduce the notion of a matrix Padé form, in which the condition of invertibility of $V(0)$ is relaxed. The definition is a generalization of a similar one given for the scalar case (cf., Gragg [12]). It is shown that matrix Padé forms always exist, but that they may not be unique. In general, matrix Padé forms need not have an invertible denominator, $V(z)$. However, for m and n given, by obtaining a basis for all the Padé forms, we are also able to construct a matrix Padé form with an invertible denominator, $V(z)$, in the case that one does exist.

* Received by the editors April 27, 1987; accepted for publication (in revised form) October 18, 1988.
† Department of Computing Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
‡ Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. The work of this author was partially supported by the Natural Science and Engineering Research Council Grant A8035.

Padé approximants have many applications in mathematics and in engineering-related disciplines. Applications include numerical computations for special power series such as the Gamma function (cf., Nemeth and Zimanyi [15]); algorithms in the field of numerical analysis (cf., Gragg [12]); triangulation of block Hankel and Toeplitz matrices (cf., Rissanen [18]); solving linear systems of equations with Hankel of Toeplitz coefficient matrices (cf., Rissanen [16]); in digital filtering theory (cf., Bultheel [7] and Brophy and Salazar [4]; and also in linear control theory (cf., Elgerd [11]).

In the one-dimensional case, examples of algorithms that calculate Padé approximants for normal power series (Gragg [12]) include the ε-algorithm of Wynn [21]; the Levinson–Durbin algorithm [10], [14]; and the algorithm of Trench [20]. Examples of algorithms that are successful in the degenerate nonnormal case include those given by Brent, Gustavson, and Yun [3]; Bultheel [6]; Cabay and Choi [8]; and Rissanen [16].

The matrix case parallels the scalar situation in that most algorithms are restricted to normal power series. Algorithms that require the normality condition include those of Bultheel [5], Bose and Basu [2], Starkand [19], and Rissanen [18]. An algorithm that calculates Padé approximants in a nonnormal case is given by Labahn [13]. However, in his algorithm there are still strict conditions that need to be satisfied by the power series before Padé approximants can be calculated.

The primary contribution of this paper is an algorithm, MPADE, for computing matrix Padé forms for a matrix power series. Central to the development of MPADE are the notions of a matrix power series remainder sequence and the corresponding cofactor sequence, which are introduced in § 4. These are generalizations of notions developed by Cabay and Kossowski [9] for power series over an integral domain. The cofactor sequence computed by MPADE yields a sequence of matrix Padé fractions along a specific off-diagonal path of the Padé table for $A(z)$.

Unlike other algorithms, there are no restrictions placed on the power series in order that MPADE succeed. For normal power series, the complexity of MPADE is $O(p^3 \cdot (m+n)^2)$ operations in K. This is the same complexity as some of the algorithms proposed by Bultheel [5], Bose and Basu [2], Starkand [19], and Rissanen [18]. In the abnormal case, the complexity of the algorithm can reach $O(p^3 \cdot (m+n)^3)$ operations in K, depending on the nature of the abnormalities.

2. Matrix Padé forms. Let $A(z)$ and $B(z)$ be formal power series

$$A(z) = \sum_{i=0}^{\infty} a_i z^i, \qquad B(z) = \sum_{i=0}^{\infty} b_i z^i \tag{2.1}$$

with coefficients from the ring of $p \times p$ matrices over some field K. Throughout this paper it is assumed that the leading coefficient, b_0, of $B(z)$ is an invertible matrix. For nonnegative integers m and n, let

$$U(z) = \sum_{i=0}^{m} u_i z^i, \qquad V(z) = \sum_{i=0}^{n} v_i z^i \tag{2.2}$$

denote $p \times p$ matrix polynomials.

DEFINITION 2.1. The pair of matrix polynomials $(U(z), V(z))$ is defined to be a **right matrix Padé form** (RMPFo) of type (m, n) for the pair $(A(z), B(z))$ if

I. $\partial(U(z)) \leqq m, \partial(V(z)) \leqq n$,†

(2.3) II. $A(z) \cdot V(z) + B(z) \cdot U(z) = z^{m+n+1} W(z)$, where $W(z)$ is a formal matrix power series, and

III. The columns of $V(z)$ are linearly independent over the field K. □

The matrix polynomials $U(z)$, $V(z)$, and $W(z)$ are usually called the right numerator, denominator, and residual (all of type (m, n)), respectively.

There is an equivalent definition for a **left matrix Padé form** (LMPFo). Condition II is replaced with an equivalent version with matrix multiplication by $U(z)$ and $V(z)$ being on the left. Condition III is replaced with the condition that the rows, rather than the columns, of the denominator are linearly independent over the base field K.

However, there is a one-to-one correspondence between RMPFo's and LMPFo's. By taking the transposes of the matrices on both sides of (2.3), it follows that

$$V^t(z) \cdot A^t(z) + U^t(z) \cdot B^t(z) = z^{m+n+1} W^t(z). \tag{2.4}$$

The degree and order conditions are identical. It is clear that if $(U(z), V(z))$ is a RMPFo for $(A(z), B(z))$, then $(U^t(z), V^t(z))$ is a LMPFo for $(A^t(z), B^t(z))$. Thus, any algorithm that calculates a right matrix Padé form of a certain type can also be used to calculate the left matrix Padé form of the same type.

For ease of discussion, we use the following notation. For any matrix polynomial

$$U(z) = u_0 + u_1 z + \cdots + u_k z^k, \tag{2.5}$$

we write U (i.e., the same symbol but without the z variable) to mean the $p(k+1)$ by p vector of matrix coefficients

$$U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_k \end{bmatrix} \tag{2.6}$$

or, equivalently, $U = [u_0^t, u_1^t, \cdots, u_k^t]^t$.

Let

$$S_{m,n} = \begin{bmatrix} a_0 & & & b_0 & & \\ & \ddots & & & \ddots & \\ \vdots & & a_0 & \vdots & & b_0 \\ & & \vdots & & & \vdots \\ a_{m+n} & \cdots & a_m & b_{m+n} & \cdots & b_n \end{bmatrix} \tag{2.7}$$

denote a Sylvester matrix for $A(z)$ and $B(z)$ of type (m, n). Then (2.3) can be written as

$$S_{m,n} \cdot \begin{bmatrix} V \\ U \end{bmatrix} = 0. \tag{2.8}$$

THEOREM 2.2 (Existence of matrix Padé forms). *For any pair of power series* $(A(z), B(z))$ *and any pair of nonzero integers* (m, n), *there exists a* RMPFo *of type* (m, n).

† $\partial(\ )$ denotes the degree of a matrix polynomial. This is the power of the largest nonzero coefficient of the polynomial.

Proof. Let X denote a vector of length $p(m+n+2)$, and consider the homogeneous system of linear equations

$$S_{m,n} \cdot X = 0. \tag{2.9}$$

Because $S_{m,n}$ has $p(m+n+1)$ rows, it follows that (2.9) has at least p linearly independent solutions. Let $[V^t, U^t]^t$ denote p such solutions arranged by columns. Then $[V^t, U^t]^t$ satisfies (2.8); consequently $U(z)$ and $V(z)$, determined according to the convention (2.5) and (2.6), satisfy (2.3). Clearly, the pair $(U(z), V(z))$ also satisfies conditions I in Definition 2.1. Finally, b_0 being nonsingular implies that the linear independence of the columns of $[V^t, U^t]^t$ is equivalent to the linear independence of the columns of $V(z)$. Thus condition III is satisfied. □

From the proof of Theorem 2.2, it follows that if $S_{m,n}$ has maximal rank, then Padé forms are unique up to multiplication of $U(z)$ and $V(z)$ on the right by a nonsingular matrix. On the other hand, if the rank of $S_{m,n}$ is less than maximal, then more than one independent Padé form exists.

Example 2.3. Let $B(z) = -I$ and

$$A(z) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} z^2 + \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} z^4 + \begin{bmatrix} -1 & 0 \\ -1 & 0 \end{bmatrix} z^5 + \cdots. \tag{2.10}$$

With $m = 2$ and $n = 3$, a basis for the solution space of (2.9) is given by the two vectors

$$X_1 = [0, 1, 0, 0, 0, -1, 0, 0, 0, 1, 0, 0, 0, 0]^t \tag{2.11}$$

and

$$X_2 = [0, 0, 0, 1, 0, 0, 0, -1, 0, 0, 0, 1, 0, 0]^t. \tag{2.12}$$

Thus,

$$V = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{bmatrix}^t \tag{2.13}$$

and

$$U = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}^t \tag{2.14}$$

is a solution of (2.8), and the pair $(U(z), V(z))$, where

$$V(z) = \begin{bmatrix} 0 & 0 \\ 1-z^2 & z-z^3 \end{bmatrix} \quad \text{and} \quad U(z) = \begin{bmatrix} 0 & 0 \\ 1 & z \end{bmatrix} \tag{2.15}$$

is a Padé form of type (2.3) for $(A(z), B(z))$. □

In Example 2.3, note that the columns of $V(z)$ are linearly independent over the field K, but that they are linearly dependent over the ring of polynomials $K[z]$ (i.e., $V(z)$ is singular). Indeed, for this example, a RMPFo $(U(z), V(z))$ of type $(2, 3)$, for which $V(z)$ is nonsingular, cannot be found. The problem occurs because, although the solution space has dimension 2 when considered as a vector space over the field K, it has only dimension 1 when considered as a module over the ring $K[z]$.

We note that having an invertible denominator is highly desirable, since often the purpose of Padé forms is to approximate the infinite power series

$$-(B(z))^{-1} \cdot A(z) \tag{2.16}$$

by the finite rational form

$$U(z) \cdot (V(z))^{-1}, \tag{2.17}$$

where the approximation is to be exact for the first $m+n+1$ terms. When the denominator is singular, we cannot form this rational expression and this limits the usefulness of Padé approximation. For example, a singular denominator gives no information from the poles since every point is a pole in this case.

3. Matrix Padé fractions. One case when the denominator of a RMPFo is invertible is given by

DEFINITION 3.1. A pair $(U(z), V(z))$ of $p \times p$ matrix polynomials is said to be a **right matrix Padé fraction** (RMPFr) of type (m, n) for the pair $(A(z), B(z))$ if

I. $(U(z), V(z))$ is a RMPFo of type (m, n) for $(A(z), B(z))$, and

II. The constant term, $V(0)$, of the denominator is an invertible matrix. □

Condition II ensures that the denominator, $V(z)$ is an invertible matrix polynomial.

As in the case of Padé forms, there is an equivalent definition for a **left matrix Padé fraction** (LMPFr). Also, there is a correspondence between RMPFr for $(A(z), B(z))$ and LMPFr for $(A(z)', B(z)')$. It is interesting to note that a power series may have a matrix Padé fraction on one side but not on the other. In Example 2.3, the power series $A(z)$ does not have a right matrix Padé fraction of type $(2, 3)$, but it does have a left matrix Padé fraction of type $(2, 3)$. When a power series does have both a right and a left matrix Padé fraction of the same type, then the two resulting rational forms are equal (cf., Baker [1]).

The problem with Padé fractions, as mentioned in the previous section, is that they do not always exist. However, let

$$T_{m,n} = \left[\begin{array}{ccc|ccc} a_0 & & & b_0 & & \\ & \ddots & & & \ddots & \\ \vdots & & a_0 & \vdots & & b_0 \\ & & \vdots & & & \vdots \\ a_{m+n-1} & \cdots & a_m & b_{m+n-1} & \cdots & b_n \end{array}\right] \tag{3.1}$$

and define

$$d_{m,n} = \begin{cases} 1, & m = 0, \quad n = 0, \\ \det(T_{m,n}), & \text{otherwise.} \end{cases} \tag{3.2}$$

Then, a sufficient condition for the existence of a RMPFr is given by the Theorem 3.2.

THEOREM 3.2. *If $d_{m,n} \neq 0$, then every* RMPFo *of type (m, n) is an* RMPFr *of type (m, n). In addition, a* RMPFr *of type (m, n) is unique up to multiplication on the right by a nonsingular $p \times p$ matrix having coefficients from the field K.*

Proof. Equation (2.8) may be written as follows:

$$\left[\begin{array}{ccc|ccc} 0 & & & b_0 & & \\ a_0 & & & & \ddots & \\ & \ddots & & & & \\ \vdots & & a_0 & \vdots & & \\ & & \vdots & & & b_0 \\ a_{m+n-1} & \cdots & a_m & b_{b+n} & \cdots & b_n \end{array}\right] \cdot \left[\begin{array}{c} v_1 \\ \\ \\ v_n \\ u_0 \\ \\ u_m \end{array}\right] = - \left[\begin{array}{c} a_0 \\ \\ \\ \vdots \\ \\ \\ a_{m+n} \end{array}\right] \cdot v_0. \tag{3.3}$$

The matrix on the left of (3.3) is nonsingular, since $d_{m,n} \neq 0$ and b_0 is nonsingular. Thus, all the solutions of (3.3) can be obtained by assigning v_0 arbitrarily and solving (3.3) for the remaining components $v_1, \cdots, v_n, u_0, \cdots, u_m$. If v_0 is chosen to be a

singular matrix, then the solution obtained by solving (3.3) violates condition III in the definition of Padé form. Thus, in this case, all RMPFo's are RMPFr's.

To show uniqueness, suppose $(U(z), V(z))$ and $(U'(z), V'(z))$ are two RMPFr's for $(A(z), B(z))$. Then, v_0 and v_0' are both nonsingular matrices with coefficients from the field K. Thus, there exists a nonsingular matrix M with coefficients from K satisfying

$$v_0 = v_0' \cdot M. \tag{3.4}$$

It follows from (3.3) that

$$V(z) = V'(z) \cdot M \quad \text{and} \quad U(z) = U'(z) \cdot M, \tag{3.5}$$

and so uniqueness holds. □

In the next section we also require the following theorem.

THEOREM 3.3. *Let $A(z)$ and $B(z)$ be given by* (2.1). *If m and n are positive integers such that $d_{m,n} \neq 0$, then* RMPFo's $(P(z), Q(z))$ *of type* $(m-1, n-1)$ *for* $(A(z), B(z))$ *are unique up to multiplication of $P(z)$ and $Q(z)$ on the right by a nonsingular matrix from K. In addition, the leading term $R(0)$ of the residual in condition* II *for* RMPFo's,

$$A(z) \cdot Q(z) + B(z) \cdot P(z) = z^{m+n-1} R(z), \tag{3.6}$$

is a nonsingular matrix.

Proof. $S_{(m-1),(n-1)}$ can be obtained from $T_{m,n}$ by deleting the last block row (i.e., the last p rows). Since $T_{m,n}$ is of maximal rank $p(m+n)$, then $S_{(m-1),(n-1)}$ has rank $p(m+n-1)$. Consequently, the dimension of the solution space to

$$S_{(m-1),(n-1)} \cdot X = 0 \tag{3.7}$$

is exactly p. Then, $[Q', P']'$ is obtained by collecting by columns a basis for the solution space of (3.7). Clearly, if $[Q'', P'']'$ and $[Q', P']'$ are two such collections, then there exists a nonsingular matrix M from K such that

$$[Q', P']' = [Q'', P'']' \cdot M. \tag{3.8}$$

Thus, $P(z) = P'(z) \cdot M$ and $Q(z) = Q'(z) \cdot M$, proving uniqueness.

To prove the invertibility of $R(0)$ in (3.6), let $r_0 = R(0)$ and suppose that r_0 is a singular $p \times p$ matrix. Then, there is a nonzero $p \times 1$ vector X that satisfies

$$r_0 \cdot X = 0. \tag{3.9}$$

But, from (3.6) and (3.7), it follows that

$$T_{m,n} \cdot \begin{bmatrix} Q \\ P \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ r_0 \end{bmatrix}. \tag{3.10}$$

Thus,

$$T_{m,n} \cdot \begin{bmatrix} Q \\ P \end{bmatrix} \cdot X = 0. \tag{3.11}$$

Since the coefficient matrix for the above system is invertible, we deduce that

$$\begin{bmatrix} Q \\ P \end{bmatrix} \cdot X = 0. \tag{3.12}$$

But this contradicts the fact that the columns of $[Q', P']'$ are made up of linearly independent vectors. This implies that r_0 is invertible. □

The fact that Padé forms of type (m, n) and $(m-1, n-1)$ are uniquely determined after suitable normalizations, when $T_{m,n}$ is nonsingular, allows us to prove such properties as argument invariance (cf., Baker [1]) for the Padé forms computed by the algorithm MPADE given in § 5.

4. Matrix power series remainder sequences. We define a **right matrix Padé table** for $(A(z), B(z))$ to be any infinite two-dimensional collection of RMPFo's of type (m, n) for $(A(z), B(z))$ with $m = 0, 1, \cdots$ and $n = 0, 1, \cdots$. It is assumed that there is precisely one entry (i.e., one RMPFo) assigned to each position in the table. From Theorem 2.2, it follows that a right matrix Padé table exists for any given $(A(z), B(z))$. However, the table is not unique, because RMPFo's are not unique. This is unlike the definition of a Padé table for scalar power series (cf., Gragg [12]), since here a Padé table consists of a collection of Padé fractions, which are unique.

A matrix power series pair $(A(z), B(z))$ is said to be **normal** (cf., Bultheel [5]) if $d_{m,n} \neq 0$ for all m, n. For normal power series, it follows from Theorem 3.2 that every entry in the right matrix Padé table is a RMPFr. Consequently, from condition II in Definition 3.1 of RMPFr's a right-matrix Padé table for normal power series may be made unique by insisting that the constant term $V(0)$ in the denominator of any Padé fraction be the identity matrix.

Following the convention used in the scalar case (cf., Gragg [12]), we also define

$$(U(z), V(z)) = (z^m I, 0) \quad \text{for } m \geqq -1,\ n = -1, \tag{4.1}$$

and

$$(U(z), V(z)) = (0, z^n I) \quad \text{for } m = -1,\ n \geqq 0. \tag{4.2}$$

A right matrix Padé table appended with (4.1) and (4.2) is called an **extended right matrix Padé table**. The use of an extended table is strictly for initialization purposes. The entries given by (4.1) and (4.2) are not right matrix Padé forms (indeed, the $(-1, -1)$ entry is not even a matrix polynomial). However, they do satisfy property II of Definition 2.1. For example, for $m \geqq -1$ and $n = -1$, we have that

$$A(z)V(z) + B(z)U(z) = z^{m+n+1} W(z) \tag{4.3}$$

with

$$W(z) = B(z); \tag{4.4}$$

whereas, for $m = -1$ and $n \geqq 0$, we obtain (4.3) with

$$W(z) = A(z). \tag{4.5}$$

Given the power series (2.1) and any nonnegative integers m and n, we introduce a sequence of points

$$(m_0, n_0), (m_1, n_1), (m_2, n_2), \cdots \tag{4.6}$$

in the extended right matrix Padé table by setting

$$(m_0, n_0) = \begin{cases} (m-n-1, -1) & \text{for } m \geqq n \\ (-1, n-m-1) & \text{for } m < n \end{cases} \tag{4.7}$$

and

$$(m_{i+1}, n_{i+1}) = (m_i + s_i, n_i + s_i),\ i = 0, 1, 2, \cdots, \tag{4.8}$$

where $s_i \geqq 1$. Observe that

$$m_i - n_i = m - n, \qquad i = 0, 1, 2, \cdots, \tag{4.9}$$

and consequently the sequence (4.6) lies along the $m-n$ off-diagonal path of the extended right matrix Padé table. In (4.8), the s_i are selected so that

$$d_{m_{i+1},n_{i+1}} \neq 0 \tag{4.10}$$

and

$$d_{(m_i+j),(n_i+j)} = 0, \tag{4.11}$$

for $j = 1, 2, \cdots, s_i - 1$.

For $i = 1, 2, \cdots$, let $(U_i(z), V_i(z))$ be the unique RMPFr (cf., Theorem 3.2) of type (m_i, n_i) for $(A(z), B(z))$. Thus $[V_i^t, U_i^t]^t$ satisfies

$$S_{m_i,n_i} \cdot \begin{bmatrix} V_i \\ U_i \end{bmatrix} = 0, \tag{4.12}$$

and, according to (2.3), there exists a matrix power series $W_i(z)$ such that

$$A(z) \cdot V_i(z) + B(z) \cdot U_i(z) = z^{m_i+n_i+1} W_i(z). \tag{4.13}$$

Generalizing the notions of Cabay and Kossowski [9], we introduce the following definition.

DEFINITION 4.1. The sequence

$$\{W_i(z)\}, \qquad i = 1, 2, \cdots, \tag{4.14}$$

is called the **power series remainder sequence** for the pair $(A(z), B(z))$. The sequence of pairs

$$\{(U_i(z), V_i(z))\}, \qquad i = 1, 2, \cdots, \tag{4.15}$$

is called the corresponding **cofactor sequence**. The integer pairs $\{(m_i, n_i)\}$ are called **nonsingular nodes** along the $m-n$ off-diagonal path of the extended right matrix Padé table for $(A(z), B(z))$. □

We note that each term of a power series remainder sequence is unique up to multiplication on the right by a nonsingular matrix. This is also true for each term of the corresponding cofactor sequence.

Initially, when $m \geqq n$, observe that $m_1 = m - n$ and $n_1 = 0$ (i.e., $s_0 = 1$), because in (3.2) the nonsingularity of b_0 implies that $d_{(m-n),0} \neq 0$. Thus, $V_1(z)$ is some arbitrary nonsingular matrix from K and, using (4.12), $U_1(z)$ can be obtained by solving

$$\begin{bmatrix} b_0 & & \\ \vdots & \ddots & \\ b_{m_1} & \cdots & b_0 \end{bmatrix} U_1 = -\begin{bmatrix} a_0 \\ \vdots \\ a_{m_1} \end{bmatrix} V_1. \tag{4.16}$$

That is, $U_1(z)$ can be obtained by multiplying the first $m_1 + 1$ terms of the quotient power series $B^{-1}(z) \cdot A(z)$ on the right by $-V_1(z)$.

Initially, when $m < n$, depending on a_0 there are two cases to consider. The simple case, when $\det(a_0) \neq 0$, yields

$$d_{0,(n-m)} = \det \begin{bmatrix} a_0 & & \\ \vdots & \ddots & \\ a_{n-m-1} & \cdots & a_0 \end{bmatrix} \neq 0. \tag{4.17}$$

Thus, $s_0 = 1$, $m_1 = 0$, and $n_1 = n - m$. Then, the RMPFr $(U_1(z), V_1(z))$ of type (m_1, n_1) is determined by setting $U_1(z)$ to be an arbitrary nonsingular matrix from K and then solving

$$\begin{bmatrix} a_0 & & \\ \vdots & \ddots & \\ a_{n_1} & \cdots & a_0 \end{bmatrix} V_1 = -\begin{bmatrix} b_0 \\ \vdots \\ b_{n_1} \end{bmatrix} U_1. \tag{4.18}$$

That is, when $m < n$ and $\det(a_0) \neq 0$, $V_1(z)$ can be obtained from the first $n_1 + 1$ terms of the quotient power series $A^{-1}(z) \cdot B(z)$ multiplied on the right by $-U_1(z)$.

When $m < n$ and $\det(a_0) = 0$, we must first determine the smallest positive integer s_0 (i.e., the smallest $m_1 = m_0 + s_0$ and $n_1 = n_0 + s_0$) so that $d_{m_1,n_1} \neq 0$. Notice that here $s_0 > 1$. Once s_0 has been obtained, then $(U_1(z), V_1(z))$ is obtained by solving

$$S_{m_1,n_1} \cdot \begin{bmatrix} V_1 \\ U_1 \end{bmatrix} = 0. \tag{4.19}$$

In § 5, we give an algorithm which computes a RMPFo of type (m, n) for $(A(z), B(z))$ by performing a sequence of the above types of initializations (albeit, each for different power series).

When the power series pair $(A(z), B(z))$ is normal, only the initializations corresponding to (4.16) and (4.18) are required. Thus, for normal power series $s_i = 1$ for all $i \geqq 1$, and the algorithm reduces to a sequence of truncated power series divisions.

There are also some nonnormal power series that share this property. For each pair of integers m and n, let $r_{m,n}$ be the rank of the matrix $T_{m,n}$. Then normality is equivalent to

$$r_{m,n} = (m + n) \cdot p \tag{4.20}$$

for all m and n. A matrix power series pair $(A(z), B(z))$ is said to be **nearly normal** (cf., Labahn [13]) if, for all integers m and n,

$$r_{m,n} = k_{m,n} \cdot p \tag{4.21}$$

for some integer $k_{m,n}$. Clearly, every normal power series is also a nearly normal power series. In addition, all scalar power series are nearly normal.

For a nearly normal power series pair $(A(z), B(z))$ it is easy to see that when a_0 is singular, then $a_0 = 0$. This follows from the observation that the rank of a_0 is just $r_{0,1}$, which, if it is not p, must be zero. Also, if $a_0 = \cdots = a_{k-1} = 0$ and $a_k \neq 0$, then a_k must be a nonsingular matrix for similar reasons. When $k > m$ this implies that there are no nonsingular nodes along the $m - n$ off-diagonal path before and including the node (m, n). Otherwise, when $k \leqq m$, the initialization (4.19) becomes

$$\left[\begin{array}{ccc|ccc} 0 & & & b_0 & & \\ a_{m_1} & & & \vdots & \ddots & \\ \vdots & \ddots & & \vdots & & b_0 \\ a_{m_1+n_1} & \cdots & a_{m_1} & b_{m_1+n_1} & \cdots & b_{n_1} \end{array}\right] \cdot \begin{bmatrix} V_1 \\ U_1 \end{bmatrix} = 0, \tag{4.22}$$

where $s_0 = k + 1$, $m_1 = k$, and $n_1 = n - m + k$. Consequently the RMPFr $(U_1(z), V_1(z))$ of type (m_1, n_1) is obtained from (4.22) first by setting $U_1(z) = z^{m_1} \cdot U$, where U is any nonsingular matrix from K. Then, $V_1(z)$ is obtained by multiplying the first $n_1 + 1$ terms of the quotient power series $(z^{-m_1} \cdot A(z))^{-1} \cdot B(z)$ on the right by $-U$. Thus, also for

nearly normal power series (and therefore also for all scalar power series), all initializations reduce to truncated power series divisions.

Corresponding to the power series remainder sequence, we introduce Definition 4.2.

DEFINITION 4.2. The sequence

$$\{(P_i(z), Q_i(z))\}, \qquad i = 1, 2, \cdots, \tag{4.23}$$

where $(P_i(z), Q_i(z))$ is the $(m_i - 1, n_i - 1)$ entry in the extended matrix Padé table for $(A(z), B(z))$, is called a **predecessor sequence** of the power series remainder sequence. □

The pair $(P_i(z), Q_i(z))$ satisfies the equation

$$A(z) \cdot Q_i(z) + B(z) \cdot P_i(z) = z^{m_i + n_i - 1} \cdot R_i(z). \tag{4.24}$$

THEOREM 4.3. *For $i = 1, 2, \cdots$, the predecessors $(P_i(z), Q_i(z))$ are unique up to right multiplication by a nonsingular matrix from K. In addition, the leading term of the residual, $R_i(0)$, is nonsingular.*

Proof. For $m_i > 0$ and $n_i > 0$, the predecessors are right matrix Padé forms and the result is a direct consequence of Theorem 3.3. Thus, it remains only to show that the result holds when either m_1 or n_1 is zero.

When $n_1 = 0$, from (4.7) and (4.8) $m \geqq n$, and the predecessor node is the $(m - n - 1, -1)$ entry of the extended right matrix Padé table. By (4.1), this entry is given uniquely by

$$(P_1(z), Q_1(z)) = (z^{m-n-1} I, 0). \tag{4.25}$$

From (4.4), the residual $R_1(z)$ is $B(z)$ and the theorem therefore holds for $n_1 = 0$, since $\det(b_0) \neq 0$.

When $m_1 = 0$, then from (4.7) and (4.8) $n_1 = n - m > 0$, and the predecessor node is the $(-1, n - m - 1)$ entry of the extended table. By (4.2), this is uniquely given by

$$(P_1(z), Q_1(z)) = (0, z^{n-m-1} I). \tag{4.26}$$

From (4.5) the residual of this node, $R_1(z)$, is $A(z)$. But by (4.17), $(0, n - m)$ is the first nonsingular node if and only if $\det(a_0) \neq 0$. Hence the leading term of the residual is nonsingular. □

The main result of this section is given in Theorem 4.4.

THEOREM 4.4. *For any positive integer k, $(k - 1, k)$ is a nonsingular node in the Padé table for $(W_i(z), R_i(z))$ if and only if $(m_i + k, n_i + k)$ is a nonsingular node in the Padé table for $(A(z), B(z))$.*

Proof. Let M_{11}, M_{21}, M_{12}, and M_{22} be matrices of dimension $p(n_i + k) \times pk$, $p(m_i + k) \times pk$, $p(n_i + k) \times p(k - 1)$, and $p(m_i + k) \times p(k - 1)$, respectively, defined by

$$M_{11} = \begin{bmatrix} v_0 & & \\ v_{n_i} & \ddots & \\ & \ddots & v_0 \\ & & v_{n_i} \end{bmatrix}, \qquad M_{12} = \begin{bmatrix} 0 & & \\ 0 & & \\ q_0 & & \\ & \ddots & \\ q_{n_i - 1} & & q_0 \\ & \ddots & \\ & & q_{n_i - 1} \end{bmatrix}, \tag{4.27}$$

$$M_{21}=\begin{bmatrix} u_0 & & \\ u_{m_i} & \ddots & \\ & \ddots & u_0 \\ & & u_{m_i} \end{bmatrix}, \qquad M_{22}=\begin{bmatrix} 0 & & \\ p_{-1} & & \\ p_0 & \ddots & \\ & & p_{-1} \\ p_{m_i-1} & & \\ & \ddots & \\ & & p_{m_i-1} \end{bmatrix}. \tag{4.28}$$

In (4.28), $p_{-1}=0$ except when $m_1=0$ and $n_1=0$, in which case, according to (4.25), $p_{-1}=1$. Let M be

$$M=\begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}. \tag{4.29}$$

If we set

$$R_i(z)=\sum_{j=0}^{\infty} r_j z^j, \quad \text{with } \det(r_0)\neq 0, \quad \text{and } W_i(z)=\sum_{j=0}^{\infty} w_j z^j, \tag{4.30}$$

then, from (4.13) and (4.24), it follows that

$$T_{(m_i+k),(n_i+k)}\cdot M=\left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 \\ w_0 & & & r_0 & & \\ & \ddots & & & \ddots & \\ \vdots & & w_0 & \vdots & & r_0 \\ & & \vdots & & & \vdots \\ w_{2k-2} & \cdots & w_{k-1} & r_{2k-2} & \cdots & r_k \end{array}\right] \tag{4.31}$$

$$=\begin{bmatrix} 0 \\ T'_{(k-1),k} \end{bmatrix},$$

where $\mathbf{0}$ represents a zero matrix of size $p(m_i+n_i+1)\times p(2k-1)$ and

$$T'_{(k-1),k}=\left[\begin{array}{ccc|ccc} w_0 & & & r_0 & & \\ & \ddots & & & \ddots & \\ & & w_0 & & & r_0 \\ w_{2k-2} & \cdots & w_{k-1} & r_{2k-2} & \cdots & r_k \end{array}\right]. \tag{4.32}$$

We are now in a position to prove the theorem. Assume $T_{(m_i+k),(n_i+k)}$ is nonsingular. We show that $T'_{(k-1),k}$ is then also nonsingular. Let

$$X=[X_1,\cdots,X_{2k-1}]^t \tag{4.33}$$

be a $p(2k-1)\times 1$ vector that satisfies

$$T'_{(k-1),k}\cdot X=0. \tag{4.34}$$

Since $T_{(m_i+k),(n_i+k)}$ is nonsingular, (4.31) implies

$$M\cdot X=0. \tag{4.35}$$

From (4.35), we then obtain that

$$v_0\cdot X_1=0, \tag{4.36}$$

and consequently $X_1 = 0$, because v_0 is nonsingular. The first block equation from (4.34) then implies

$$r_0 \cdot X_{k+1} = 0. \tag{4.37}$$

Thus, $X_{k+1} = 0$ because r_0 is nonsingular. In a similar fashion, it follows that $X_2 = 0$ and $X_{k+2} = 0$. Continuing in this way, we obtain that $X = 0$, that is, $T'_{(k-1),k}$ is nonsingular.

Conversely, suppose that $T'_{(k-1),k}$ is nonsingular. Let $X = (X_1, \cdots, X_{2k-1})$ be a $1 \times p(2k-1)$ vector, $Y = (Y_1, \cdots, Y_{m_i+n_i})$ a $1 \times p(m_i + n_i)$ vector, and Z a $1 \times p$ vector. Consider

$$(Z, Y, X) \cdot T_{(m_i+k),(n_i+k)} = 0. \tag{4.38}$$

Multiplying both sides of (4.38) on the right by M, and using equation (4.31), it follows that

$$X \cdot T'_{(k-1),k} = 0. \tag{4.39}$$

Since $T'_{(k-1),k}$ is nonsingular, then $X = 0$. Then, in (4.38), using block columns 2 through $n_i + 1$ and block columns $n_i + k + 2$ through to $m_i + n_i + k + 1$ of $T_{(m_i+k),(n_i+k)}$, we obtain

$$Y \cdot T_{m_i,n_i} = 0. \tag{4.40}$$

Since T_{m_i,n_i} is nonsingular, $Y = 0$. Finally, block column $n_i + k + 1$ of $T_{(m_i+k),(n_i+k)}$ now yields

$$Z \cdot b_0 = 0. \tag{4.41}$$

Since b_0 is nonsingular, $Z = 0$. Hence, $T_{(m_i+k),(n_i+k)}$ is nonsingular.

Theorem 4.4 allows us to calculate nonsingular nodes of a pair of power series by calculating nonsingular nodes of the residual pair of power series. This gives us an iterative method of calculating nonsingular nodes.

THEOREM 4.5. *The cofactor and predecessor sequences for* $(A(z), B(z))$ *satisfy*

$$\begin{bmatrix} U_{i+1}(z) & P_{i+1}(z) \\ v_{i+1}(z) & Q_{i+1}(z) \end{bmatrix} = \begin{bmatrix} U_i(z) & P_i(z) \\ V_i(z) & Q_i(z) \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & z^2 \cdot I \end{bmatrix} \cdot \begin{bmatrix} V'(z) & Q'(z) \\ U'(z) & P'(z) \end{bmatrix}, \tag{4.42}$$

where $(U'(z), V'(z))$ *is the* RMPFr *of type* $(s_i - 1, s_i)$ *for* $(W_i(z), R_i(z))$ *and* $(P'(z), Q'(z))$ *is its predecessor.*

Proof. Since $(U_i(z), V_i(z))$ and $(U_{i+1}(z), V_{i+1}(z))$ are successive elements of the cofactor sequence (4.14), then, according to (4.10) and (4.11), (m_i, n_i) and (m_{i+1}, n_{i+1}) are successive nonsingular nodes along the $m - n$ off-diagonal path of the Padé table for $(A(z), B(z))$. By Theorem 4.4, then s_i is the smallest positive integer for which $(s_i - 1, s_i)$ is a nonsingular node in the Padé table for $(W_i(z), R_i(z))$. Accordingly, we can determine $(U'(z), V'(z))$ to be the RMPFr of type $(s_1 - 1, s_i)$ for $(W_i(z), R_i(z))$ and $(P'(z), Q'(z))$ to be its predecessor.

Let $U(z)$, $V(z)$, $P(z)$, and $Q(z)$ be defined by

$$\begin{bmatrix} U(z) & P(z) \\ V(z) & Q(z) \end{bmatrix} = \begin{bmatrix} U_i(z) & P_i(z) \\ V_i(z) & Q_i(z) \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & z^2 \cdot I \end{bmatrix} \cdot \begin{bmatrix} V'(z) & Q'(z) \\ U'(z) & P'(z) \end{bmatrix}. \tag{4.43}$$

We shall first show that $(U(z), V(z))$ given by (4.43) is the RMPFr of type (m_{i+1}, n_{i+1}) for $(A(z), B(z))$. Because RMPFr's are unique, then $(U(z), V(z))$ must be the $(i+1)$st term in the cofactor sequence.

Since $(U'(z), V'(z))$ is a RMPFr of type $(s_i - 1, s_i)$ for $(W_i(z), R_i(z))$, it satisfies

$$W_i(z) \cdot V'(z) + R_i(z) \cdot U'(z) = z^{2s_i} W'(z), \tag{4.44}$$

where $V'(0)$ is nonsingular. Then, using (4.13), (4.24), (4.43), and (4.44), we get

$$\begin{aligned} & A(z)\cdot V(z)+B(z)\cdot U(z) \\ & \quad = A(z)\cdot\{V_i(z)\cdot V'(z)+z^2Q_i(z)\cdot U'(z)\} \\ & \qquad + B(z)\cdot\{U_i(z)\cdot V'(z)+z^2P_i(z)\cdot U'(z)\} \\ & \quad = \{A(z)\cdot V_i(z)+B(z)\cdot U_i(z)\}\cdot V'(z) \\ & \qquad + \{A(z)\cdot Q_i(z)+B(z)\cdot P_i(z)\}\cdot z^2U'(z) \\ & \quad = z^{m_i+n_i+1}\cdot\{W_i(z)\cdot V'(z)+R_i(z)\cdot U'(z)\} \\ & \quad = z^{(m_i+s_i)+(n_i+s_i)+1}\cdot W'(z). \end{aligned} \tag{4.45}$$

Thus, condition II for a RMPFo of type (m_i+s_i, n_i+s_i) for $(A(z), B(z))$ is satisfied.

To verify condition I, expanding (4.43) gives

$$U(z) = U_i(z)\cdot V'(z)+z^2P_i(z)\cdot U'(z), \tag{4.46}$$

so that

$$\begin{aligned} \partial(U(z)) &\leqq \max\,(m_i+s_i, 2+(m_i-1)+s_i-1) \\ &= m_i+s_i. \end{aligned} \tag{4.47}$$

Similarly,

$$\partial(V(z)) \leqq n_i+s_i. \tag{4.48}$$

Finally, to verify condition II for a RMPFr (and, thus, condition III for a RMPFo, as well), observe that

$$V(0) = V_i(0)\cdot V'(0), \tag{4.49}$$

and, consequently, $V(0)$ is invertible since both $V_i(0)$ and $V'(0)$ are invertible. Notice that we have a somewhat stronger result here, namely that if $V_i(z)$ and $V'(z)$ are both normalized with $V_i(0)=I$ and $V'(0)=I$, then so is $V(z)$.

Therefore, $(U(z), V(z))$ is a RMPFr of type (m_i+s_i, n_i+s_i) for $(A(z), B(z))$. Thus, it is the $(i+1)$st term in the cofactor sequence and $W'(z)$ is the $(i+1)$st term in the power series remainder sequence for $(A(z), B(z))$.

Notice that the above arguments also hold in the special case when $m=n$ and $i=1$. In this case $P_1(z)=z^{-1}I$ which is not a matrix polynomial. However, the right side of equation (4.43) immediately multiplies the predecessor by z^2 which subsequently results in a matrix polynomial.

A similar argument shows that $(P(z), Q(z))$ given in (4.43) is the predecessor of the nonsingular node (m_{i+1}, n_{i+1}). Hence the recurrence relation (4.42) holds. □

For purposes of the algorithm given in the next section, observe that if $(U'(z), V'(z))$ is a RMPFo of type $(s-1, s)$ for $(W_i(z), R_i(z))$ and $(P'(z), Q'(z))=(0, I)$ then in (4.43) $(U(z), V(z))$ yields a RMPFo (rather than a RMPFr) of type (m_i+s, n_i+s) for $(A(z), B(z))$ and $(P(z), Q(z))=(U_i(z), V_i(z))$.

5. The algorithm. Given nonnegative integers m and n, the algorithm MPADE below makes use of Theorem 4.5 to compute the cofactor and predecessor sequences

(4.10) and (4.18), respectively. Thus, intermediate results available from MPADE include those RMPFr's $(U_i(z), V_i(z))$ for $(A(z), B(z))$ at all the nonsingular nodes (m_i, n_i), $i = 1, 2, \cdots, k-1$, smaller than (m, n), along the off-diagonal path $m_i - n_i = m - n$. The output gives results associated with the final node (m_k, n_k). If (m, n) is also a nonsignular node, then the output $(U_k(z), V_k(z))$ is a RMPFr of type (m, n) for $(A(z), B(z))$, and $(P_k(z), Q_k(z))$ is a RMPFo of type $(m-1, n-1)$. If (m, n) is a singular node, then the output $(U_k(z), V_k(z))$ is simply a RMPFo of type (m, n) for $(A(z), B(z))$, and now $(P_k(z), Q_k(z))$ is set to be the RMPFr of type (m_{k-1}, n_{k-1}). An exception occurs in the latter case when $k = 0$ and $m < n$. Here, all nodes along the off-diagonal path must have been singular, and for $(P_k(z), Q_k(z))$ the algorithm returns instead the initial value $(0, z^{n-m-1}I)$.

Note that, when (m, n) is not a nonsingular node, a simple modification of MPADE allows the computation of all RMPFo's of type (m, n) for $(A(z), B(z))$. It is only necessary to arrange to compute q columns of $[V_k', U_k']$, rather than p, in order to form a basis for the solution space of the equation in step 3.1 of MPADE. From this basis, it is then possible to construct a $p \times p$ matrix $V(z)$, and a corresponding $U(z)$, for which $(U(z), V(z))$ is a RMPFo of type (m, n) for $(A(z), B(z))$ and has the property that $V(z)$ is an invertible matrix, assuming such a RMPFo exists. This enhancement is not included in MPADE primarily to simplify the presentation of the algorithm.

MPADE(A, B, m, n)

If $m \geqq n$

then

M1) $\begin{bmatrix} m_0 \\ n_0 \end{bmatrix} \leftarrow \begin{bmatrix} m-n-1 \\ -1 \end{bmatrix}$

M2) $\left(s_0, \begin{bmatrix} V_1(z) & Q_1(z) \\ U_1(z) & P_1(z) \end{bmatrix}\right) \leftarrow INITIAL\_PADE(B(z), A(z), n, m)$

else

M3) $\begin{bmatrix} m_0 \\ n_0 \end{bmatrix} \leftarrow \begin{bmatrix} -1 \\ n-m-1 \end{bmatrix}$

M4) $\left(s_0, \begin{bmatrix} U_1(z) & P_1(z) \\ V_1(z) & Q_1(z) \end{bmatrix}\right) \leftarrow INITIAL\_PADE(A(z), B(z), m, n)$

M5) $\begin{bmatrix} m_1 \\ n_1 \end{bmatrix} \leftarrow \begin{bmatrix} m_0 + s_0 \\ n_0 + s_0 \end{bmatrix}$

M6) $i \leftarrow 1$

M7) Do while $m_i < m$

M8) Compute $R_i(z)$ satisfying (4.24)

M9) Compute $W_i(z)$ satisfying (4.13)

M10) $\left(s_i, \begin{bmatrix} U'(z) & P'(z) \\ V'(z) & Q'(z) \end{bmatrix}\right) \leftarrow INITIAL\_PADE(W_i(z), R_i(z), m - m_i - 1, n - n_i)$

M11) $\begin{bmatrix} U_{i+1}(z) & P_{i+1}(z) \\ V_{i+1}(z) & Q_{i+1}(z) \end{bmatrix} \leftarrow \begin{bmatrix} U_i(z) & P_i(z) \\ V_i(z) & Q_i(z) \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & z^2 \cdot I \end{bmatrix} \cdot \begin{bmatrix} V'(z) & Q'(z) \\ U'(z) & P'(z) \end{bmatrix}$

M12) $\begin{bmatrix} m_{i+1} \\ n_{i+1} \end{bmatrix} \leftarrow \begin{bmatrix} m_i + s_i \\ n_i + s_i \end{bmatrix}$

M13) $i \leftarrow i + 1$

End do

M14) $k \leftarrow i$

M15) Return $\left(\begin{bmatrix} U_k(z) & P_k(z) \\ V_k(z) & Q_k(z) \end{bmatrix}\right)$

End MPADE

INITIAL_PADE(W(z), R(z), m', n')

I1) $s \leftarrow 0$

I2) $d \leftarrow 0$

I3) Do while $s \leqq m'$ and $d = 0$

I4) Compute $d \leftarrow \det(T_{s,n'-m'+s})$

I5) $s \leftarrow s+1$

End do

I6) Solve

$$S_{s-1,n'-m'+s-1} \cdot \begin{bmatrix} V' \\ U' \end{bmatrix} = 0$$

If $s > 1$ and $d \neq 0$

then

I7) Solve

$$S_{s-2,n'-m'+s-2} \cdot \begin{bmatrix} Q' \\ P' \end{bmatrix} = 0$$

else

I8) $$\begin{bmatrix} Q' \\ P' \end{bmatrix} \leftarrow \begin{bmatrix} z^{n'-m'-1} I \\ 0 \end{bmatrix}$$

I9) Return $\left(s, \begin{bmatrix} U' & P' \\ V' & Q' \end{bmatrix}\right)$

End INITIAL_PADE

THEOREM 5.1. *The* MPADE *algorithm is valid.*

Proof. The argument is by induction on i.

Initially, in step M2 of MPADE, where $m \geqq n$, the parameters input to INITIAL_PADE are $W(z) = B(z)$, $R(z) = A(z)$, $m' = n$ and $n' = m$. Consequently, INITIAL_PADE computes $s = 1$, since in step I5

$$d = \det(T_{0,m-n}) = \det \begin{bmatrix} b_0 & & \\ & \ddots & \\ b_{m-n-1} & & b_0 \end{bmatrix} \neq 0 \tag{5.1}$$

when $m > n$, and $d = 1$ when $m = n$. In step I6, the algorithm solves

$$S_{0,m-n} \begin{bmatrix} V' \\ U' \end{bmatrix} = \begin{bmatrix} b_0 & & & a_0 \\ & \ddots & & \\ b_{m-n} & & b_0 & a_{m-n} \end{bmatrix} \begin{bmatrix} V' \\ U' \end{bmatrix} = 0 \tag{5.2}$$

and step I8 yields

$$\begin{bmatrix} Q' \\ P' \end{bmatrix} = \begin{bmatrix} z^{m-n-1} I \\ 0 \end{bmatrix}. \tag{5.3}$$

Since the substitution in step M2 of MPADE yields $s_0 = s$ and

$$\begin{bmatrix} U_1 & P_1 \\ V_1 & Q_1 \end{bmatrix} = \begin{bmatrix} V' & Q' \\ U' & P' \end{bmatrix}, \tag{5.4}$$

it follows that the initialization for $m \geqq n$ is exactly that given by (4.16) and (4.25).

Alternately, when $m < n$, initialization is accomplished by step M4. In this case, the parameters input to INITIAL_PADE are $W(z) = A(z)$, $R(z) = B(z)$, $m' = m$, and $n' = n$. If $\det(a_0) \neq 0$, INITIAL_PADE again computes $s = 1$, since in step I5

$$d = \det(T_{0,n-m}) = \det \begin{bmatrix} a_0 & & \\ & \ddots & \\ a_{n-m-1} & & a_0 \end{bmatrix} \neq 0. \tag{5.5}$$

Step I6 then solves (4.18) with $U_1 = U'$ and $V_1 = V'$, and, with the substitution $(P_1, Q_1) = (P', Q')$, step I7 yields the required predecessor (4.26). If $\det(a_0) = 0$, then step I5 determines the smallest integer $s \geqq 2$, if one exists, for which $d = \det(T_{s-1,n-m+s-1}) \neq 0$. Consequently, step I6 solves exactly the system (4.19) and step I7 must then yield the correct predecessor.

Assume that, for $i \geqq 1$, MPADE calculates $(U_i(z), V_i(z))$ and $(P_i(z), Q_i(z))$ correctly. We shall show that one pass through the while loop M7 correctly computes $(U_{i+1}(z), V_{i+1}(z))$ and its predecessor.

In step M10 and MPADE, the parameters input to INITIAL_PADE are $W(z) = W_i(z)$, $R(z) = R_i(z)$, $m' = m - m_i - 1$, and $n' = n - n_i$. Noting (4.9), step I4 computes the smallest positive integer s, if one exists, for which $d = \det(T_{s-1,s}) \neq 0$. Clearly, then I6 computes a RMPFr of type $(s-1, s)$ for $(W_i(z), R_i(z))$, and steps I7 and I8 its predecessor. Thus, the matrix polynomials in step M11 correspond exactly to those of (4.42); that is, the algorithm correctly computes $(U_{i+1}(z), V_{i+1}(z))$ and its predecessor.

To complete the proof of algorithm validity, consideration must be given to the case for which there exists no s such that $d \neq 0$ in the while loop I3 of INITIAL_PADE. On exit from the while loop, observe that $s = m' + 1$. Step I6 then computes $(U'(z), V'(z))$ to be a RMPFo of type (m', n') for $(W(z), R(z))$ and sets $(P'(z), Q'(z)) = (0, z^{n-m-1}I)$.

The case where there exists no s such that $d \neq 0$ can occur when INITIAL_PADE is invoked in steps M4 and M10 of MPADE, only. If it occurs at step M4, then $(U_1(z), V_1(z))$ becomes a RMPFo of type (m, n) for $(A(z), B(z))$ as computed by INITIAL_PADE, and $(P_1(z), Q_{1(z)}) = (0, z^{n-m-1}I)$. Since step M5 next yields $(m_1, n_1) = (m, n)$, the algorithm immediately terminates. On the other hand, if it occurs at step M10, then $s_i = m - m_i$, $(U'(z), V'(z))$ is a RMPFo of type $(s_i - 1, s_i)$ for $(W_i(z), R_i(z)$ and $(P'(z), Q'(z)) = (0, I)$. Accordingly (cf., last paragraph of § 4), $(U_{i+1}(z), V_{i+1}(z))$ computed in step M11 is a RMPFo of type (m, n) for $(A(z), B(z))$ and $(P_{i+1}(z), Q_{i+1}(z)) = (U_i(z), V_i(z))$. Since step M12 yields $(m_{i+1}, n_{i+1}) = (m, n)$, the algorithm terminates. □

6. Complexity of the MPADE algorithm. Note that, in steps M8 and M9 of MPADE, only the first $m + n - m_i - n_i$ terms in $R_i(z)$ and $W_i(z)$ are required to ensure the subsequent success of step M10. Indeed, only the first $2s_i$ terms, $s_i \leqq m - m_i$, are sufficient, but unfortunately s_i is not known prior to step M10. Nevertheless, an efficient implementation can take advantage of this observation by delaying the computation of $R_i(z)$ and $W_i(z)$. Declaring $(A(z), B(z))$, $(U_i(z), V_i(z))$, $(P_i(z), Q_i(z))$ to be global variables, the coefficients of $R_i(z)$ and $W_i(z)$ can be computed in INITIAL_PADE only when they become necessary. The cost analysis below assumes that the algorithm has been implemented in such a fashion.

In assessing the costs of MPADE, it is assumed that classical algorithms are used for the multiplication of polynomials. Only the more costly steps are considered. For these steps, Table 6.1 below provides cruder upper bounds on the number of multiplications in K required. The table provides separate bounds for the normal and abnormal cases.

In step I5 of INITIAL_PADE, it is assumed that the Gaussian elimination method is used to obtain the LU decomposition of $T_{(s-1),n'-m'+s-1}$. In addition, it is assumed that Gaussian elimination is accompanied with bordering techniques. Thus, as s increases by 1 in step I4, the results of the previous pass through the while loop are used to achieve the current LU decomposition. The bound for step I5 in Table 6.1 for the abnormal case assumes we do not take any advantage of the special nature of $T_{(s-1),n'-m'+s-1}$. In the normal case $s=1$, and $T_{0,n'-m'}$ is already in triangular form, and so no computation is required in step I5.

For step I6, it is assumed that the LU decomposition of $T_{(s-1),n'-m'+s-1}$ from step I5, is used to simplify the triangulation of $S_{(s-1),n'-m'+s-1}$. The solution $[V', U']$ is obtained finally by solving this triangularized S. Similar observations apply to step I7 in the abnormal case.

Since steps M2, M4, and M10 simply invoke INITIAL_PADE, estimates of their costs are obtained by summing the costs of steps I5, I6, and I7 with appropriate substitutions of variables. Note that the cost of M2 is the same in both the normal and abnormal case, since for $m \geqq n$ it is always true that $s_0 = 1$.

An upper bound for the number of multiplications in K required by MPADE is obtained by summing the costs of the last six rows in Table 6.1 for $i = 0, 1, \cdots, k$. We use the fact that

$$\sum_{i=0}^{k} s_i = m, \text{ if } m \geqq n, \text{ and } \sum_{i=0}^{k} s_i = n, \text{ if } m < n. \tag{6.1}$$

In addition,

$$\sum_{i=0}^{k} m_i^{\alpha} s_i^{\beta} \leqq m^{\alpha+\beta} \text{ and } \sum_{i=0}^{k} n_i^{\alpha} s_i^{\beta} \leqq n^{\alpha+\beta}. \tag{6.2}$$

Then, steps M4 and M10 in the abnormal case have a complexity of $O(p^3(m+n)^3)$ and the remaining steps a complexity of $O(p^3(m+n)^2)$, at worst. When $(A(z), B(z))$ is normal, then due to the fact that $T_{s-1,n'-m'+s-1}$ is always in triangular form, the complexity of MPADE reduces to $O(p^3(m+n)^2)$. This is also true when $(A(z), B(z))$ is nearly normal. In this case s_i is often larger than 1, but the matrix $T_{s-1,n'-m'+s-1}$ is

TABLE 6.1
Bounds on operations per step.

Step	Normal case	Abnormal case
I5	0	$p^3(n'-m'-1+2(s-1))^3/3$
I6	$p^3(n'-m'+1)^2/2$	$p^3(n'-m'+2s-1)^2/2$
I7	0	$p^3(n'-m'+2s-1)^2/2$
M2	$p^3(m-n)^2$	$p^3(m-n)^2$
M4	$p^3(m-n)^2$	$p^3[(n-m-1+2(s_0-1))^3/3+(n-m+2s_0-1)^2]$
M8	$2p^3(m_i+n_i+2)$	$2p^3(m_i+n_i+2)s_i$
M9	$2p^3(m_i+n_i+2)$	$2p^3(m_i+n_i+2)s_i$
M10	0	$8p^3(s_i-1)^3/3$
M11	$8p^3(m_i+n_i+2)$	$8p^3(m_i+n_i+2)s_i$

also always in triangular form and so again the complexity is $O(p^3(m+n)^2)$. In particular, in the scalar case the complexity of MPADE is $O((m+n)^2)$.

The algorithm gives the worst performance when no nonsingular nodes are encountered along the $m-n$ off-diagonal path. In this case, with $m < n$, the algorithm reduces to solving one Sylvester system

$$S_{m,n}\begin{bmatrix} V_1 \\ U_1 \end{bmatrix} = 0 \tag{6.3}$$

in step M4 of MPADE. In Table 6.1, with $s_0 = m+1$, then the cost is simply that of Gaussian elimination, namely, approximately $p^3(m+n)^3/3$. Note that with the existence of even one nonsingular node the cost of MPADE can be dramatically reduced. If, for example, this one nonsingular node is $(m_1, n_1) = (m/2, n-m/2)$, where m is even, then $s_0 = 1+m/2$, $s_1 = m/2$ and the algorithm reduces essentially to solving (in steps M4 and M10) two Sylvester systems, each of approximately half the total size. This results in a saving of a factor of 4 over the simple use of Gaussian elimination. Algorithms requiring normality, on the other hand, break down when even one intermediate node is singular.

7. Conclusions. We have considered the problem of determining an adequate definition for a rational approximant of a formal matrix power series and also, given a suitable definition, the problem of computing it. We have restricted our attention to square matrix power series.

In attempting to extend the notion of Padé approximation to matrix power series, we have followed the classical theory of Padé approximants for scalar power series. We introduce the notion of a Padé form, which always exist but may not be unique, and also the notion of Padé fraction, which is unique but need not exist. The definition of Padé form is meant to be as broad as possible. By constructing all the Padé forms of type (m, n), it is always possible to determine ones for which the denominator is invertible, should one exist.

The notion of a matrix power series remainder sequence introduced in this paper is a generalization of one given by Cabay and Kossowski [9] for scalar power series. The cofactor sequence, which is shown to be associated with the remainder sequence, yields directly all the Padé fractions at the nonsingular nodes of a particular off-diagonal path of the Padé table. By determining also the (unique) Padé form at nodes preceding the nonsingular nodes, we are able to compute Padé fractions iteratively from one nonsingular node to the next. The resulting algorithm is at least as fast as other algorithms for computing matrix Padé fractions, and it is the only one that succeeds in the abnormal case.

The algorithm can be improved in a number of ways. We expect that the cost of the decomposition of $T_{s-1,n'-m'+s-1}$ in step I5 and, consequently, of $S_{s-1,n'-m'+s-1}$ in step I6 can be improved by taking advantage of the special structure of Sylvester matrices. The algorithm would also experience an improvement if it were possible to identify additional points between nonsingular nodes for which Theorem 4.5 is valid. This would improve the algorithm by decreasing the s_i. This, and in general the nature of Padé forms between nonsingular nodes, is a subject for further research. Finally, by appealing to fast methods for polynomial arithmetic, it is of interest to attempt to develop a recursive divide-and-conquer version of MPADE.

For normal and nearly normal power series, progressing from one nonsingular node to the next is equivalent to power series division of the residuals associated with the nonsingular nodes (because $S_{s-1,n'-m'+s-1}$ in step I5 of INITIAL_PADE, with the

exception of one column, reduces to a triangular matrix). Thus, in this case and in addition when $A(z)$ and $B(z)$ are matrix polynomials, there is a strong analogy between MPADE and Euclid's algorithm. It is a subject for future research to investigate the possibility of using MPADE to compute the greatest common divisor of two matrix polynomials in the abnormal case.

REFERENCES

[1] G. A. BAKER, *Essentials of Padé Approximants*, Academic Press, New York, 1975.

[2] N. K. BOSE AND S. BASU, *Theory and recursive computation of* 1-D *matrix Padé approximants*, IEEE Trans. on Circuits and Systems, 4 (1980), pp. 323-325.

[3] R. BRENT, F. G. GUSTAVSON, AND D. Y. Y. YUN, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, J. Algorithms, 1 (1980), pp. 259-295.

[4] F. BROPHY AND A. C. SALAZAR, *Considerations of the Padé approximant technique in the synthesis of recursive digital filters*, IEEE Trans. Audio and Electroacoustics, AU-21 (1973), pp. 500-505.

[5] A. BULTHEEL, *Recursive algorithms for the matrix Padé table*, Math. Computation, 35 (1980), pp. 875-892.

[6] ———, *Recursive algorithms for nonnormal Padé tables*, SIAM J. Appl. Math., 39 (1980), pp. 106-118.

[7] ———, *Algorithms to compute the reflection coefficients of digital filters*, Numer. Methods Approx. Theory, 7 (1983), pp. 33-50.

[8] S. CABAY AND D. K. CHOI, *Algebraic computations of scaled Padé fractions*, SIAM J. Comput., 15 (1986), pp. 243-270.

[9] S. CABAY AND P. KOSSOWSKI, *Power series remainder sequences and Padé fractions over integral domains*, J. Symbolic Computation, to appear.

[10] J. DURBIN, *The fitting of time-series models*, Rev. Inst. Internat. Statist., 28 (1960), pp. 233-244.

[11] O. I. ELGERD, *Control Systems Theory*, McGraw-Hill, New York, 1967.

[12] W. B. GRAGG, *The Padé table and its relation to certain algorithms of numerical analysis*, SIAM Rev., 14 (1972), pp. 1-61.

[13] G. LABAHN, *Matrix Padé approximants*, Master's thesis, Dept. of Computing Science, University of Alberta, Edmonton, Canada, 1986.

[14] N. LEVINSON, *The Wiener* RMS *(root mean square) error in filter design*, J. Math. and Phys., 25 (1947), pp. 261-278.

[15] G. NEMETH AND M. ZIMANYI, *Polynomial type Padé approximants*, Math. Comp., 38 (1982), pp. 553-565.

[16] J. RISSANEN, *Solution of linear equations with Hankel and Toeplitz matrices*, Numer. Math., 22 (1974), pp. 361-366.

[17] ———, *Recursive evaluation of Padé approximants for matrix sequences*, IBM J. Res. Develop. (1972), pp. 401-406.

[18] ———, *Algorithms for triangular decomposition of block Hankel and Toeplitz matrices with application to factoring positive matrix polynomials*, Math. Comp., 27 (1973), pp. 147-154.

[19] Y. STARKAND, *Explicit formulas for matrix-valued Padé approximants*, J. Comp. and Appl. Math., 5 (1979), pp. 63-65.

[20] W. F. TRENCH, *An algorithm for the inversion of finite Hankel matrices*, SIAM J. Appl. Math., 13 (1965), pp. 1102-1107.

[21] P. WYNN, *The rational approximation of functions which are formally defined by a power series expansion*, Math. Comp., 14 (1960), pp. 147-186.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 658-669, August 1989

002

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*

COSTAS S. ILIOPOULOS†

Abstract. An $O(s^5M(s^2))$ algorithm for computing the canonical structure of a finite Abelian group represented by an integer matrix of size s (this is the Smith normal form of the matrix) is presented. Moreover, an $O(s^3M(s^2))$ algorithm for computing the Hermite normal form of an integer matrix of size s is given.

The upper bounds derived on the computational complexity of the algorithms above improve the upper bounds given by Kannan and Bachem in [*SIAM J. Comput.*, 8 (1979), pp. 499-507] and Chou and Collins in [*SIAM J. Comput.*, 11 (1982), pp. 687-708].

Key words. Smith normal form, Hermite normal form, integer matrices, computational complexity

AMS(MOS) subject classifications. 15A21, 20K01, 20K05

1. Introduction. Recently Chou and Collins [2] improved the results of Kannan and Bachem [6] on the computation of the Hermite and Smith normal form (abbreviated HNF and SNF, respectively) of an integer matrix. Reduction to the HNF and SNF can be done via integer row-column operations (abbreviated IRC operations; see [8]).

A closely related problem is the computation of the canonical structure [8] of a finite Abelian group G represented by a set of defining relations that is associated with an integer matrix. We can reduce the exponents of the generators in the set of defining relations modulo the order of the group, say d, and add to the set the relation $x^d = 1$, for every generator x, respecting the structure of the group. This fact allows the use of arithmetic modulo d for the IRC operations over the associated matrix (as formulated in § 2).

The computational complexity is measured in terms of elementary operations. An *elementary operation* is a Boolean operation on a single binary bit or pair of bits or an input or shift of a binary bit. $M(n)$ denotes an upper bound on the number of elementary operations required for multiplication of two integers of length n bits [10] and the size s of an $m \times n$ matrix A is the number $m+n+\log \|A\|$, with $\|A\| = \max_{i,j} \{|a_{ij}|\}$.

The main results of this paper are the following:

(i) An $O(s^3M(s^2))$ elementary operations algorithm for computing the HNF of a nonsingular integer matrix of size s.

(ii) An $O(s^5M(s^2))$ elementary operations algorithm for computing the SNF of a nonsingular integer matrix of size s.

The upper bounds found for (i) and (ii) improve the upper bounds given by Chou and Collins in [2] (see also Kannan and Bachem [6]). For (i) Chou and Collins proved an upper bound of $O(s^8)$ elementary operations and for (ii), using the Kannan-Bachem

* Received by the editors December 1, 1983; accepted for publication (in revised form) March 3, 1988. This research was supported by the Greek section of scholarships and research Alexander S. Onassis public benefit foundation.

† Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom. Present address, Royal Holloway College, University of London, Department of Computer Science, Egham TW20 0EX, United Kingdom.

method for reduction to SNF together with the improvements of Chou and Collins, we can derive an upper bound $O(s^{11})$ elementary operations.

The IRC operations performed on A in reduction to HNF matrix T are associated with a factorization of T as MA, where M is a square unimodular matrix. Similarly the IRC operations performed on A in reduction to SNF matrix D are associated with a factorization of D as BAC, where B and C are square unimodular matrices. Bounds on the complexity of the computation of the matrices M, B, C are also provided.

1.1. Preliminaries. Some upper bounds on the size of the determinant of an integer matrix and on the running time of algorithms for matrix multiplication and for computing the greatest common division (gcd) of two integers are exposed here.

THEOREM 1.1. (Hadamard). *Let A be an $n \times n$ matrix and let $d_i = \max_{1 \leqq j \leqq n} \{|\alpha_{ij}|\}$ for $1 \leqq i \leqq n$. Then*

$$|\det(A) \leqq \prod_{i=1}^{n} \left(\sum_{j=1}^{n} |a_{ij}|^2 \right)^{1/2}.$$

COROLLARY 1.2. *Let A be an $n \times n$ matrix with integer entries. Then we have the following*:

(i) $|\det(A)| \leqq n^{n/2} \|A\|^n$.

(ii) $\log(|det(A)|) = O(s^2)$, *where s is the size of A.*

THEOREM 1.3 (Gaussian elimination). *There exists an algorithm for computing the row-echelon form of an $m \times n$ matrix A with entries from a field in $O(mnrM(\log|d|))$ elementary operations, where $r = \operatorname{rank}(A)$, and d is the largest in absolute value determinant of a minor of A.*

Proof. It follows from the fact that the numerators and denominators of the rationals arising during the Gaussian elimination have modulos at most d, with d the maximum determinant of all square submatrices of A (see [4, p. 26]). □

The problems "computing the determinant of a square matrix," "computing the inverse of a nonsingular matrix," and "matrix multiplication" are computationally equivalent (for details see Strassen [12], Winograd [13], or Aho et al. [1]). In 1969 Strassen showed in [12] that matrix multiplication of 2×2 matrices can be done with seven multiplications instead of eight and, in general, the multiplication of $n \times n$ matrices can be done with $O(n^{\log 7}) = O(n^{2.81})$ multiplications instead of $O(n^3)$ multiplications required by the classical algorithm. This surprising result led to a series of improved algorithms for matrix multiplication, requiring even fewer multiplications. Currently the best upper bound on the number of multiplications required for matrix multiplication is the following.

THEOREM 1.4 (Coppersmith-Winograd [3]). *There exists an algorithm for multiplying two $n \times n$ matrices using $O(n^{\alpha})$ multiplications, where*

$$\alpha = 2.376.$$

In the sequel, when the time required for the "computation of the determinant" is not the bottleneck of the overall time complexity of the problem in investigation, we make use of the Gaussian elimination; otherwise we make use of the algorithm of Theorem 1.4. The reason is that the Coppersmith-Winograd algorithm is not practical, due to the very large hidden constant in the upper bound on the number of multiplications required.

The following theorem yields an upper bound on Knuth's algorithm (see [7]) for computing the greatest common divisor of two integers. This upper bound was proved by Schönhage in [9].

THEOREM 1.5 (Extended Euclidean Algorithm (EEA)). *There exists an algorithm for computing the greatest common divisor r of two n bit integers a_1 and a_2 and two integers x_1 and x_2 such that*

$$x_1a_1 + x_2a_2 = r$$

with

$$|x_1| \leqq |a_2|/2, \qquad |x_2| \leqq |a_1|/2$$

in $O(\log nM(n))$ elementary operations.

COROLLARY 1.6. *There exists an algorithm for computing r, the gcd of the integers $a_1, \cdots, a_k$ of length at most n binary bits and k integers $t_1, \cdots, t_k$ such that*

$$t_1a_1 + \cdots + t_ka_k = r$$

with

$$\log \|t\| := \max_i \{|t_i|\} = O(n \log k)$$

in $O(k \log nM(n))$ elementary operations.

Proof. Without loss of generality, assume that k is a power of 2. Let $a_1, \cdots, a_k$ be leaves of a balanced binary tree. Moreover let each parent node be the gcd of its children. Therefore the root of the tree is the gcd $(a_1, \cdots, a_k)$.

We can build up the tree using the EEA of Theorem 1.5. The t_i's can be computed by combining the expressions of the gcd's at each level of the tree.

The upper bounds can be found by a simple manipulation of the upper bounds of Theorem 1.5 and using the fact that the height of the tree is equal to $\log k$. □

2. The procedures ELIMINATECOL–ELIMINATEROW. Suppose that A is an $m \times n$ integer matrix associated with the set of defining relations of a finite Abelian group G. The matrix A has rank n and if B is an $n \times n$ nonsingular submatrix of A, then the determinant of B denoted $\det(B)$, is readily shown to be a multiple of the order $|G|$ of the group G. Moreover if I_n denotes the $n \times n$ identity matrix, then the $(m+n) \times n$ matrix

$$K = \begin{bmatrix} A \\ dI_n \end{bmatrix} \quad \text{with } d = \det(B) \tag{2.1}$$

presents the same group.

Matrix B and its determinant d can be computed by means of Gaussian elimination over the rationals.

ALGORITHM 2.1.

INPUT: The matrix K of (2.1) and an integer ρ such that $1 \leqq \rho \leqq n$.

OUTPUT: An $(m+n) \times n$ matrix K^* of the form

$$\begin{bmatrix} A^* \\ dI_n \end{bmatrix}$$

where the ρth column of A^* is of the form $(a^*_{1\rho}, \cdots, a^*_{\rho\rho}, 0, \cdots, 0)^T$ and K^* is a transformation of K via integer row operations.

Procedure ELIMINATECOL (K, d, ρ)

begin

if $k_{\rho\rho} \neq 0$ *then*
begin
(1) $r \leftarrow \gcd(k_{\rho\rho}, k_{\rho+1,\rho})$;
(2) compute integers x_1, x_2: $x_1 k_{\rho\rho} + x_2 k_{\rho+1,\rho} = r$;
comment Use the algorithm of Theorem 1.5.
(3) $y_1 \leftarrow k_{\rho+1,\rho}/r$;
(4) $y_2 \leftarrow -k_{\rho\rho}/r$;

$$L \leftarrow \left[\begin{array}{c|c|c} I_{\rho-1} & 0 & 0 \\ \hline 0 & \begin{matrix} y_1 & y_2 \\ x_1 & x_2 \end{matrix} & 0 \\ \hline 0 & 0 & I_{m+n-\rho-1} \end{array}\right];$$

comment Note that L is unimodular.
(5) $K \leftarrow L \cdot K$;
comment The result of step (5) is a matrix K having $k_{\rho\rho} = 0$ and $k_{\rho+1,\rho} = r$.
end
(6) $s \leftarrow \gcd(k_{\rho+1,\rho}, k_{\rho+2,\rho}, \cdots, k_{m\rho}, d)$;
(7) compute integers t_j for $\rho+1 \leqq j \leqq m+1$: $\sum_{j=\rho+1}^{m} t_j k_{j\rho} + t_{m+1} d = s$;
(8) $t_j \leftarrow t_j \pmod d$ for $\rho+1 \leqq j \leqq m$;
$t_{m+1} \leftarrow (s - \sum_{j+\rho+1}^{m} t_j k_{j\rho})/d$;
(9) ROW $(\rho) \leftarrow$ ROW $(\rho) + \sum_{j=\rho+1}^{m} t_j$ ROW $(j) + t_{m+1}$ ROW $(m+\rho)$;
comment The entry $k_{\rho\rho}$ of K is equal to s. Note that the computation of t_{m+1} is not necessary, since we can merely assign $k_{\rho\rho} = s$
(10) ROW $(i) \leftarrow$ ROW $(i) - (k_{i\rho}/s)$ ROW (i), for $\rho+1 \leqq i \leqq m$;
comment Now COL $(\rho) = (k_{1\rho}, \cdots, k_{\rho\rho}, 0, \cdots, 0, d, 0, \cdots, 0)^T$
(11) ROW $(i) \leftarrow$ ROW $(i) - (\lfloor k_{ij}/d \rfloor)$ ROW $(m+j)$, $1 \leqq i \leqq m$, $1 \leqq j \leqq n$;
comment At this step all the entries of K are reduced mod d.
return K;
end

Remark 2.1. We may speed up (in practice) the above procedure by making the following modifications:

(i) At the beginning of the procedure ELIMINATECOL we can check if there exists a $k_{i\rho}$ for some $\rho \leqq i \leqq m$ such that $k_{i\rho}$ divides $k_{j\rho}$ for every $\rho \leqq j \leqq m+\rho$ and if it exists, then interchange ROW (i) and ROW (ρ) and go to step (10) with $s = k_{i\rho}$.

(ii) At the beginning of the procedure we may check whether or not there exists a $k_{i\rho}$ for some $\rho \leqq i \leqq m$ such that $k_{i\rho} = 0$ and if it exists then interchange ROW (i) and ROW (ρ) and go to Step (6).

Note that the amount of computational time that can be saved with the above modifications does not effect the asymptotic worst-case time complexity.

Remark 2.2. Suppose that the matrix K of (2.1) is transformed to K^* after an application of the procedure ELIMINATECOL (K, d, ρ). Since every integer-row operation can be expressed as matrix multiplication of K by a unimodular matrix, we can easily modify ELIMININATECOL to compute a unimodular matrix L such that $LK = K^*$. Note that this can be done without an increase in asymptotic time complexity and $\log \|L\| = O(\log \|K\|)$.

PROPOSITION 2.2. *The procedure* ELIMINATECOL *terminates in*

$$O(mnM(\log(m\|K\|) + mM(\log \|K\|) \log\log \|K\|)$$

elementary operations in the worst-case and the norm $\|K^*\|$ *of the output matrix* K^* *is equal to* $|d|$.

Proof. Steps (1)-(2) require $O(M(\log \|A\|) \log\log \|A\|)$ elementary operations for an application of the EEA and

$$|x_1| \leqq |k_{\rho+1,\rho}|/2, \qquad |x_2| \leqq |k_{\rho\rho}|/2. \tag{2.2}$$

Steps (3)-(4) require $O(M(\log \|A\|))$ elementary operations for divisions.

In view of (2.2), step (5) requires at most $4n$ multiplications comprising $O(nM(\log \|A\|))$ elementary operations. If A' denotes the m top rows of K after step (5), then we can see that

$$\|A'\| = O(\|A\|^2). \tag{2.3}$$

Steps (6)-(7) require $O(mM(\log \|K\|) \log\log \|K\|)$ elementary operations for an application of the algorithm of Corollary 1.6, since, using (2.3), it follows that

$$\max\{k_{\rho+1,\rho}, \cdots, k_{m\rho}, d\} = O(\|A\|^2 + |d|) = O(\|K\|^2)$$

where K denotes the input matrix. (Note that $|d| = O(\|K\|)$.)

Step (8) requires $O(mM(\log \|K\|))$ elementary operations for divisions.

Step (9) requires $O(mnM(\log \|K\|))$ elementary operations for mn multiplications in the worst case. If A'' denotes the m top rows of K after Step (9) then

$$\|A''\| = O(m\|K\|^3). \tag{2.4}$$

Step (10) requires $O(mnM(\log \|A''\|)) = O(mnM(\log (m\|K\|)))$ elementary operations for divisions/multiplications when we use (2.4).

Step (11) requires $O(mnM(\log (m\|K\|)))$ elementary operations. The entries of K are reduced modulo d; hence the entries of the output matrix are bounded by $|d| = O(\|K\|)$. □

Remark 2.3. We can observe that the dominant complexity of the procedure is the computation in lines (9), (10), and (11) that require $O(mnM(\log (m\|K\|)))$ elementary operations, except in the case of A with enormous entries

$$\|A\| \geqq 2^{2^{m+\varepsilon}}.$$

ALGORITHM 2.3.

INPUT: The matrix K of (2.1) and an integer ρ such that $1 \leqq \rho \leqq m$.

OUTPUT: An $(m+n) \times n$ matrix K^* of the form

$$\begin{bmatrix} A^* \\ dI_n \end{bmatrix}$$

where the ρth row of A^* is of the form $(a^*_{\rho 1}, \cdots, a^*_{\rho\rho}, 0, \cdots, 0)$ and K^* is a transformation of K via IRC operations.

(Note that procedure ELIMINATEROW is almost symmetric with the procedure ELIMINATECOL; steps (6), (10), and (12) are the only nonsymmetries.)

Procedure ELIMINATEROW (K, d, ρ)
  begin
    if $k_{\rho\rho} \neq 0$ *then*
      begin

(1) $r \leftarrow \gcd(k_{\rho\rho}, k_{\rho,\rho+1})$;

(2) compute x_1, x_2: $x_1 k_{\rho\rho} + x_2 k_{\rho,\rho+1} = r$ with $|x_1| \leqq |k_{\rho,\rho+1}|/2$, $|x_2| \leqq |k_{\rho\rho}|/2$;

(3) $y_1 \leftarrow k_{\rho,\rho+1}/r$;

(4) $y_2 \leftarrow -k_{\rho\rho}/r$;

(5)
$$K \leftarrow K \begin{bmatrix} I_{\rho-1} & 0 & 0 \\ 0 & \begin{matrix} y_1 & x_1 \\ y_2 & x_2 \end{matrix} & 0 \\ 0 & 0 & I_{n-\rho-1} \end{bmatrix}.$$

(6) $k_{m+\rho,\rho} \leftarrow d$; $k_{m+\rho+1,\rho+1} \leftarrow d$; $k_{m+\rho+1,\rho} \leftarrow 0$; $k_{m+\rho,\rho+1} \leftarrow 0$;

comment The above operation can be expressed as a sequence of row operations. See Remark 2.4 below.

end

(7) $s \leftarrow \gcd(k_{\rho,\rho+1}, \cdots, k_{\rho n})$;

(8) Compute t_j for $\rho+1 \leqq j \leqq n$: $\sum_{j=\rho+1}^{n} t_j k_{\rho j} = s$;
$t_j \leftarrow t_j \bmod d$ for $\rho+1 \leqq j \leqq n$;

(9) $\mathrm{COL}(\rho) \leftarrow \mathrm{COL}(\rho) + \sum_{j=\rho+1}^{n} t_j \,\mathrm{COL}(j)$;

(10) $\mathrm{ROW}(m+j) \leftarrow \mathrm{ROW}(m+j) - t_j \,\mathrm{ROW}(m+\rho)$, for $\rho+1 \leqq j \leqq n$;

comment This step is not necessarily executed, since we can merely assign $k_{m+\rho+1,\rho} = \cdots = k_{m+n,\rho} = 0$.

(11) $\mathrm{COL}(j) \leftarrow \mathrm{COL}(j) - (k_{\rho j}/s)\,\mathrm{COL}(\rho)$, $\rho+1 \leqq j \leqq n$;

(12) $\mathrm{ROW}(m+\rho) \leftarrow \mathrm{ROW}(m+\rho) - \sum_{j=\rho+1}^{n} (k_{\rho j}/s)\,\mathrm{ROW}(m+j)$;

comment This step is not necessarily executed, since we can merely assign $k_{m+\rho,\rho+1} = \cdots = k_{m+\rho,n} = 0$

(13) $\mathrm{ROW}(i) \leftarrow \mathrm{ROW}(i) - (\lfloor k_{ij}/d \rfloor)\,\mathrm{ROW}(m+j)$, $\rho \leqq i \leqq n$, $\rho \leqq j \leqq m$;

return K;

end

Remark 2.4. If M is the inverse of

$$\begin{bmatrix} y_1 & x_1 \\ y_2 & x_2 \end{bmatrix},$$

then we can see that

$$\begin{bmatrix} I_{m+\rho-1} & 0 & 0 \\ 0 & M & 0 \\ 0 & 0 & I_{m+n-\rho-1} \end{bmatrix}.$$

K is equivalent to step (6), and thus it can be expressed as a sequence of row operations.

The row operations at steps (6), (10), and (12) are necessary in order to preserve the form of the n bottom rows of K as dI_n.

Remark 2.5. We may speed up the above procedure by modifying it in a way similar to the modification suggested for procedure ELIMINATECOL in Remark 2.1.

Remark 2.6. Suppose that K^* is the output matrix of ELIMINATEROW (K, d, ρ). Then it is not difficult to modify ELIMINATEROW in order to compute two unimodular matrices L and R such that

$$LKR = K^*.$$

PROPOSITION 2.4. *The procedure* ELIMINATEROW *terminates in*

$$O(mnM(\log(m\|K\|)) + nM(\log\|K\|)\log\log\|K\|)$$

elementary operations and the norm $\|K^*\|$ *of the output matrix* K^* *is equal to* $|d|$.

Remark 2.7. We can show that the computation of the matrices L and R of Remark 2.6 can be done without any asymptotic increase in the worst-case complexity of the procedure ELIMINATEROW and $\log(\max\{\|L\|, \|R\|\}) = O(\log \|K\|)$.

3. The main algorithm.

ALGORITHM 3.1.

INPUT: An $m \times n$ matrix representing a finite Abelian group G.

OUTPUT: The canonical structure of the group G.

begin

(1) $d \leftarrow$ The determinant of an $n \times n$ nonsingular submatrix of A;

(2) $K = \begin{bmatrix} A \\ dI_n \end{bmatrix}$;

(3) *for* $\rho = 1$ *to* n *do*
begin

(4) *repeat*

(5) ELIMINATEROW (K, d, ρ);

(6) ELIMINATECOL (K, d, ρ);

(7) *until* $k_{\rho\rho} | k_{\rho i}$ for $\rho + 1 \leqq i \leqq n$;
$k_{\rho i} \leftarrow 0$ for $\rho + 1 \leqq i \leqq n$;

(8) *end*
$A \leftarrow$ the n top rows of K;

(9) *for* $p = 1$ *to* n *do*
begin

(10) *for* $q = p + 1$ *to* n *do*
begin
$h \leftarrow a_p$;
$a_{pp} \leftarrow \gcd(a_{qq}, h)$;
$a_{qq} \leftarrow a_{qq} \cdot h / a_{pp}$;

(11) *end*

(12) *end*
return A;

end

PROPOSITION 3.2. *Algorithm* 3.1 *correctly computes a canonical basis for the Abelian group* G *in* $O(mn(n + \log\log \|K\|) \log \|K\| M(\log(m\|K\|)) + mn^2 M(\log d^*))$ *elementary operations, where* $\|K\| = \max\{|d|, \|A\|\}$ *and* $d^* = \max\{|\delta|: \delta$ *is the determinant of a square submatrix of* $A\}$.

Proof. The computation of a multiple of the order of the group G (Step (1)) requires $O(mn^2 M(\log d^*))$ elementary operations by Theorem 1.3.

Steps (5) and (6) require $O(mnM(\log m\|K\|) + mM(\log \|K\|) \log\log \|K\|)$ elementary operations; this follows from Propositions 2.4 and 2.2 and the facts

$$\|K^{(i)}\| \leqq |d| \leqq \|K\|, \qquad m \geqq n$$

where $K^{(i)}$ denotes the current matrix K at the ith iteration of the loop (4)-(7). The number of iterations required by loop (4)-(7) is at most $O(\log \|K\|)$ since

$$|k_{\rho\rho}^{(i)}| \leqq |k_{\rho\rho}^{(i-1)}|/2. \tag{3.1}$$

Hence loop (3)-(8) requires $O(n \log \|K\|)$ iterations.

Loop (9)-(12) requires $O(n^2)$ applications of EEA and $O(n^2)$ multiplications/divisions comprising $O(n^2M(\log|d|)\log\log|d|) = O(n^2M(\log\|K\|)\log\log\|K\|)$ elementary operations.

The proposition follows from the above analysis. □

COROLLARY 3.3. *Algorithm* 3.1 *terminates in* $O(mn^2[n+\log(n\log(n\|A\|))]\log(n\|A\|)M(n\log(mn\|A\|)))$ *elementary operations.*

Proof. We derive the proof from Proposition 3.2, using the fact that

$$\max\{\|K\|, d^*\} = \max\{\|A\|, |d|, d^*\} \leqq (n\|A\|)^n. \qquad \square$$

COROLLARY 3.4. *If a finite Abelian group is represented by a matrix of size s, then we can compute its canonical structure in* $O(s^5M(s^2))$ *elementary operations.* □

4. Hermite normal form.

THEOREM 4.1 (Hermite, see [5]). *Given a nonsingular* $n \times n$ *integer matrix A, there exists an* $n \times n$ *unimodular matrix M such that* $MA = T$ *is upper triangular with positive diagonal elements. Further, each off-diagonal element of T is nonpositive and strictly less in absolute value than the diagonal elements in its column.*

The algorithm below for computing the HNF of an integer matrix makes use of a version of the procedure ELIMINATECOL (K, d, ρ) with $K = \begin{bmatrix} A \\ dI_n \end{bmatrix}$, which eliminates all the entries below the diagonal element $k_{\rho\rho}$ of the ρth column of the matrix K. The modified version of ELIMINATECOL is as follows:

(i) Eliminate the entries $\alpha_{\rho+1,\rho}, \cdots, \alpha_{n,\rho}$ of K in a similar manner as ELIMINATECOL followed by reduction modulo d.

(ii) Do the row operation $\mathrm{ROW}(m+\rho) \leftarrow \mathrm{ROW}(m+\rho) - (d/k_{\rho\rho})\,\mathrm{ROW}(\rho)$ and let $A \leftarrow$ the n top rows of K and $\mathrm{ROW}(m+\rho)$ as last row.

(iii) The output matrix is of the form

$$\begin{bmatrix} & A & \\ dI_{\rho-1} & 0 & \\ & \vdots & 0 \\ 0 & 0 & dI_{n-\rho} \end{bmatrix}.$$

The above modified version of ELIMINATECOL will be called ELIMINATECOL* in the sequel.

Remark 4.1. The ρth column of the output matrix (say K^*) is of the form $(k_{1\rho}, \cdots, k_{\rho\rho}, 0, \cdots, 0)^T$. The running time of ELIMINATECOL* is asymptotically the same with the running time of ELIMINATECOL. Moreover, there exists a unimodular matrix L such that $L \cdot K = K^*$ and $\log\|L\| = O(\log\|K\|)$.

ALGORITHM 4.2.

INPUT: An $n \times n$ nonsingular matrix A.

OUTPUT: A matrix T the HNF of A and a matrix M such that $MA = T$.

begin

(1) $d \leftarrow \det(A)$;

(2) $K \leftarrow \begin{bmatrix} A \\ dI_n \end{bmatrix}$;

(3) *for* $i = 1$ *to* n *do*
begin
ELIMINATECOL* (K, d, i);

(4) *end*

$T^* \leftarrow [k_{ij}]$, $1 \leqq j \leqq n$, $1 \leqq i \leqq n$;

comment T^* is upper triangular now, but its off-diagonal entries do not satisfy the conditions of Theorem 4.1.

(4.1) $K \leftarrow \begin{bmatrix} T^* \\ dI_n \end{bmatrix}$;

ROW $(i) \leftarrow -$ROW (i) for each $k_{ii} < 0$;

(5) *for* $i = 2$ *to* n *do*

begin

(4.2) ROW $(j) \leftarrow$ ROW $(j) - (\lceil k_{ij}/k_{ii} \rceil)$ ROW (i) for $i \leqq j \leqq i$;

(4.3) ROW $(j) \leftarrow$ ROW $(j) - (\lfloor k_{jl}/d \rfloor)$ ROW $(n+l)$ for $1 \leqq j \leqq i$, $i \leqq l \leqq n$;

(6) *end*

comment The re-introduction of K at (4.1) was necessary, since the entries of K would grow exponentially at (4.2) without the reduction modulo d at (4.3).

$T \leftarrow [k_{ij}]$ for $1 \leqq i, j \leqq n$

(7) Solve the system: $X \cdot T = A$;

(8) return $M = X^{-1}$, T;

end

Remark 4.2. The employment of ELIMINATECOL* was necessary, since the use of ELIMINATECOL instead of ELIMINATECOL* would lead to an upper triangular of determinant qd, with q not necessarily equal to ± 1.

PROPOSITION 4.3. *Algorithm* 4.2 *correctly computes the* HNF *of* A *in* $O(n^2[n + \log(n \log(n\|A\|))]M(n \log(n\|A\|)))$ *elementary operations.*

Proof. First observe that there exists an integer matrix such that

$$WA = dI_n$$

and since every application of ELIMINATECOL* and every IRC operation corresponds to a multiplication of K by an $(m+n) \times (m+n)$ unimodular matrix, this implies the existence of a matrix L such that

$$L \begin{bmatrix} A \\ dI_n \end{bmatrix} = \begin{bmatrix} T^* \\ 0 \end{bmatrix},$$

and thus

$$L \begin{bmatrix} I_n & 0 \\ W & I_n \end{bmatrix} \begin{bmatrix} A \\ 0 \end{bmatrix} = \begin{bmatrix} T^* \\ 0 \end{bmatrix}.$$

Hence, if

$$\begin{bmatrix} Z_1 & Z_2 \\ Z_3 & Z_4 \end{bmatrix} := L \begin{bmatrix} I_n & 0 \\ W & I_n \end{bmatrix}$$

with Z_i matrices of dimension $n \times n$ for $1 \leqq i \leqq 4$, it then follows that

$$Z_1 A = T^*.$$

The matrices T^* and A represent the same group; hence $\det(T^*) = \det(A)$ and thus Z_1 is unimodular.

In a similar way we can show the existence of a unimodular matrix Z^* such that $Z^*T^* = T$ and thus prove that T is the HNF of A.

The computation of the determinant requires $O(n^3 M(n \log(n\|A\|)))$ elementary operations when we use Theorem 1.3 and Corollary 1.2(i).

Loop (3)-(4) requires $O(n^3M(\log n\|K\|)+n^2M(\log\|K\|)(\log\log\|K\|))$ elementary operations when we use Proposition 2.2.

We can show easily that loop (5)-(6) requires $O(n^3)$ multiplications comprising $O(n^3M(\log|d|))$ elementary operations.

The computation of X is not difficult since T is triangular and it requires $O(n^2)$ divisions comprising $O(n^2M(\log|d|))$ elementary operations.

The computation of the inverse of X requires $O(n^3M(n\log(n\|A\|)))$ elementary operations, by Theorem 1.3 and Corollary 1.2(i) (see the discussion below Theorem 1.3).

Hence, when we use

$$\|K\|\leqq\max\{\|A\|,|d|\}\leqq(n\|A\|)^n,$$

the proposition follows. □

COROLLARY 4.4. *There exists an algorithm for computing the* HNF *of a nonsingular matrix with integer entries of size s and the multiplier unimodular matrix M of Theorem* 4.1 *in* $O(s^3M(s^2))$ *elementary operations.*

In [2] Chou and Collins gave an $O(n^4(n+n\log(n\|A\|))^2)=O(s^8)$ elementary operations upper bound for computing the HNF of an integer matrix. This bound has been improved by a factor of $O(s^{3-\varepsilon})$ for any $\varepsilon>0$.

5. Smith normal form.

PROPOSITION 5.1 (Smith, see [11]). *Given a nonsingular* $n\times n$ *integer matrix A, there exist* $n\times n$ *unimodular matrices B, C such that* $D=BAC$ *is a diagonal matrix with positive diagonal elements such that* $d_{11}|d_{22}|\cdots|d_{nn}$.

From Corollary 3.3 we have the following.

PROPOSITION 5.2. *The* SNF *of an* $n\times n$ *matrix A can be computed in* $O(n^3[n+\log(n\log(n\|A\|))]\log(n\|A\|)M(n\log(n\|A\|)))$ *elementary operations.*

COROLLARY 5.3. *The* SNF *of a nonsingular matrix A of size s can be computed in* $O(s^5M(s^2))$ *elementary operations.*

PROPOSITION 5.4. *There exists an algorithm for computing the matrices B and C of Proposition* 5.1 *in* $O(n^{\alpha+2}\log(n\|A\|)\log(n\log(n\|A\|))M(n\log(n\|A\|)))$ *elementary operations, where* $O(n^\alpha)$ *multiplications are required for matrix multiplication.*

Proof. We can compute the matrix D the SNF of A using Algorithm 3.1. Each application of ELIMINATECOL in Algorithm 3.1 is equivalent to a pro-multiplication of K (as in (2.1)) by a unimodular matrix L_i (see Remark (2.2)) and each application of ELIMINATEROW is equivalent to L_jKR_i, where L_j, R_i are unimodular matrices (see Remark 2.6).

Hence there exist unimodular matrices $L_1,\cdots,L_\mu$ of dimension $2n\times 2n$ and $R_1,\cdots,R_\lambda$ of dimension $n\times n$ (corresponding to applications of ELIMINATEROW-ELIMINATECOL) such that

$$L_\mu\cdots L_1\begin{bmatrix}A\\ dI_n\end{bmatrix}R_1\cdots R_\lambda=\begin{bmatrix}D\\ dI_n\end{bmatrix}. \tag{5.1}$$

Let

$$L=L_\mu\cdots L_1=:\begin{bmatrix}\Lambda_1 & \Lambda_2\\ \Lambda_3 & \Lambda_4\end{bmatrix},$$

where Λ_i is an $n\times n$ matrix for $1\leqq i\leqq 4$, and $R=R_1\cdots R_\lambda$.

Moreover we can compute an $n\times n$ matrix W such that

$$WA=dI_n \tag{5.2}$$

by computing the inverse of A.

From (5.1), (5.2) it follows that

$$L\begin{bmatrix} I_n & 0 \\ W & I_n \end{bmatrix}\begin{bmatrix} A \\ 0 \end{bmatrix}, \qquad R = \begin{bmatrix} D \\ dI_n \end{bmatrix},$$

which implies that

$$(\Lambda_1 + \Lambda_2 W)AR = D$$

and the matrix $\Lambda_1 + \Lambda_2 W$ is unimodular, since $\det(R) = 1$ and $\det(A) = \det(D)$ (from Proposition 5.2). Hence in order to compute the matrices B and C, it suffices to compute the matrices L, R, and W.

Proposition 3.2 and Remarks 2.2, 2.6, and 2.7 yield the running time required for the computation of the L_i's and R_i's. Moreover the computation of W requires $O(n^\alpha)$ multiplications (see comments below Theorem 1.3).

Hence the analysis of the computation of L and R only remains. Using (3.1), we can show

$$\max\{\lambda, \mu\} = O(n \log |d|) = O(n^2 \log (n\|A\|)), \tag{5.3}$$

and using Remarks 2.2 and 2.7 we can show that

$$Q := \max_{i,j}\{\|L_i\|, \|R_j\|\} \leqq \max\{\|A\|, |d|\} \leqq (n\|A\|)^n. \tag{5.4}$$

The computation of L is done as follows. Let $L_\mu, \cdots, L_1$ be the leaves of a balanced binary tree and without loss of generality assume that μ is a power of 2. Moreover, let each parent node represent the product (in order) of its children. Then the root represents L.

We can observe that the entries of a matrix at the ith level of the tree is at most Q^{2^i} in absolute value, and the computation of the root requires $O(\sum_{i=1}^{h}(\mu/2^i))$ matrix multiplications, where $h = O(\log \mu)$ is the height of the tree. Therefore the computation of L requires

$$O\left(n^\alpha \sum_{i=1}^{h}\left(\frac{\mu}{2^i}\right)M(2^i \log Q)\right) \quad \text{elementary operations,}$$

comprising $O(n^{\alpha+2} \log (n\|A\|) \log (n \log (n\|A\|))M(n \log (n\|A\|)))$ elementary operations, using (5.3) and (5.4). The computation of R is done in a similar way.

COROLLARY 5.5. *Let the matrices A, B, and C be as in Proposition* 5.1. *Given A, there exists an algorithm for computing the matrices B and C in $O(s^{5.376} \log sM(s^2))$ elementary operations, where s is the size of A.*

Acknowledgments. The author owes a very significant debt to Dr. W. M. Beynon for his direct and indirect aid in this research. Thanks to J. D. Dixon for helpful discussions and for pointing out reference [2]. Thanks also to the referee for pointing out various inaccuracies in earlier versions of the paper and for his helpful suggestions.

REFERENCES

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

[2] T. J. CHOU AND G. E. COLLINS, *Algorithms for the solution of systems of linear diophantine equations*, SIAM J. Comput., 11 (1982), pp. 687–708.

[3] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 1-6.
[4] F. R. GANTMACHER, *Matrix Theory*, Vol. 1, Chelsea, New York, 1960, p. 26.
[5] C. HERMITE, *Sur l'introduction des variables continues dans la théorie des nombres*, J. R. Augeur, Math., 41 (1851), pp. 191-216.
[6] 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.
[7] D. KNUTH, *Seminumerical Algorithms*, 2nd edition, Addison-Wesley, Reading, MA, 1974.
[8] C. C. SIM, *The influence of computers in algebra*, Proc. Sympos. Appl. Math., 20 (1974), pp. 13-30.
[9] A. SCHÖNHAGE, *Schnelle Berechnung vor Kettenbruchentwicklungen*, Acta Inform., 1 (1971), pp. 139-144.
[10] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation Grosser Zahlen*, Computing, 7 (1971), pp. 281-292.
[11] H. J. S. SMITH, *On systems of intermediate equations and congruences*, Philos. Trans., 151 (1861), pp. 293-326.
[12] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354-356.
[13] S. WINOGRAD, *The algebraic complexity of functions*, in Proc. International Congress of Mathematicians, Vol. 3, 1970, pp. 283-288.

SIAM J. COMPUT
Vol. 18, No. 4, pp. 670-678, August 1989

003

WORST-CASE COMPLEXITY BOUNDS ON ALGORITHMS FOR COMPUTING THE CANONICAL STRUCTURE OF INFINITE ABELIAN GROUPS AND SOLVING SYSTEMS OF LINEAR DIOPHANTINE EQUATIONS*

COSTAS S. ILIOPOULOS†

Abstract. An $O(s^5M(s^2))$ elementary operations algorithm for computing the canonical structure of infinite Abelian groups represented by a matrix of size s is presented. Also given is an algorithm for solving systems of linear Diophantine equations (say, $Ax=b$) in $O(s^{3.376}\log sM(s^2)+s^2M(s^*))$ elementary operations, where s is the size of A and s^* is the size of b. The upper bounds mentioned above improve the results given by Chou and Collins in [SIAM *J. Comput.*, 11 (1982), pp. 687-708].

Key words. complexity, Abelian group, Hermite normal form, Smith normal form, Diophantine equations

AMS(MOS) subject classifications. 15A21, 20K01, 20K05

1. Introduction. Recently in [1] Chou and Collins gave an algorithm for transforming an integer matrix via integer row-column operations (abbreviated IRC operations) into a diagonal one whose nonzero diagonal entries have the property of a Smith normal form matrix. They make use of an improved version of the Kannan-Bachem polynomial algorithm (see [8]).

Here the equivalent problem of computing the canonical structure of an infinite Abelian group represented by a set of defining relations is considered. Let A be the $m\times n$ matrix of the coefficients of a set of defining relations for an Abelian group G and let $s=m+n+\log\|A\|$, with $\|A\|=\max_{i,j}\{|a_{ij}|\}$ be the size of A. An algorithm for computing the canonical structure of G in $O(s^5M(s^2))$ elementary operations is presented here, where $M(k)$ denotes an upper bound on the number of elementary operations required for multiplication of two integers of length of k bits (see [9]).

From [1] we can derive an upper bound of $O(s^{11})$ for computing the canonical structure of an infinite Abelian group; this bound has been improved by a factor of $O(s^{3-\varepsilon})$.

Moreover, in [1] an algorithm for solving a system of linear Diophantine equations (say, $Ax=b$) is given, which requires $O(s^8+s^4s^*)$ elementary operations, where s is the size of A and s^* is the size of b. An algorithm for solving the above system in $O(s^{3.376}\log sM(s^2)+s^2M(s^*))$ elementary operations is presented here; in this case the bound is improved by a factor of at least $O(s^2)$.

Both the algorithms presented here make use of procedures and algorithms given in [7].

2. Preliminaries. The algorithms and theorems presented in this section form the necessary background for proving the main results of this paper.

THEOREM 2.1 (see [7]). *Suppose that $m\times n$ integer matrix A with* rank $(A)=n$, *an integer ρ such that $1\leqq\rho\leqq n$ and d the determinant of an $n\times n$ nonsingular submatrix of*

* Received by the editors December 1, 1983; accepted for publication (in revised form) March 3, 1988. This research was supported by the Greek section of scholarships and research's Alexander S. Onassis public benefit foundation.

† Department of Computer Science, University of Warwick, Coventry CV4 7AL, United Kingdom. Present address, Royal Holloway College, University of London, Department of Computer Science, Egham TW20 0EX, United Kingdom.

A are given. Moreover, let

$$K = \begin{bmatrix} A \\ dI_n \end{bmatrix}.$$

Then there exists an algorithm (namely procedure ELIMINATECOL* (K, D, ρ)*) for transforming K, via integer row operations, into a matrix* K^* *of the form*

$$\begin{bmatrix} & A' & \\ dI_{\rho-1} & 0 & 0 \\ & \vdots & \\ 0 & 0 & dI_{n-\rho} \end{bmatrix}, \tag{2.1}$$

where the ρ*th column of* A' *is of the form* $(\alpha'_{1\rho}, \cdots, \alpha'_{\rho\rho}, 0, \cdots, 0)^T$*, that requires* $O(mnM(\log m \| K \|) + mM(\log \| K \|) \log\log \| K \|)$ *elementary operations and* $\| K^* \| = O(|d|)$*. It also computes a unimodular matrix L such that*

$$L \cdot K = K^* \quad \textit{with } \log \| L \| = O(\log \| K \|).$$

Now we consider the following problem.

Suppose that A is an $m \times n$ integer matrix with rank n. Compute a unimodular $(m+n) \times (m+n)$ matrix L such that

$$L \begin{bmatrix} A \\ 0_n \end{bmatrix} = \begin{bmatrix} T \\ 0 \end{bmatrix}, \tag{2.2}$$

where T is an upper triangular $n \times n$ matrix, and zero represents an $m \times n$ zero matrix.

ALGORITHM 2.2.

INPUT: An $m \times n$ matrix A with integer entries and rank $(A) = n$.

Output: A matrix L satisfying (2.2).

begin

(1) $B \leftarrow$ an $n \times n$ nonsingular submatrix of A;

(2) $d \leftarrow \det(B)$;

(3) Transform A via row interchanges to a matrix that the n top rows form the matrix B;

(4) Let L' be the unimodular matrix corresponding to the above transformation;

(5) $W \leftarrow (dI_n)B^{-1}$;

$$L^* \leftarrow \begin{bmatrix} I_n & 0 & 0 \\ 0 & I_{m-n} & 0 \\ W & 0 & I_n \end{bmatrix};$$

(6) $K \leftarrow L^* \begin{bmatrix} A \\ 0_n \end{bmatrix}$;

comment Note that $K = \begin{bmatrix} A \\ dI_n \end{bmatrix}$.

(7) *for* $i = 1$ *to* n *do*

begin

(8) ELIMINATECOL* $(K, d, 1)$

Let L_i be the corresponding unimodular matrix as in Theorem 2.1;

$$L_i \leftarrow \begin{vmatrix} I_{i-1} & 0 \\ 0 & L_i \end{vmatrix};$$

$K \leftarrow [K_{lj}]$ for $2 \leqq l \leqq n-i+1$, $2 \leqq j \leqq n-i+1$;

(9) *end*

(10) $L \leftarrow L_n L_{n-1} \cdots L_1 L^* L'$;
end

PROPOSITION 2.3. *Algorithm* 2.2 *correctly computes* L *in*

$$O(n[n^\alpha \log n + mn + m \log (n \log (n\|A\|))]M(n \log (m\|A\|)))$$

elementary operations, where $O(n^\alpha)$ *denotes an upper bound on the number of multiplications required for multiplication of two* $n \times n$ *matrices. Moreover,*

$$\log \|L\| = O(n^2 \log (n\|A\|)).$$

Proof. The computation of the determinant d and the matrix B can be done by means of the Gaussian elimination in $O(mn^2 M(\log d^*))$ elementary operations, where $d^* = \max\{|\delta|: \delta$ is the determinant of a square submatrix of $A\}$ (see Gantmacher [5, p. 26] and the remark below).

Step (3) requires at most $O(n^2)$ elementary operations for row interchanges and it is not difficult to show that

$$\|L'\| = 1.$$

Step (5) requires $O(n^3 M(\log d^*))$ elementary operations for computing the inverse of B and $W = dI_n B^{-1}$. Moreover, we can show that

$$\|W\| = O(d^*).$$

Therefore

$$\|L^*\| = O(d^*).$$

Step (6) merely introduces dI_n in the bottom n rows of the matrix $\begin{bmatrix} A \\ 0_n \end{bmatrix}$ of (2.2).

Using Theorem 2.1 we can find that loop (7)–(9) requires $O(mn^2 M(\log m\|K\|) + mnM(\log \|K\|)\log\log \|K\|)$ elementary operations.

The computation of L at Step (9) can be done as follows. Let the L_i's, L^*, and L' be the leaves of a balanced binary tree (without loss of generality we assume that the number of matrices is a power of two). Moreover, let each parent node represent the product of its children (in order). Then the root of the tree represents L. It is not difficult to see that the computation of all nodes of the tree require

$$O\left(\sum_{i=1}^{\lceil \log n \rceil} \left(\frac{n}{2^i}\right)\right)$$

matrix multiplications. From Theorem 2.1 it follows that $\|L_i\| = O(\|K\|^c)$ for some constant $c > 0$ for all i's; it is not difficult to show that the matrices of the ith level of the tree have entries of $O(\|K\|^{c2^i})$. Therefore the computation of L requires

$$O\left(n^\alpha \sum_{i=1}^{\lceil \log n \rceil} \left(\frac{n}{2^i}\right) M(2^i \log \|K\|)\right) = O(n^{\alpha+1} \log n M(nM(n \log (n\|A\|))))$$

elementary operations.

Now, using the fact that

$$\max\{\|K\|, d^*\} = O(n^n \|A\|^n),$$

the upper bound on the running time of Algorithm 2.2 follows and

$$\|L\| = O(\|K\|^{c2^{\log n}}) = O(\|K\|^{cn}) = O(n^{cn^2}\|A\|^{cn^2}).$$ □

Remark 2.1. Above we make use of the Gaussian elimination procedure for computing the determinant of a matrix and for computing the inverse of a matrix. Strassen in [11] and Coppersmith and Winograd in [2] give asymptotically faster methods for these computations. Here and in the sequel [7] we shall make use of the Gaussian elimination instead of the Coppersmith-Winograd algorithm (currently the fastest asymptotic method) when the computation of the determinant (or inverse) is not the bottleneck of the algorithm. This is because the Coppersmith-Winograd algorithm supersedes the Gaussian elimination only in the case of very large dimension matrices.

Remark 2.2. Suppose that A is the input matrix of Algorithm 2.2 and L is the output matrix. Let

$$L=\begin{bmatrix}\Lambda_1 & \Lambda_2\\ \Lambda_3 & \Lambda_4\end{bmatrix},$$

where Λ_1 is an $m\times m$ matrix, Λ_2 is an $m\times n$ matrix, Λ_3 is an $n\times m$ matrix, and Λ_4 is an $n\times n$ matrix.

Then

$$\Lambda_1 A=\begin{bmatrix}T\\ 0\end{bmatrix} \quad \text{where zero is an } m\times n \text{ zero matrix,}$$

but Λ_1 is not necessarily unimodular. The author is not aware of an efficient way of computing a unimodular matrix Λ such that

$$\Lambda A=\begin{bmatrix}T\\ 0\end{bmatrix} \quad \text{where zero is an } m\times n \text{ zero matrix.}$$

This will lead to some inelegancies in the presentation of algorithms for computing the structure of infinite Abelian groups and for solving systems of linear Diophantine equations.

PROPOSITION 2.4 (see [7, Cor. 3.3]). *There exists an algorithm for computing the canonical structure of a finite Abelian group G represented by an $m\times n$ integer matrix A in $O(mn^2(n+\log(n\log(n\|A\|)))\log(n\|A\|)M(n\log(mn\|A\|)))$ elementary operations.*

3. Infinite Abelian groups. An algorithm for computing the canonical structure of an infinite Abelian group represented by a matrix is presented below. The basic idea involved is the computation of a matrix representing the subgroup of all the finite-order elements of the group, and then the algorithm for computation of the canonical structure of finite Abelian groups given in [7] (see Proposition 2.4) is applied.

ALGORITHM 3.1.

INPUT: An $m\times n$ matrix A representing the infinite Abelian group G.

OUTPUT: The canonical structure of G.

begin

(1) $r\leftarrow \text{rank}(A)$;

(2) $B\leftarrow$ an $r\times r$ nonsingular submatrix of A;

(3) $A\leftarrow\begin{bmatrix}B & C\\ D & E\end{bmatrix}$;

comment The matrix D is an $(n-r)\times r$ matrix, C is an $r\times(n-r)$ matrix, and E is an $(m-r)\times(n-r)$ matrix. The transformation is done with column and row interchanges in such a way that the r top and the r

left-row columns form the matrix B in the left top corner of A.

$$A \leftarrow \left[\frac{A}{0_r 0}\right];$$

comment Note that the addition of trivial relations to the set of defining relations of G does not change its structure.

(4) Compute a unimodular $(m+r)\times(m+r)$ matrix L such that:

$$L\begin{vmatrix} B \\ D \\ 0_r \end{vmatrix} = \begin{vmatrix} V \\ 0 \end{vmatrix} \quad \text{where } V \text{ is an } r\times r \text{ upper triangular matrix;}$$

comment We can use Algorithm 2.2, since

$$\operatorname{rank}\left(\begin{vmatrix} B \\ D \\ 0_r \end{vmatrix}\right) = r.$$

(5) $A \leftarrow LA$;

(6) Let $A = \left[\begin{smallmatrix} V & M \\ 0 & 0 \end{smallmatrix}\right]$;

comment See the proof of Proposition 3.2, part (ii).

(7) *for* $i = r$ *down to* 1 *do*

begin

(8) $\mathrm{COL}(j) \leftarrow \mathrm{COL}(j) - (\lfloor m_{ij}/t_{ii} \rfloor)\,\mathrm{COL}(i)$ for $r+1 \leqq j \leqq n$;

comment We reduce M's entries modulo the diagonal elements

(9) *end*

(10) $A^* \leftarrow [V, M]^T$

comment A^* is an $n\times r$ matrix with rank r and thus represents a finite Abelian group, say, G^*.

(11) Compute the canonical structure of G^* using the algorithm of Proposition 2.4.

end

PROPOSITION 3.2. *Algorithm* 3.1 *correctly computes the canonical structure of* G.

Proof. (i) All steps are expressed in terms of IRC operations, which respect the structure of the group G.

(ii) In step (6) the bottom right corner $(m-r)\times(m-r)$ submatrix of A is a matrix where all of its entries are zeros, because if its jth column had a nonzero entry for some j, then A would have $r+1$ linearly independent columns (the first r columns of A and the jth of the submatrix), which contradicts the fact that $r = \mathrm{rank}(A)$.

(iii) In step (10), A^* represents the finite group G^*. Let G' be the maximal finite subgroup of G. It will be shown that G^* is isomorphic to G'. Let $A_1 = [V, M]$. Since A_1 represents G, there exists unimodular matrices L and R such that

$$LA_1R = D \tag{3.1}$$

where D represents the canonical structure of G. Moreover, let

$$G' = C(d_{11})\times\cdots\times C(d_{rr}), \tag{3.2}$$

where $C(k)$ represents a cyclic group of order k.

From (3.1) it follows that

$$R^T A_1^T L^T = D \quad \text{or} \quad R^T A^* L^T = D^T$$

and D^T represents the canonical structure of G^*. Therefore

$$G^* = C(d_{11}) \times \cdots \times C(d_{rr}),$$

and using (3.2) it follows that $G^* \approx G'$. □

PROPOSITION 3.3. *Algorithm* 3.1 *computes the canonical structure of* G *in*

$$O(r[r^\alpha \log r + mn + nr \log (r\|A\|)(r + \log (r \log (r\|A\|)))]$$
$$\cdot M(r \log (n\|A\|) + (r^2 n + mn) M(r^2 \log (r\|A\|))))$$

elementary operations.

Proof. Step (1)-(2) requires $O(mnrM(\log |d^*|))$ elementary operations by means of Gaussian elimination, where $|d^*| = \max \{|d|: d$ is the determinant of a square submatrix of $A\}$. Moreover using Hadamard's bound on the size of a determinant of a square matrix, it follows that

$$|d^*| \leqq (r\|A\|)^r.$$

The running time of step (4) is given by Proposition 2.3.

Step (5) requires $O(mn)$ multiplications comprising $O(mnM(r^2 \log (r\|A\|)))$ elementary operations, using that $\log \|L\| = O(r^2 \log (r\|A\|))$ by Proposition 2.3. Loop (7)-(9) requires $O(r^2 n)$ multiplications comprising $O(r^2 nM(r^2 \log (r\|A\|)))$ elementary operations.

Using Theorem 2.1 (bound on $\|T\|$), we can see that after the reduction modulo the t_{ii}'s, it follows that

$$\|A^*\| \leqq \|T\| \leqq (r\|A\|)^r. \tag{3.3}$$

From Proposition 2.4 and (3.3) we can derive the running time of step (11) comprising $O(nr^2 \log (r\|A\|)[r + \log (r \log (r\|A\|))]M(r \log (n\|A\|)))$ elementary operations. □

COROLLARY 3.4. *There exists an algorithm computing the canonical structure of an infinite Abelian group* G *represented by a matrix* A *of size* s, *in* $O(s^5 M(s^2))$ *elementary operations.*

Proof. From Proposition 3.3 and using the fact that $\alpha \leqq 3$ ($\alpha = 3$ in the classical algorithm for matrix multiplication). □

4. Linear Diophantine equations. Let A be an $m \times n$ matrix with integer entries and b an $n \times 1$ vector with entries from the integers. Then the system of equations

$$Ax = b, \qquad x \in \mathbb{Z}^{n \times 1} \tag{4.1}$$

is called a *system of linear Diophantine equations.*

The computation of a solution or all (if any) of the system (4.1) is closely related to the triangularization of the matrix A. If the matrix A of (4.1) is of rank n, then (4.1) has exactly one solution or none and this can be found by means of Gaussian elimination. In the case in which the rank r of A is less than n, the system (4.1) has an infinite number of solutions ($n - r$ linearly independent solutions) or the system is inconsistent. In this case the classical algorithm for solution of a system of linear Diophantine equations makes use of the classical triangularization algorithm (see Smith [10]) and therefore has the problem of "Intermediate expression swell."

The first polynomial algorithms for solution of (4.1) were given by Frumkin in [3] (see also [4]) in some special cases. Frumkin's algorithm for computing a *particular* solution of (4.1) or establishing that there is not one requires in the worst case

$$O(n^2 m^2 \log (n\|A\|) M(n \log (n\|A\|)) + n^2 M(n \log (n\|A\|) + \log \|b\|))$$
$$= O(s^5 M(s^2) + s^2 M(s^*)),$$

where s is the size of the matrix A and s^* is the size of the vector b. Moreover, in (4) Frumkin gave an algorithm for computing the set of all solutions (if any) of an *homogeneous system* of linear Diophantine equations (that is, (4.1) with $b=(0,\cdots,0)$), which requires

$$O(n^3 m \log(n\|A\|)M(m \log m \log(m\|A\|))) = O(s^5 M(s^2 \log s))$$

elementary operations.

The best-known polynomial algorithm for solving a general system of linear Diophantine equations is given by Chou and Collins in [1]. Their algorithm requires

$$O(n^3(m+n)[n+r\log(r\|A\|)]^2 + r(m+n)\log\|b\|[n+r\log(n\|A\|)]) = O(s^8 + s^4 s^*)$$

elementary operations for the computation of the set of all the solutions of (4.1), if any, where s is the size of A and s^* is the size of the vector b.

An algorithm for solving (4.1) whose upper bound improves the upper bounds of Frumkin and Chou and Collins is presented below.

ALGORITHM 4.1.

INPUT: The system of equations (4.1).

OUTPUT: A set of all integral solutions of the system (4.1), if any.

begin

(1) $r \leftarrow \text{rank}(A)$;

(2) $B \leftarrow$ an $r \times r$ nonsingular submatrix of A;

(3) $A \leftarrow [A \mid \begin{smallmatrix}0\\0\end{smallmatrix}_r]$;

(4) compute unimodular matrices L_1, R_1 such that $L_1 A R_1 = [\begin{smallmatrix}B & C\\D & E\end{smallmatrix}]$;

comment The matrix D is an $(m-r)\times r$ matrix, C is an $r\times(n-r)$ matrix, and E is an $(m-r)\times(n-r)$ matrix. The transformation is done with column and row interchanges in such a way that the top left $r\times r$ submatrix of A is the matrix B.

(5) $A \leftarrow L_1 A R_1$;

(6) $b \leftarrow L_1 \cdot b$;

(7) $H \leftarrow [B\ 0_r]^T$;

(8) Compute a square unimodular matrix L: $LH = |\begin{smallmatrix}U\\0\end{smallmatrix}|$,
where U is an $r\times r$ upper triangular matrix.

comment We may use Algorithm (2.2), since rank $(H)=r$ equal to the number of columns of H.

$R_2 \leftarrow L^T$;

(9) $A \leftarrow AR_2$;

(10) $U \leftarrow U^T$;

(11) Let $A = |\begin{smallmatrix}U & 0\\M & 0\end{smallmatrix}|$;

comment M denotes the $(n-r)\times r$ submatrix at the bottom left of the matrix A of Step (9).

(12) compute a particular solution $\vec{W}$ of the system $Ax=b$ over the rationals;

comment The computation of $\vec{W}$ is done by means of U^{-1} (note that U is triangular).

if $\vec{W}$ has noninteger entries *then*

return "The system (4.1) has not any integral solutions"

else

$\vec{V} \leftarrow$ the first n coordinates of $\vec{W}$;

(13) $R \leftarrow R_1 \cdot R_2$;

(14) $R' \leftarrow R^{-1}$;

$R^* \leftarrow [r_{ij}]$, $r+1 \leqq j \leqq n+r$, $1 \leqq i \leqq n$;

return $\{\vec{x} = R'\vec{V} + R^*(t_1, \cdots, t_n)$;
comment The t_i's are free variables.
end

PROPOSITION 4.2. *Algorithm* 4.1 *correctly computes a solution of the system* (4.1), *if any.*

Proof. Let A' denote the matrix A at Step (3) and let A'' denote the matrix A at Step (11). Then we have

$$A'' = A' \cdot R. \tag{4.2}$$

Let $\vec{z} = \vec{v} + (y_1, \cdots, y_r, t_1, \cdots, t_n)^T$ be the set of all integral solutions of the system

$$A'' \cdot \vec{z} = b. \tag{4.3}$$

Then we have

$$A''\vec{z} = A''\vec{v} + A''(y_1, \cdots, y_r, t_1, \cdots, t_n)^T. \tag{4.4}$$

When we use (4.3), (4.4), and the fact that $A\vec{v} = b$ follows

$$A''(y_1, \cdots, y_r, t_1, \cdots, t_n)^T = 0$$

or equivalently (see Step (11))

$$\begin{bmatrix} U & 0 \\ M & 0 \end{bmatrix} (y_1, \cdots, y_r, t_1, \cdots, t_n)^T = 0,$$

implies that $y_i = 0$ for $1 \leqq i \leqq r$.

Now let $\vec{x} = R'\vec{z}$. Then $\vec{x}$ yields the set of all integral solutions for the system

$$A'\vec{x} = b,$$

since $A'\vec{x} = A''RR^{-1}z = A''z = b$, when we use (4.2) and (4.3). Moreover,

$$\vec{x} = R'\vec{v} + R'(0, \cdots, 0, t_1, \cdots, t_n)^T = R'\vec{v} + R^*(t_1, \cdots, t_n)^T. \qquad \square$$

PROPOSITION 4.3. *Algorithm* 4.1 *correctly computes the set of all integral solutions of* (4.1) *in* $O(r[r^\alpha \log r + mn + n \log (r \log (r\|A\|))]M(r \log (n\|A\|)) + mnM(r^2 \log (r\|A\|) + \log \|b\|))$ *elementary operations.*

Proof. Steps (1)-(5), (7)-(11) require

$$O(r[r^\alpha \log r + mn + n \log (r \log (r\|A\|))]M(r \log (n\|A\|)) + mnM(r^2 \log (r\|A\|)))$$

elementary operations. Their analysis is the same as in Proposition 3.3 (Steps (1)-(5)).

Step (6) requires $O(n \log \|b\|)$ elementary operations when we use the fact that the rows of L_1 are of the form $(0, \cdots, 0, 1, 0, \cdots, 0)$.

Step (12) requires $O(mn)$ divisions/multiplications comprising $O(mnM(r^2(\log (r\|A\|)) + \log \|b\|)$ elementary operations, since if A^* denotes the matrix A at step (8) then

$$\log \|A^*\| = \log \|L_1 A R_1 R_2\| = O(r^2 \log (r\|A\|))$$

when we use $\|L_1\| = \|R_2\| = 1$ and $\|R_2\| \leqq \|A\|^{cr^2}$ by Proposition 2.3, for some constant $c > 0$.

Step (13) requires only $O(mn)$ elementary operations for column interchanges on R_2. Step (13) makes use of the Gaussian elimination (or the Coppersmith-Winograd algorithm).

The proposition follows from the above. $\square$

COROLLARY 4.4. *There exists an algorithm for computing the set of all solutions, if any, of a linear Diophantine system* (4.1) *in*

$$O(s^{\alpha+1} \log sM(s^2) + s^2 M(s^*))$$

elementary operations, where s is the size of A and s^* *is the size of b.*

Acknowledgment. I thank the referee for pointing out various errors and for his helpful suggestions.

REFERENCES

[1] T. J. CHOU AND G. E. COLLINS, *Algorithms for the solution of systems of linear Diophantine equations*, SIAM J. Comput., 11 (1982), pp. 687-708.

[2] D. COPPERSMITH AND G. WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 1-6.

[3] M. A. FRUMKIN, *Polynomial time algorithms in the theory of linear Diophantine equations*, in Fundamentals on Computation Theory, M. Karpinski ed., Lecture Notes in Computer Science 56, Springer-Verlag, New York, 1977, pp. 386-392.

[4] ———, *An application of modular arithmetic to the construction of algorithms for solving systems of linear equations*, Soviet Math. Dokl., 17 (1976), pp. 1165-1169.

[5] F. R. GAUTMACHER, *Matrix Theory*, Vol. I, Chelsea, New York, 1960.

[6] C. S. ILIOPOULOS, *Algorithms in the theory of Abelian groups*, Ph.D. thesis, Warwick University, Coventry, United Kingdom, 1983.

[7] ———, *Worst-case complexity bounds on algorithms for computing the canonical structure of finite Abelian groups and Hermite and Smith normal form of an integer matrix*, SIAM J. Comput., this issue, pp. 658-669.

[8] 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.

[9] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation Grosser Zahlen*, Computing, 7 (1971), pp. 281-292.

[10] D. A. SMITH, *A basis algorithm for finitely generated abelian groups* in Math. Alg. Vol. 1, 1966, pp. 13-66.

[11] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354-356.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 679-689, August 1989

004

ON THE COMPLEXITY OF PARTIAL ORDER PRODUCTIONS*

ANDREW CHI-CHIH YAO†

Abstract. Let $P=(<_P, Y)$ be a partial order on a set $Y=\{y_1, y_2, \cdots, y_n\}$ of n elements. The problem of *P-production* is as follows: Given an input of n distinct numbers $x_1, x_2, \cdots, x_n$, find a permutation σ of $(1, 2, \cdots, n)$ such that $y_i <_P y_j$ implies $x_{\sigma(i)} < x_{\sigma(j)}$. Let $C(P)$, $\bar{C}(P)$ be, respectively, the minimum number and the minimum average number of binary comparisons $x_i : x_j$ needed by any decision-tree algorithm to produce P. It is proved that $C(P)=\Theta(\bar{C}(P))$. As an intermediate result, it is shown that $C(P)=O(\log_2(n!/\mu(P))+n)$, where $\mu(P)$ is the number of permutations consistent with P, thus proving a conjecture of Saks.

Key words. algorithm, complexity, decision tree, Dilworth's theorem, partial order, permutation

AMS(MOS) subject classifications. 06A10, 68P10, 68Q25

1. Introduction. Sorting and median-finding of a set of n numbers are two of the classical problems in combinatorial computation. It is well known (see Knuth [Kn, § 5.3]) that sorting n numbers takes asymptotically $\Theta(n \log n)$ binary comparisons of the form $x_i : x_j$, both in the worst case and in the average case. For median-finding, it was first proved that the average-case complexity is $\Theta(n)$ (Floyd and Rivest [FR]), and later it was discovered that the worst-case complexity is also $\Theta(n)$ (Blum et al. [BFPRT]). Thus, in both problems, the worst-case complexity and the average-case complexity are of the same order of magnitude. Are they special cases of a general class of problems for which this phenomenon is true? In this paper we will show that this is indeed so.

Let $P=(<_P, Y)$ be a partial order on a set $Y=\{y_1, y_2, \cdots, y_n\}$. The *P-production problem* is the following. Given n distinct numbers $x_1, x_2, \cdots, x_n$, find a permutation σ of $(1, 2, \cdots, n)$ such that $y_i <_P y_j$ implies $x_{\sigma(i)} < x_{\sigma(j)}$. We are interested in the intrinsic complexity of this problem in the decision tree model. Clearly, sorting and median-finding are both special cases of the P-production problem.

A *decision tree* T is a binary tree, each of whose internal nodes u contains a comparison of the form $x_i : x_j$, and has two outgoing edges labeled by "$<$" and "$>$"; associated with each leaf l is a permutation σ_l of $(1, 2, \cdots, n)$. Given any input $\tilde{x}=(x_1, x_2, \cdots, x_n)$ of distinct numbers, we traverse a path $\xi(T, \tilde{x})$ in T from the root down, making comparisons and branching according to the outcomes, until a leaf $l_{\tilde{x}}$ is reached. We call T an *algorithm* for P-production if, for every $\tilde{x}$, $y_i <_P y_j$ implies $x_{\rho(i)} < x_{\rho(j)}$, where $\rho=\sigma_l$. Let cost $(T, \tilde{x})$ denote the number of comparisons made by T along the path $\xi(T, \tilde{x})$, and let cost (T) be $\max_{\tilde{x}}$ cost $(T, \tilde{x})$. Denote by $\mathscr{A}_P$ the family of all algorithms for P-productions. The *minimax complexity* $C(P)$ of P-production is defined as $\min\{\text{cost}(T) \mid T \in \mathscr{A}_P\}$.

Let Γ_n be the set of all permutations of $(1, 2, \cdots, n)$. A permutation ρ is said to be *consistent* with P, if $y_i <_P y_j$ implies $\rho(i) < \rho(j)$ for all i, j. Let $\Delta(P) \subseteq \Gamma_n$ be the set of all permutations consistent with P, and define $\mu(P)=|\Delta(P)|$.

The complexity problem of P-production was formulated and investigated by Schönhage [Sch], who showed by an information-theoretic argument that $C(P) \geqq \log_2(n!/\mu(P))$. Further results on this problem have been derived by Aigner [Ai].

* Received by the editors April 5, 1988; accepted for publication (in revised form) October 11, 1988. This research was supported in part by National Science Foundation grant CCR-8813283.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

Saks has conjectured [Sa] that Schönhage's lower bound can be achieved asymptotically, in the sense that $C(P) = O(\log_2(n!/\mu(P)) + n)$.

For any $T \in \mathscr{A}_P$, the *average cost* of T is defined as $\text{cost}'(T) = 1/n! \sum_{\rho \in \Gamma_n} \text{cost}(T, \tilde{x}_\rho)$, where $\tilde{x}_\rho = (\rho(1), \rho(2), \cdots, \rho(n))$. The *minimean complexity* of P-production is defined as $\bar{C}(P) = \min\{\text{cost}'(T) | T \in \mathscr{A}_P\}$.

A partial order $P = (<_P, Y)$ is said to be *connected*, if for every two distinct elements y and y' in Y, there exists a sequence $y = y_1, y_2, \cdots, y_m = y'$ such that $y_i <_P y_{i+1}$ or $y_i >_P y_{i+1}$ for all i. Every partial order P can be uniquely decomposed into the disjoint union of connected partial orders $P_i = (<_{P_i}, Y_i)$, where the sets Y_i form a partition of Y. Let $\beta(P)$ denote the number of connected components in this decomposition.

In this paper we will prove the following results.

THEOREM 1. *For all* P, $\bar{C}(P) = \Omega(n - \beta(P) + \log_2(n!/\mu(P)))$.

THEOREM 2. *For all* P, $C(P) = O(n - \beta(P) + \log_2(n!/\mu(P)))$.

THEOREM 3. *For all* P, $C(P) = \Theta(\bar{C}(P))$.

Theorem 2 proves the conjecture of Saks [Sa] mentioned earlier. Since $C(P) \geqq \bar{C}(P)$ by definition, Theorem 3 is an immediate consequence of Theorems 1 and 2. The rest of this paper is devoted to a proof of Theorem 1 and Theorem 2.

2. Proof of Theorem 1. We will prove two lemmas. The first is an extension of Schönhage's lower bound on the minimax complexity $C(P)$ to the minimean complexity.

LEMMA 1. $\bar{C}(P) \geqq \log_2(n!/\mu(P))$.

Proof. Let $T \in \mathscr{A}_P$. We will prove that

$$\text{cost}'(T) \geqq \log_2\left(\frac{n!}{\mu(P)}\right). \tag{1}$$

For each leaf l of T, let Q_l be the partial order on X generated by the constraints $x_i > x_j$ along the path from the root to l. As $T \in \mathscr{A}_P$, each Q_l contains an isomorphic copy of P as a subpartial order. This implies that $\mu(Q_l) \leqq \mu(P)$. Let $q_l = \mu(Q_l)/n!$ Then

$$q_l \leqq \frac{\mu(P)}{n!}. \tag{2}$$

If we consider a random input $\tilde{x}_\rho = (\rho(1), \rho(2), \cdots, \rho(n))$, where ρ is uniformly chosen from Γ_n, then q_l is the probability that the traversed path $\xi(T, \tilde{x}_\rho)$ in T will end in the leaf l. Let d_l be the distance from the root to l. Then,

$$\sum_l q_l = 1, \tag{3}$$

$$\text{cost}'(T) = \sum_l q_l d_l. \tag{4}$$

It follows from (3) and (4) that $\text{cost}'(T)$ is the expected length of a uniquely decipherable code for an alphabet with symbol frequencies q_l. It is a well-known fact (see, e.g., [Ab, § 4.1]) in information theory that a lower bound is given by the entropy, that is,

$$\text{cost}'(T) \geqq \sum_l q_l \log_2 \frac{1}{q_l}.$$

Inequality (1) now follows from (2) and (3). □

LEMMA 2. $\bar{C}(P) \geqq n - \beta(P)$.

Proof. Suppose the lemma is false. Then there exists $T \in \mathscr{A}_P$ and an input $\tilde{x} = (x_1, x_2, \cdots, x_n)$, such that $\text{cost}(T, \tilde{x}) < n - \beta(P)$. Let σ be the output permutation for $\tilde{x}$. We will derive a contradiction.

Let W be the sequence of inequalities $x_i < x_j$ generated along the path $\xi(T, \tilde{x})$; then $|W| < n - \beta(P)$. Denote by Q the partial order on X imposed by W. Then Q has more than $n-(n-\beta(P)) = \beta(P)$ connected components. Thus, there are integers r, s such that y_r, y_s are in the same component in P, while $x_{\sigma(r)}, x_{\sigma(s)}$ are in different components in Q.

Let $y_r = y_{i_1}, y_{i_2}, \cdots, y_{i_m} = y_s$ be such that, for each j, either $y_{i_j} <_P y_{i_{j+1}}$ or $y_{i_j} >_P y_{i_{j+1}}$. By the definition of $T \in \mathscr{A}_P$, every adjacent pairs in the sequence $x_{\sigma(r)} = x_{\sigma(i_1)}, x_{\sigma(i_2)}, \cdots, x_{\sigma(i_m)} = x_{\sigma(s)}$ must also be related in Q. This contradicts the assumption that $x_{\sigma(r)}$ and $x_{\sigma(s)}$ are in different components in Q. □

Theorem 1 follows immediately from Lemmas 1 and 2.

3. Reduction. In this section, we show that to prove Theorem 2, it suffices to prove the following result.

THEOREM 4. *There exists a constant $\lambda > 0$ such that*

$$C(P) \leqq \lambda \left(n - 1 + \log_2 \left(\frac{n!}{\mu(P)} \right) \right).$$

Assume that Theorem 4 is true. We will prove Theorem 2. Let $c = \beta(P)$. By definition, P can be written as the disjoint union of partial orders $P_i = (<_{P_i}, Y_i)$, $1 \leqq i \leqq c$, where the Y_i's form a partition of Y. Let $|Y_i| = n_i$. Then $n_i > 0$ for all i, and

$$\sum_i n_i = n, \tag{5}$$

$$\mu(P) = \left(\frac{n!}{n_1! n_2! \cdots n_c!} \right) \mu(P_1)\mu(P_2) \cdots \mu(P_c). \tag{6}$$

We now describe an algorithm T for P-production. Let $N_0 = 0$. Define $N_i = \sum_{1 \leqq k \leqq i} n_k$, and $I_i = \{j | N_{i-1} < j \leqq N_i\}$ for $1 \leqq i \leqq c$. Given any input set $X = \{x_i | 1 \leqq i \leqq n\}$, consider for each $1 \leqq i \leqq c$, the set X_i as input to P_i-production, where $X_i = \{x_j | j \in I_i\}$. Apply Theorem 4 to each P_i, and let $\sigma_i : I_i \to I_i$ be the permutation found for P_i-production. Then define the output permutation $\sigma \in \Gamma_n$ by $\sigma(j) = \sigma_i(j)$ if $j \in I_i$. It is clear that $T \in \mathscr{A}_P$, since the output permutation σ satisfies the constraint that $y_j <_P y_k$ implies $x_{\sigma(j)} < x_{\sigma(k)}$.

Using (5) and (6), we obtain

$$\begin{aligned} \text{cost}(T) &\leqq \lambda \sum_{1 \leqq i \leqq c} \left(n_i - 1 + \log_2 \left(\frac{n_i!}{\mu(P_i)} \right) \right) \\ &= \lambda \left(n - c + \log_2 \left(\frac{n_1! n_2! \cdots n_c!}{\mu(P_1)\mu(P_2) \cdots \mu(P_c)} \right) \right) \\ &= \lambda \left(n - c + \log_2 \left(\frac{n!}{\mu(P)} \right) \right). \end{aligned}$$

We have proved Theorem 2, assuming that Theorem 4 is true. In the next two sections we will prove Theorem 4.

4. An algorithm.

4.1. Preliminaries. Let $P = (<_P, Y)$ be a partial order. A subset $A \subseteq Y$ is an *independent set* of P, if $y \not<_P y'$ is true for all distinct $y, y' \in A$. The *width* of P, denoted by width (P), is the maximum size of any independent set of P. We associate with each P an independent set Y_P of maximum size. (Pick any one if there are several choices.)

Let $k \geqq 2$ be any integer. A *k-partition* of P is a k-tuple $(A_1, A_2, \cdots, A_k)$, where the A_i's are disjoint subsets of Y whose union equals Y, such that $y <_P y'$, $y \in A_i$ and $y' \in A_j$ imply $i \leqq j$. We are interested in two special k-partitions. Let $\mathscr{B}_P = (B_{P,1}, B_{P,2}, B_{P,3})$, where $B_{P,1} = \{y | y <_P y' \text{ for some } y' \in Y_P\}$, $B_{P,2} = Y_P$, and $B_{P,3} = Y - Y_P - B_{P,1}$. Clearly, $\mathscr{B}_P$ is a 3-partition. Note that some $B_{P,i}$ may be empty. To describe the second partition, let $\mathscr{M}_P$ be the set of all 2-partitions (A_1, A_2) of P such that $|A_1| = \lceil n/2 \rceil$ and $|A_2| = \lfloor n/2 \rfloor$. Let $\mathscr{D}_P = (D_{P,1}, D_{P,2})$ be a member of $\mathscr{M}_P$ such that $\mu(P_1)\mu(P_2)$ is maximum over all possible $(A_1, A_2) \in \mathscr{M}_P$, where P_i is the partial order P restricted to A_i.

Notation. For the rest of the paper, $P = (<_P, Y)$ will denote a partial order on $Y = \{y_1, y_2, \cdots, y_n\}$, and X will denote the input set $\{x_1, x_2, \cdots, x_n\}$. For any $J \subseteq \{1, 2, \cdots, n\}$, we will use Y_J to denote the set $\{y_j | j \in J\}$, and P_J to denote the partial order induced by P on Y_J; we agree that $\mu(P_J) = 1$ when $J = \varnothing$. Similarly, for any $I \subseteq \{1, 2, \cdots, n\}$, we use X_I to denote the set $\{x_i | i \in I\}$. For any two sets of numbers A, B, we write $A < B$ if $y < z$ for all $y \in A$ and $z \in B$. We adopt the convention that $0! = 1$, and we will employ two constants $c_2 = 40$ and $c_3 = 80$.

LEMMA 3. *Let $k \in \{2, 3\}$. Let I be a nonempty subset of $\{1, 2, \cdots, n\}$, and $n_1, n_2, \cdots, n_k$ be nonnegative integers satisfying $\sum_{1 \leqq i \leqq k} n_i = |I|$. Then there is a decision tree T of height $c_k|I|$ such that, given any input of n distinct numbers $X = \{x_1, x_2, \cdots, x_n\}$, T determines disjoint $I_1, I_2, \cdots, I_k$ satisfying* (a) $\cup_i I_i = I$, (b) $|I_i| = n_i$ *for all i, and* (c) $X_{I_1} < X_{I_2} < \cdots < X_{I_k}$.

Proof. If $k = 2$, the decision tree first finds the $(n_1 + 1)$st smallest element x_j in X_I, and then determines $I_1 = \{i | x_i < x_j\}$ and $I_2 = \{i | x_i \geqq x_j\}$. This can be done in less than $40n$ comparisons using the selection algorithm in [BFPRT]. Similarly, if $k = 3$, we can find the $(n_1 + 1)$st smallest and the $(n_1 + n_2 + 1)$st smallest elements in X_I, by applying the selection algorithm in [BFPRT] twice, and then find I_1, I_2, I_3. □

4.2. Procedure POPROD. The algorithm can be described as a recursive procedure. Depending on the width of P, we will either use comparisons to divide X into three parts X_I, satisfying $X_{I_1} < X_{I_2} < X_{I_3}$, or use comparisons to divide X into two parts X_{I_i} satisfying $X_{I_1} < X_{I_2}$. In the first case, we can match the elements $y_j \in B_{P,2}$ with elements in X_{I_2} in any fashion, and then recursively solve two subproblems: X_{I_1} as input to the production problem of P restricted to $B_{P,1}$, and X_{I_3} as input to the production problem of P restricted to $B_{P,3}$. This gives a valid final output, because $B_{P,2}$ is an independent set in P and $(B_{P,1}, B_{P,2}, B_{P,3})$ is a 3-partition. In the other case, we will simply solve recursively two subproblems: X_{I_1} as input to the production problem of P restricted to $D_{P,1}$, and X_{I_2} as input to the production problem of P restricted to $D_{P,2}$. Of course, the cardinality of the sets I_i needs to be chosen to match those of the 3-partitions and 2-partitions.

The criterion for deciding which case to use is whether the width of P is greater than a fraction of n. Intuitively, the first case is more like the median-finding problem and the second case is more like the sorting problem. In the first case, we would like to get immediately a large independent subset of the elements y_j in Y assigned, while in the second case, we rely on the technique of divide-and-conquer, and try to divide the problems into two subproblems of nearly equal size.

As an example, consider the partial order P shown in Fig. 1. The width of P is relatively large, and we have the first case. For this partial order, $B_{P,1} = \{y_1, y_2, y_3, y_4\}$, $B_{P,2} = \{y_5, y_8, y_{10}, y_{11}\}$, and $B_{P,3} = \{y_6, y_7, y_9, y_{12}, y_{13}\}$. We thus use comparisons to divide X into three parts X_{I_i} satisfying $X_{I_1} < X_{I_2} < X_{I_3}$, where $|I_1| = 4$, $|I_2| = 4$, and $|I_3| = 5$. The elements in $B_{P,2}$ can be assigned in a $1-1$ way to the x's in X_{I_2} without

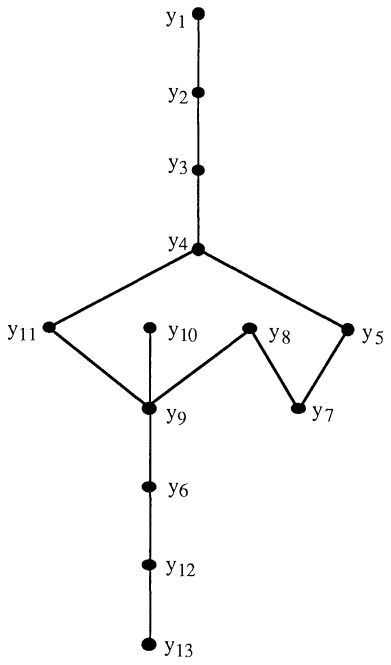


FIG. 1. *A partial order P; smaller elements on top, e.g.,* $y_2 <_P y_4$.

further comparisons. Of the two subproblems to be solved recursively, we will examine just the first one. To match the elements in $B_{P,1}$ with X_{I_1}, we observe that Q, the partial order P restricted to $B_{P,1}$, is $y_1 <_Q y_2 <_Q y_3 <_Q y_4$, which has width equal to 1. This means we have the second case for this subproblem. Clearly, $D_{Q,1} = \{y_1, y_2\}$ and $D_{Q,2} = \{y_3, y_4\}$. Therefore, we use comparisons to divide X_{I_1} into two parts A and A' with $A < A'$. Now we need to solve two subproblems: matching $D_{Q,1}$ to A, and $D_{Q,2}$ to A'.

We now specify the algorithm formally. Given an input set $X = \{x_1, x_2, \cdots, x_n\}$, the algorithm will output a permutation σ in the form of a set $\{(i, \sigma(i)) | 1 \leqq i \leqq n\}$; the correctness requirement is that $y_i <_P y_j$ implies $x_{\sigma(i)} < x_{\sigma(j)}$. We will give a recursive algorithm that takes as additional input arguments two sets $J \subseteq \{1, 2, \cdots, n\}$ and $I \subseteq \{1, 2, \cdots, n\}$ of equal size, and returns a *matching* between J and I, i.e., a set $V \subseteq J \times I$ such that each $j \in J$ appears exactly in one element $(j, k) \in V$, and each $i \in I$ appears exactly in one element $(m, i) \in V$. We will later prove that the matching produced satisfies the condition that, for $(j, m) \in V$ and $(j', m') \in V$, $y_j <_P y_{j'}$ implies $x_m < x_{m'}$. Thus, if we let $J = I = \{1, 2, \cdots, n\}$, we obtain the required permutation σ in the output.

PROCEDURE POPROD (X, J, I).

Case 1. $|J| \leqq 1$:

if $J = I = \varnothing$, then return $\varnothing$;
if $J = \{j\}$, $I = \{i\}$ then return $\{(j, i)\}$;

Case 2. $(|J| > 1) \wedge (\text{width}(P_J) > \lceil |J|/100 \rceil)$:

(a) Use $c_3|I|$ or less comparisons to divide I into disjoint I_1, I_2, I_3 such that $X_{I_1} < X_{I_2} < X_{I_3}$ and $|I_i| = |B_{P_{J,i}}|$ for $1 \leqq i \leqq 3$;
[Comments: This can be done by Lemma 3.]

(b) Suppose $B_{P_{J,i}} = \{y_j | j \in J_i\}$, for $1 \leqq i \leqq 3$;
let $V_2 \leftarrow \{(k_s, i_s) | 1 \leqq s \leqq |J_2|\}$, where k_s and i_s are the sth smallest elements in J_2 and I_2.
[Comments: No comparisons are used here.]
(c) $V_1 \leftarrow$ POPROD (X, J_1, I_1);
$V_3 \leftarrow$ POPROD (X, J_3, I_3);
(d) return $V \leftarrow V_1 \cup V_2 \cup V_3$;

Case 3. $(|J| > 1) \wedge (\text{width}(P_J) \leqq \lceil |J|/100 \rceil)$:
(a) Use $c_2|I|$ or less comparisons to divide I into disjoint I_1, I_2 such that $X_{I_1} < X_{I_2}$, $|I_1| = \lceil n/2 \rceil$, and $|I_2| = \lfloor n/2 \rfloor$;
[Comments: This can be done by Lemma 3.]
(b) Suppose $D_{P_{J,i}} = \{y_j | j \in J_i\}$, for $1 \leqq i \leqq 2$;
(c) $V_1 \leftarrow$ POPROD (X, J_1, I_1);
$V_2 \leftarrow$ POPROD (X, J_2, I_2);
(d) return $V \leftarrow V_1 \cup V_2$;

4.3. Correctness.

LEMMA 4. *In Procedure* POPROD, *the returned value V is a matching between J and I.*

Proof. We prove the lemma by induction on the size of J. The base case $|J| \leqq 1$ is obvious. Inductively, suppose $|J| > 1$, and that the first recursive call results in Case 2. Then V_2 constructed in step (b) is clearly a matching between J_2 and I_2. In step (c), by induction hypothesis, V_i is a matching between J_i and I_i for $i \in \{1, 3\}$. Thus, V is a matching between J and I. A similar argument can be given when the first recursive call results in Case 3. □

LEMMA 5. *Let V be the returned value in Procedure* POPROD, *when X, J, I are the input arguments. If* $(k, k') \in V$, $(m, m') \in V$ *and* $y_k <_P y_m$, *then* $x_{k'} < x_{m'}$.

Proof. We prove the lemma inductively on the size of the set J. The base case $|J| \leqq 1$ is obvious. Inductively, suppose $|J| > 1$, and that the first recursive call results in Case 2. Suppose that $(k, k') \in V_i$ and $(m, m') \in V_j$. As $y_k <_P y_m$, we have $i \leqq j$ (since $(B_{P_J,1}, B_{P_J,2}, B_{P_J,3})$ is a 3-partition of P_J by definition). If $i < j$, then $x_{k'} < x_{m'}$, as $x_{k'} \in X_{I_i}$, $x_{m'} \in X_{I_j}$, and $X_{I_i} < X_{I_j}$. If $i = j \in \{1, 3\}$, then the lemma holds by the induction hypothesis. The case $i = j = 2$ does not arise, since no two distinct y_k and y_m in $B_{P_J,2}$ are comparable in P_J. A similar argument can be given when the first recursive call results in Case 3. □

This proves that Procedure POPROD defines a decision tree algorithm for P-production, when we set $J = I = \{1, 2, \cdots, n\}$ in the input arguments. To complete the proof of Theorem 4, it remains to analyze the number of comparisons used in this procedure. This will be done in the next section.

5. Analysis of POPROD. Let $f_P(J)$ be the maximum number of comparisons used in POPROD (X, J, I) for any I and any relative ordering of the elements in X. Let $\lambda_2 = 5000c_2$, and $\lambda_3 = 100c_3$. In this section, we will prove

$$f_P(J) \leqq \lambda_2 n + \lambda_3 \log_2 \left(\frac{|J|!}{\mu(P_J)} \right). \tag{7}$$

This will complete the proof of Theorem 4.

5.1. Two lemmas. We digress to prove two auxiliary lemmas before proving (7). We need a classical theorem due to Dilworth [D].

DILWORTH'S THEOREM [D]. *Let* $P=(<_P, W)$ *be any partial order, and* width $(P)=m>0$. *Then* W *can be written as the disjoint union of* m *nonempty sets* $W_i=\{w_{i,1}, w_{i,2}, \cdots, w_{i,t_i}\}$, $1 \leqq i \leqq m$, *such that* $w_{i,1}<_P w_{i,2}<_P \cdots <_P w_{i,t_i}$ *for all* i.

Proof. See [D] for the proof. □

Let $P=(<_P, Y)$ be any partial order on a nonempty set $Y=\{y_1, y_2, \cdots, y_n\}$. Let $\mathscr{B}_P=(B_{P,1}, B_{P,2}, B_{P,3})$ and $\mathscr{D}_P=(D_{P,1}, D_{P,2})$ be the two special k-partitions defined in § 4.1.

LEMMA 6. *Let* J_1, J_2, J_3 *be such that* $B_{P,i}=\{y_j | j \in J_i\}$ *for* $i \in \{1,2,3\}$. *Then*

$$\frac{\mu(P_{J_1})}{|J_1|!}\frac{\mu(P_{J_3})}{|J_3|!} \geqq \frac{\mu(P)}{|Y|!}.$$

Proof. Take a random $\sigma \in \Gamma_n$. Let E be the event that $\sigma(j)<\sigma(k)$ for all $y_j <_P y_k$, and for $i \in \{1,3\}$, let E_i be the event that $\sigma(j)<\sigma(k)$ for all $y_j <_P y_k$ with $j, k \in J_i$. Thus, E is the event that $\sigma \in \Delta(P)$, and E_i is the event that σ is consistent with P_{J_i}.

Clearly, E implies $E_1 \wedge E_3$. Furthermore, E_1 and E_3 are independent events. It follows that

$$\begin{aligned}\frac{\mu(P)}{|Y|!} &= \Pr\{E\} \\ &\leqq \Pr\{E_1 \wedge E_3\} \\ &= \Pr\{E_1\} \cdot \Pr\{E_3\} \\ &= \frac{\mu(P_{J_1})}{|J_1|!}\frac{\mu(P_{J_3})}{|J_3|!}.\end{aligned}$$

This proves Lemma 6. □

LEMMA 7. *Suppose* $n>1$. *Let* J_1, J_2 *be such that* $D_{P,i}=\{y_j | j \in J_i\}$ *for* $i \in \{1,2\}$. *If* width $(P) \leqq \lceil n/100 \rceil$, *then*

$$\frac{\mu(P_{J_1})}{|J_1|!}\frac{\mu(P_{J_2})}{|J_2|!} \geqq (1.01)^{|Y|}\frac{\mu(P)}{|Y|!}.$$

Proof. Let $m=$ width (P). If $m=1$, then P is a linear order, and it is easy to see that the lemma is true in this case. We can thus assume that $m>1$. Since $\lceil n/100 \rceil \geqq m$, we have $n>100$. Clearly, both J_i are nonempty. By Dilworth's Theorem, P can be covered by m chains of lengths, say, $l_1, l_2, \cdots, l_m>0$. Thus, each 2-partition $(A_1, A_2) \in \mathscr{M}_P$ can be specified by integers $k_1, k_2, \cdots, k_m$, where k_i is the number of elements of A_1 on the ith chain. After standard manipulations for optimizing expressions, we obtain

$$\begin{aligned}|\mathscr{M}_P| &\leqq \prod_{1 \leqq i \leqq m}(1+l_i) \\ &\leqq \left(\frac{2n}{m}\right)^m \\ &\leqq (200)^{\lceil n/100 \rceil}.\end{aligned} \tag{8}$$

For each $\sigma \in \Delta(P)$, let $K_\sigma=\{y_j | 1 \leqq \sigma(j) \leqq \lceil n/2 \rceil\}$. Then $(K_\sigma, Y-K_\sigma) \in \mathscr{M}_P$. If we partition $\Delta(P)$ according to the value of K_σ, then there are $|\mathscr{M}_P|$ classes, each containing at most $\mu(P_{J_1})\mu(P_{J_2})\sigma$'s. Thus,

$$\mu(P) \leqq |\mathscr{M}_P|\mu(P_{J_1})\mu(P_{J_2}). \tag{9}$$

From (8), (9), and the fact $n > 100$, we obtain

$$\mu(P) \leqq (200)^{\lceil n/100 \rceil} \mu(P_{J_1}) \mu(P_{J_2})$$

$$\leqq \frac{1}{(1.01)^n} \binom{n}{\lceil n/2 \rceil} \mu(P_{J_1}) \mu(P_{J_2}).$$

Rearranging terms in the above expression gives the inequality to be proved in the lemma. □

5.2. The analysis. If $n = 1$, then $f_P(J) = 0$ and (7) is clearly true. We can thus assume $n \geqq 2$. In the algorithm POPROD, we can group the comparisons used into *steps*, each step performing either a 2-partition or a 3-partition of a subset of the input X. In our analysis, we will estimate separately the number of comparisons used for 2-partitions and for 3-partitions. To facilitate the analysis, we will utilize an auxiliary tree, with each internal node representing a step; the structure of the tree, together with information associated with the nodes, will contain enough details about the execution of the algorithm to permit an upper-bound estimate of the cost.

Given P, J, where $J \neq \emptyset$, we construct a *cost tree* $V_{P,J}$. Each node v will be associated with a triplet $\eta(v) = (\delta(v), \alpha(v), S(v))$, where $\delta(v) \in \{0, 2, 3\}$, $\alpha(v)$ is a nonnegative integer, and $S(v) \subseteq Y$. We will say that v is of *type* $\delta(v)$ and *weight* $\alpha(v)$.

The cost trees are recursively constructed. If $|J| = 1$ with, say, $J = \{j\}$, then $V_{P,J}$ consists of a single node v with $\eta(v) = (0, 0, \{y_j\})$.

If $(|J| > 1) \wedge (\text{width}(P_J) > \lceil |J|/100 \rceil)$, then the root v of $V_{P,J}$ has $\eta(v) = (3, c_3|J|, B_{P_{J,2}})$, and for each nonempty $B_{P_{J,i}}$, $i \in \{1, 3\}$, there is a son v_i of the root such that the subtree rooted at v_i is V_{P,J_i}, where J_i is the set of j with $y_j \in B_{P_{J,i}}$.

If $(|J| > 1) \wedge (\text{width}(P_J) \leqq \lceil |J|/100 \rceil)$, then the root v of $V_{P,J}$ has $\eta(v) = (2, c_2|J|, \emptyset)$, and for each $i \in \{1, 2\}$, there is a son v_i of the root such that the subtree rooted at v_i is V_{P,J_i}, where J_i is the set of j with $y_j \in D_{P_{J,i}}$. (Note that in this case both J_i are nonempty.)

We have defined the cost tree $V_{P,J}$. An example of $V_{P,J}$ is shown in Fig. 2, where P is the partial order in Fig. 1 and $J = \{1, 2, \cdots, n\}$; square nodes are of type 0, oval nodes with c_2 beside them are of type 2, and those with c_3 beside them are of type 3.

We now relate $f_P(J)$ to the cost tree. Let $a_i(P, J)$ be the total weight of type-i nodes in $V_{P,J}$. That is, let $a_i(P, J) = \sum_{v, \delta(v) = i} \alpha(v)$ for $i \in \{2, 3\}$.

LEMMA 8. $f_P(J) \leqq a_2(P, J) + a_3(P, J)$.

Proof. We prove the lemma by induction on the size of $|J|$. If $|J| = 1$, then $f_P(J) = a_2(P, J) = a_3(P, J) = 0$, and the lemma holds. Inductively, assume that $|J| = m > 1$, and that we have proved the lemma for all J with size less than m. If $\text{width}(P_J) > \lceil |J|/100 \rceil$, then in the execution of Procedure POPROD, for any I and X, Case 2 occurs at the top level, and thus

$$f_P(J) \leqq c_3|J| + \sum_{i \in \{1,3\}, J_i \neq \emptyset} f_P(J_i),$$

where J_i are the sets of j such that $y_j \in B_{P_{J,i}}$. Applying the induction hypothesis, we have

$$f_P(J) \leqq c_3|J| + \sum_{i \in \{1,3\}, J_i \neq \emptyset} (a_2(P, J_i) + a_3(P, J_i))$$

$$= a_2(P, J) + a_3(P, J).$$

A similar argument works when $\text{width}(P_J) \leqq \lceil |J|/100 \rceil$. This completes the inductive step of the proof. □

We now analyze $a_i(P, J)$.

LEMMA 9. $a_3(P, J) \leqq 100 c_3 |J|$.

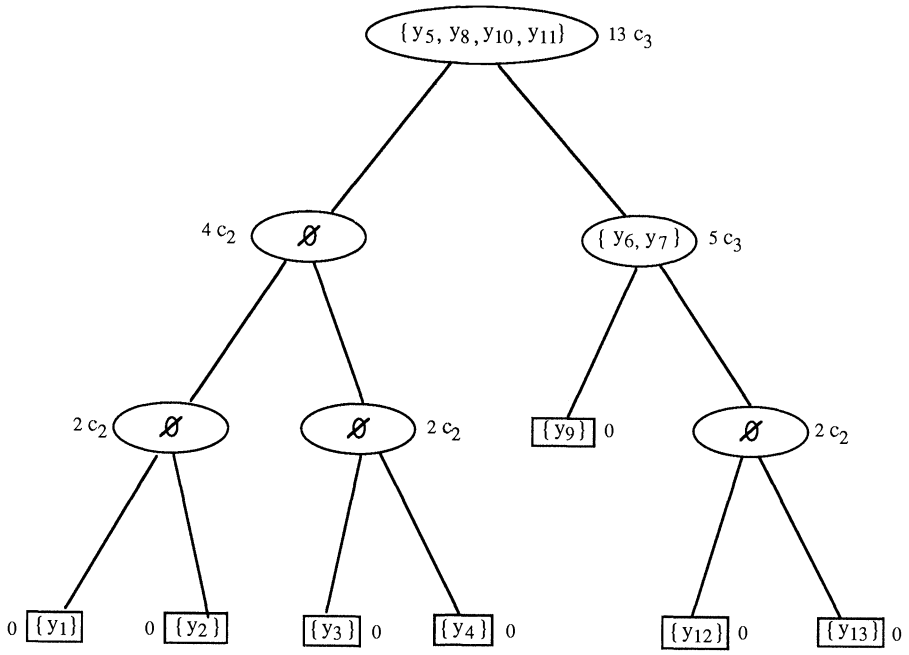


FIG. 2. *$V_{P,J}$ with the P in Fig. 1 and $J=\{1,2,\cdots,13\}$; $S(v)$ is shown inside the nodes v, and $\alpha(v)$ is shown just outside v.*

Proof. We first state two facts that can be easily verified. For all type-3 internal nodes v,

$$|S(v)|>\lceil\alpha(v)/(100c_3)\rceil. \tag{10}$$

For any two distinct internal nodes v and v',

$$S(v)\cap S(v')=\varnothing. \tag{11}$$

It follows from (10) and (11) that

$$\begin{aligned} a_3(P,J) &= \sum_{v,\delta(v)=3}\alpha(v) \\ &\leqq 100c_3\sum_{v,\delta(v)=3}|S(v)| \\ &\leqq 100c_3|Y_J| \\ &= 100c_3|J|, \end{aligned}$$

where the summation of v is over nodes in $V_{P,J}$. This proves the lemma. □

LEMMA 10. $a_2(P,J)\leqq 5000c_2\log_2(|J|!/\mu(P_J))$.

Proof. We prove the lemma inductively on the size of J. If $|J|=1$, then $a_2(P,J)=0$, $|J|=\mu(P_J)=1$, and the lemma is valid. Inductively, suppose that $|J|=m>1$ and that we have proved the lemma for all J, P with $|J|<m$.

If width $(P_J)>\lceil|J|/100\rceil$, then in the execution of Procedure POPROD, for any I and X, Case 2 occurs at the top level. Let J_i be the set of j with $y_j\in B_{P_{J,i}}$. Applying

the induction hypothesis to each son v_i, and keeping in mind that we employ the convention that $0! = \mu(P_\emptyset) = 1$, we obtain

$$a_2(P, J) = \sum_{i \in \{1,3\}, J_i \neq \emptyset} a_2(P, J_i)$$

$$\leqq 5000c_2 \sum_{i \in \{1,3\}} \log_2 \left(\frac{|J_i|!}{\mu(P_{J_i})} \right)$$

$$= 5000c_2 \log_2 \left(\frac{|J_1|!|J_3|!}{\mu(P_{J_1})\mu(P_{J_3})} \right).$$

Applying Lemma 6 to the partial order P_J, we then have

$$a_2(P, J) \leqq 5000c_2 \log_2 \left(\frac{|J|!}{\mu(P_J)} \right).$$

If width $(P_J) \leqq \lceil |J|/100 \rceil$, Case 3 occurs in the execution of POPROD. Let J_i be the set of j with $y_j \in D_{P_{J,i}}$. Both J_i are nonempty in this case. We have

$$a_2(P, J) \leqq c_2|J| + 5000c_2 \log_2 \left(\frac{|J_1|!}{\mu(P_{J_1})} \right) + 5000c_2 \log_2 \left(\frac{|J_2|!}{\mu(P_{J_2})} \right)$$

$$= c_2 m + 5000c_2 \log_2 \left(\frac{|J_1|!|J_2|!}{\mu(P_{J_1})\mu(P_{J_2})} \right).$$

Applying Lemma 7 to P_J, we obtain

$$a_2(P, J) \leqq c_2 m + 5000c_2 \log_2 \left(\frac{m!}{\mu(P_J)} \frac{1}{(1.01)^m} \right)$$

$$\leqq 5000c_2 \log_2 \left(\frac{m!}{\mu(P_J)} \right).$$

This completes the inductive step of the proof. □

Inequality (7), and hence Theorem 4, follows immediately from the preceding three lemmas. This completes the proof of Theorem 2.

6. Remarks. In this paper we have determined up to a constant factor the complexity of a class of problems involving partial orders, in terms of a familiar combinatorial quantity $\mu(P)$. It is of interest to explore the complexity of other classes of problems involving partial orders. An excellent survey of this topic can be found in Saks [Sa]. We list below some open problems directly related to our present discussion.

(a) Can we characterize the complexity of producing σ that satisfies more general constraints than a single partial order P? For example, let $P_1, P_2, \cdots, P_m$ be partial orders on $Y = \{y_1, y_2, \cdots, y_n\}$. What is the complexity of producing, for any input $X = \{x_1, x_2, \cdots, x_n\}$, a σ such that for some i, $y_j <_{P_i} y_k$ implies $x_{\sigma(j)} < x_{\sigma(k)}$ for all j, k?

(b) The results in this paper imply that the randomized decision tree complexity for P-production is asymptotically of the same order of magnitude as the worst-case complexity. Is this true for more general class of production problems, such as the one mentioned in (a)? It is also of interest to study the question of whether randomization helps for other types of partial order problems.

(c) The present paper gives an existence proof of a near-optimal height decision tree for P-production. If the partial order itself is also given as an input, is there a

polynomial time algorithm (counting all the bookkeeping steps) that uses a near-optimal number of comparisons for producing a partial order?

Acknowledgments. The author would like to thank the referees for many helpful comments and suggestions.

REFERENCES

[Ab] N. ABRAMSON, *Information Theory and Coding*, McGraw-Hill, New York, 1963.

[Ai] M. AIGNER, *Producing posets*, Discrete Math., 35 (1981), pp. 1–15.

[BFPRT] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST, AND R. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448–461.

[D] R. P. DILWORTH, *A decomposition theorem for partially ordered sets*, Ann. of Math., 2 (1950), pp. 161–166.

[FR] R. W. FLOYD AND R. L. RIVEST, *Expected time bounds for selection*, Commun. ACM, 18 (1975), pp. 165–172.

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

[Sa] M. E. SAKS, *The information theoretic bound for problems on ordered sets and graphs*, in Graphs and Order, I. Rival, ed., D. Reidel, Boston, MA, 1985, pp. 137–168.

[Sch] A. SCHÖNHAGE, *The production of partial orders*, Astérisque, 38–39 (1976), pp. 229–246.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 690-710, August 1989

005

A FAST ALGORITHM FOR MULTIPROCESSOR SCHEDULING OF UNIT-LENGTH JOBS*

BARBARA B. SIMONS† AND MANFRED K. WARMUTH‡

Abstract. An efficient polynomial time algorithm for the problem of scheduling n unit length jobs with rational release times and deadlines on m identical parallel machines is presented. By using preprocessing, a running time of $O(mn^2)$ is obtained that is an improvement over the previous best running time of $O(n^3 \log\log n)$. The authors also present new NP-completeness results for two closely related problems.

Key words. scheduling, multiprocessor, release time, deadline, parallel computing, computational complexity, NP-complete

AMS(MOS) subject classifications. 05, 68

1. Introduction. We present an efficient polynomial time algorithm for the problem of scheduling n unit length jobs with rational release times and deadlines on m identical parallel machines. The question of how the requirement that the jobs all have the same length affects the problem was first answered in [10], where a polynomial time algorithm with time complexity $O(n^2 \log n)$ is presented for the single machine case. An alternative algorithm with the same time complexity was subsequently obtained by [1]. Finally, an algorithm with time complexity $O(n \log n)$ for the single machine case was presented in [5].

In the multimachine case, the only previously known polynomial time algorithm has a worst-case running time of $O(n^3 \log\log n)$ [11]. We improve this running time to $O(mn^2)$ by doing some preprocessing before the jobs are actually scheduled. This speedup is obtained by generalizing to an arbitrary number of machines the notion of "forbidden regions," which was developed in [5].

If different integer job lengths are allowed, then by a simple reduction from 3 PARTITION [4], determining whether or not a schedule exists is strongly NP-complete, even if $m = 1$ and all release times and deadlines are integers. If m is arbitrary, then there is a similar reduction in which all jobs are released at time 0 and have the same integer deadline.

We strengthen the above NP-completeness result by allowing only a small number of integer job lengths:

(A) Three integer job lengths (1, 3, and q for some integer q), m machines, integer deadlines, but only one overall release time;

(B) Two integer job lengths (1 and q), m machines, integer release times and deadlines.

2. An overview. The algorithm, called BOUNDED_REGION, has two major sections. The first, called the BACKSEQUENCE Algorithm, is the preprocessing section. It determines a set BR of regions in which only a bounded number of jobs can be started in any feasible schedule. The actual scheduling of the jobs is done by SCHEDULE (BR). This procedure schedules jobs as early as possible subject to the restriction that the regions of BR are not violated. Below is a top level description of the algorithm.

* Received by the editors June 16, 1986; accepted for publication (in revised form) June 15, 1988.

† IBM Almaden Research Center, K53-802, 650 Harry Road, San Jose, California 95120-6099.

‡ Computer and Information Sciences, University of California, Santa Cruz, California 95064. The work of this author was supported by Office of Naval Research grant N00014-86-K-0454.

```
ALGORITHM BOUNDED_REGION.
Begin
  Read input;
  BR := BACKSEQUENCE;
  If BACKSEQUENCE returns failure condition then HALT in failure;
  Call SCHEDULE (BR);
end BOUNDED_REGION.
```

3. Definitions. We shall assume that there is a set J of *n jobs*, $J(1), J(2), \cdots, J(n)$, and a set M of *m machines*. Each job $J(i)$ has a nonnegative integer *processing requirement*, $p(i)$, a nonnegative rational *release time*, $r(i)$, and a nonnegative rational *deadline*, $d(i)$, with $d(i) \geqq r(i)+p(i)$. When we speak of a job $J(i)$ being *released* by time t, we mean that $r(i) \leqq t$. If job $J(i)$ is *started* at time t, then it is *finished* at time $t+p(i)$ and occupies the interval $[t, t+p(i))$. A *schedule* SCH for a problem instance J and M is an assignment of a nonnegative start time $s(i)$ and a machine $m(i)$, $0 \leqq m(i) \leqq m-1$, for each $J(i) \in J$ such that the following conditions hold.

(1) No job is started before its release time or finished later than its deadline, i.e., $s(i) \geqq r(i)$ and $s(i)+p(i) \leqq d(i)$.

(2) No two jobs overlap, i.e., if job $J(i)$ is started at time t on machine $m(k)$, then no other job assigned to $m(k)$ is started in the interval $[t, t+p(i))$.

Note that in our definition of schedule *preemption* is not allowed, that is, once a job has begun execution it cannot be interrupted and consequently must run until it is completed.

Except for the section on NP-completeness, in which $p(i)$ is allowed to assume one of several integer values, we shall assume that $p(i)=1$. For this case a schedule consists of a *sequence* of start times, where a sequence S for n jobs and m machines is a nondecreasing list of n nonnegative start times with the additional constraint that for $i > m$ the ith start time of the sequence is at least one unit greater than the $(i-m)$th start time. This constraint guarantees that no more than m jobs are being processed simultaneously. We say that such a sequence S is of *length* n, i.e., $|S|=n$. If S is a sequence with $|S|=n$, then S_i, $1 \leqq i \leqq n$, denotes the ith start time counting forward from the beginning of the sequence. Likewise, S^i denotes the ith start time counting backward from the end of the sequence. Note that $S_i = S^{n+1-i}$, for $1 \leqq i \leqq n$, and that S_i is the ith smallest and S^i is the ith largest start time of S. Given a problem instance J and M, a sequence S of length n is an *r-sequence* if there exists a 1-1 mapping s (written $s(i)$ instead of $s(J(i))$) from jobs to elements of S such that $r(i) \leqq s(i)$. Similarly, a sequence S of length n is a *d-sequence* if there exists a 1-1 mapping s from jobs to elements of S such that $s(i) \leqq d(i)-1$. The aim is to produce an *rd-sequence* of length n for the set of n jobs, namely, a sequence for which there exists a 1-1 mapping s such that $r(i) \leqq s(i) \leqq d(i)-1$.

Given a schedule SCH for a problem instance J and M, then the sorted list of start times is clearly an *rd*-sequence. For the opposite direction, assume the m machines are numbered $0, 1, \cdots, m-1$. Given an *rd*-sequence of length n for J, the jobs in J can be assigned start times in SCH by applying the following Earliest Deadline Rule first to S_1, then S_2, then S_3, and so on: the unscheduled job with the smallest deadline from among all jobs released by time S_i is started at time S_i on machine i mod m.

4. Bounded and forced regions. If the release times are integers, the deadlines rationals, and the processing requirement one unit, then it is possible to construct a schedule in linear time if one exists [3], [7]. (Recall that our definition of schedule is an assignment of start times and machines to jobs.) A simple reduction from sorting

shows, however, that the construction of an *rd*-sequence requires $\Omega(n \log n)$ time. This time bound is attained by the Earliest Deadline Algorithm [8], which constructs an *rd*-sequence as well as a schedule for that sequence directly from the problem instance in $O(n \log n)$ time. The Earliest Deadline Algorithm computes start time S_i by setting S_i to be the minimum of the release times of the unscheduled jobs and $S_{i-m}+1$, for $i > m$. It then uses the Earliest Deadline Rule to select the job that is assigned start time S_i and machine $i \bmod m$.

This approach fails if arbitrary rational release times and deadlines are allowed. Intuitively, this happens for two reasons. First, jobs may be released during the time that other jobs are already running. (Note that this can be avoided if the release times are integers, since this implies that the start times can be constrained to be integers.) Second, there may be a set of jobs all of which are released on or after some release time, say $r(j)$, and have deadlines less than or equal to some deadline, say $d(i)$, such that the jobs in this set fill up almost the entire interval between $r(j)$ and $d(i)$. If "too many" jobs are started just before $r(j)$, then these jobs will extend into the interval $[r(j), d(i))$, and there will not be enough space in which to schedule all the jobs from the set. Consequently, it is necessary to bound the number of jobs that start less than one unit prior to $r(j)$. In the single machine case, it may be necessary to construct a forbidden region in which no job can start. Such a region has length no greater than one and has a release time as its right endpoint [5]. The m machine case becomes more complex because there can be up to m such intervals, each interval having a different length and a different restriction on the number of jobs that can start in its interior, but having the same release time as its right endpoint.

A *region* is defined to be an interval, either opened at both ends or closed at both ends. The length of a region is always no greater than one and can equal one only if the region is an open interval. The BACKSEQUENCE Algorithm computes regions in which at least k jobs *must* start in any *rd*-sequence. Because each job runs for one unit of time, this implies regions in which at most $m-k$ jobs *are allowed* to start in any *rd*-sequence. We refer to such a region as a *k-forced start region* and an $(m-k)$-*bounded start region*, respectively. We also say that k (respectively, $m-k$) is the *degree* of the k-forced (respectively, $(m-k)$-bounded) start region. If the region $[\alpha, \beta]$ is a k-forced start region, then the region $(\beta-1, \alpha)$ is an $(m-k)$-bounded start region. Note that if more than $m-k$ jobs start in the region $(\beta-1, \alpha)$, then k jobs cannot start in the region $[\alpha, \beta]$. (We require that the parameter k always lies in the range from one to m.) Lemma 0 below follows directly from the definitions of forced and bounded regions.

LEMMA 0. *If $[\alpha, \beta]$ is a k-forced region, then $(\beta-1, \alpha)$ is an $(m-k)$-bounded region.*

We say that an $(m-k)$-bounded start region is *correct* if there is no *rd*-sequence for the problem instance in which more than $(m-k)$ jobs start in the region. If a sequence or schedule has no more than $m-k$ jobs started in an $(m-k)$-bounded region, we say that the $(m-k)$-bounded region has been *avoided*. If more than $m-k$ jobs are scheduled to start in a $(m-k)$-bounded region, then we say that the $(m-k)$-bounded region is *violated*. The terms *correct*, *avoided*, and *violated* are defined similarly for k-forced regions.

Forced and bounded regions always come in pairs. The reader should bear in mind that we are concerned only with the number of jobs that actually start on all m machines in such a region (as opposed to those that have started earlier and are already running in the region). For reasons that will become apparent later, a k-forced region has a release time at its left endpoint and an $(m-k)$-bounded region has a release

time as its right endpoint. Bounded regions are open intervals and forced regions are closed intervals.

4.1. How to avoid bounded regions. We distinguish between two modes of constructing sequences. In the *backward* mode we select the latest possible start time by scanning backward starting from a deadline. In the *forward* mode we select the earliest possible start time for each job by scanning forward starting from a release time. BACKSEQUENCE constructs n sequences using the backward mode starting from each of the deadlines. At the end of each iteration, BACKSEQUENCE uses information from the sequences it has constructed to create a new set of up to m bounded regions. It then calls procedure ADD, which computes additional bounded regions implied by those already constructed. Finally, forward sequencing is used in the procedure SCHEDULE to construct the schedule for the problem instance.

Let S be a sequence, $|S| = v$. The subroutine NEXTFORWARD (S, BR, t), given below, returns the smallest start time S^0 such that $S^0 \geqq t$, $S^0 \geqq S^i$, $1 \leqq i \leqq v$, and all the bounded regions of BR are avoided by the start times of $S|S^0$, where | denotes "appended to." Note that if $S' = S|S^0$, then $S'^i = S^{i-1}$, for $1 \leqq i \leqq v+1$. Similarly, the subroutine NEXTBACKWARD (S, BR, t) returns the largest start time S_0 such that $S_0 \leqq t$, $S_0 \leqq S_i$, for $1 \leqq i \leqq v$, and all the bounded regions of BR are avoided by $S_0|S$. We use both procedures iteratively to construct sequences that avoid all bounded regions of BR. For examples of how bounded regions are avoided, see Figs. 3 and 4 in § 11.

SUBROUTINE NEXTFORWARD (S, BR, t) (* returns minimum start time $S^0 \geqq t$ such that the bounded regions of BR are avoided *)
(0) Let $|S| = v$;
  if $v < m$ then $S^m := t-1$; (* This prevents S^m from being undefined *)
(1) $S^0 := \max(t, S^1, S^m+1)$; (* This guarantees that $S|S^0$ is a sequence *)
(2) process the bounded regions of BR in order of nondecreasing left endpoints:
  let R be the next bounded region of BR, let $m-k$ be the degree of R, and let $\beta-1$ and α be the left and right endpoints, respectively, of R;
    if $m-k \leqq v$ then
    if $S^{m-k} \in (\beta-1, \alpha)$ then $S^0 := \max\{S^0, \alpha\}$; (* S^0 is increased by the minimum amount such that the $(m-k)$-bounded region $(\beta-1, \alpha)$ is avoided *)
(3) return (S^0).

SUBROUTINE NEXTBACKWARD (S, BR, t) (* returns maximum start time $S_0 \leqq t$ such that the bounded regions of BR are avoided *)
(0) Let $|S| = v$;
  if $v < m$ then $S_m := t+1$; (* This prevents S_m from being undefined *)
(1) $S_0 := \min(t, S_1, S_m-1)$; (* This guarantees that $S_0|S$ is a sequence *)
(2) process the bounded regions of BR in order of nonincreasing right endpoints:
  let R be the next bounded region of BR, let $m-k$ be the degree of R, and let $\beta-1$ and α be the left and right endpoints, respectively, of R;
    if $m-k \leqq v$ then
    if $S_{m-k} \in (\beta-1, \alpha)$ then $S_0 := \min(S_0, \beta-1)$; (* S_0 is decreased by the minimum amount such that the $(m-k)$-bounded region $(\beta-1, \alpha)$ is avoided *)
(3) return (S_0).

LEMMA 1. *Let S be a sequence with $|S| = v$ in which all bounded regions of* BR *are avoided. Then* NEXTFORWARD *computes the minimum start time $S^0 \geqq t$ such that $S|S^0$ is a sequence and the bounded regions of* BR *are avoided.*

Proof. Clearly, $S^0 \geqq t$. Since S is a sequence, $S^0 \geqq S^1$ implies that $S^0 \geqq S^i$, $1 \leqq i \leqq v$. Because $S^0 \geqq S^m + 1$ for $v \geqq m$, it follows that $S|S^0$ is a sequence. The regions are processed according to nondecreasing left endpoint; consequently, S^0 is never assigned a value that violates a region that has been previously processed. (We could have processed the bounded regions by right endpoint instead of left endpoint, as long as the approach was used consistently throughout the processing of the bounded regions.) Since S^0 is increased only if it falls within an $(m-k)$-bounded region already containing $m-k$ jobs, and since S^0 is increased the minimum amount required to avoid the region, S^0 is the minimum start time greater than or equal to t that avoids the bounded regions of BR. □

LEMMA 2. *Let S be a sequence, with $|S| = v$, in which all the bounded regions of* BR *are avoided. Then* NEXTBACKWARD *computes the maximum start time $S_0 \leqq t$ such that $S_0|S$ is a sequence and the bounded regions of* BR *are avoided.*

The proof is similar to the proof of Lemma 1.

5. Computing bounded regions by sequencing backward. Without loss of generality, assume that the jobs are sorted by release times and renamed, so that $r(1) \geqq r(2) \geqq \cdots \geqq r(n)$. In addition, let $d'(1), \cdots, d'(n)$ be the set of deadlines listed in sorted order so that $d'(1) \leqq d'(2) \leqq \cdots \leqq d'(n)$. Note that $r(i)$ remains the release time for $J(i)$ but that $d'(i)$ is almost certainly not the deadline for $J(i)$. We call i the *index* of $J(i)$.

Let $\Sigma(i, j)$ be the set of jobs with index less than or equal to j and deadlines less than or equal to $d'(i)$, $1 \leqq i, j \leqq n$, and let $n_{ij} := |\Sigma(i, j)|$. Note that all jobs in $\Sigma(i, j)$ have release times that are at least as large as $r(j)$. The BACKSEQUENCE Algorithm (presented below) iteratively constructs a set of n sequences, SE$[i]$, $1 \leqq i \leqq n$, and a set of bounded regions BR. SE$[i]$ corresponds to deadline $d'(i)$, and the jth iteration $(1 \leqq j \leqq n)$ corresponds to processing job $J(j)$ with release time $r(j)$. At the end of the jth iteration SE$[i]$ contains n_{ij} start times for the jobs $\Sigma(i, j)$. All these start times are at most $d'(i) - 1$ and are as late as possible subject to the constraint that all bounded regions already constructed are avoided. (Details are given in Theorem 1.) Since no job of $\Sigma(i, j)$ is released before $r(j)$, the algorithm stops in failure if SE$[i]_1 < r(j)$ for some i during the jth iteration. Also, if $r(j) < \text{SE}[i]_1$ and $\text{SE}[i]_1 - r(j) < 1$, for some i, then some job(s) of $\Sigma(i, j)$ must be started in the interval $[r(j), d'(i) - 1)$. This constraint triggers the creation of a bounded region with right endpoint $r(j)$. The bounded region guarantees that not too many jobs are started just prior to $r(j)$ such that they are running in the interval $[r(j), d'(i)]$ and interfering with the scheduling of $\Sigma(i, j)$.

```
SUBROUTINE BACKSEQUENCE
Initialize all sequences of SE[1..n] and BR to ∅;
(* SE[i] is the sequence ending at deadline d'(i) *)
(1) For j = 1 to n do (* Update SE[i] so that it contains precisely n_ij start times *)
Begin for loop
(2) For all i, 1 ≦ i ≦ n, such that d'(i) ≧ d(j) do
    (* d(j) is the deadline of the job with release time r(j) *)
      begin
      (* since each value of i corresponds to a different list, the order in which the
      d'(i)'s are processed is irrelevant *)
        S_0 := NEXTBACKWARD (SE[i], BR, d'(i) - 1);
        SE[i] := S_0|SE[i];
      end;
```

(3) For $1 \leqq k \leqq m$ do $f_k := \infty$;
    For $1 \leqq i \leqq n$ do
      $f_k := \min(f_k, \mathrm{SE}[i]_k)$;
(4) If $f_1 < r(j)$, then declare "infeasibility" and halt;
(5) For $k = 1$ to m do
      If $f_k - r(j) < 1$ then
        $\mathrm{BR} := \mathrm{BR} \cup \{$the $(m-k)$-bounded region $(f_k - 1, r(j))\}$.

End for loop;
Return BR;
End BACKSEQUENCE.

For the correctness proof below we use the notation BR$[j]$ to denote the set BR after the jth iteration of the global for loop of step (1). Examples of both the construction of the sequences and the creation of the bounded regions by BACKSEQUENCE are given in Figs. 1 and 2 in § 11.

THEOREM 1. *Let j be between one and n. The following statements are true at the completion of the processing of the jth iteration of the global for loop of* BACKSEQUENCE.

(i) *If* BACKSEQUENCE *does not halt in failure, then* $\mathrm{SE}[i]_1 \geqq r(j)$ *for all i such that* $\mathrm{SE}[i]_1$ *is not empty, and there is no d-sequence for $\Sigma(i, j)$ in which all regions of* BR$[j-1]$ *are avoided whose kth smallest start time is larger than* $\mathrm{SE}[i]_k$, $1 \leqq k \leqq |\mathrm{SE}[i]|$.

(ii) *All regions of* BR$[j]$ *are correct and are no more than one unit in length.*

(iii) *All regions of* BR$[j]$ *are avoided by* SE$[i]$ *for all i such that* $\mathrm{SE}[i]_1$ *is not empty.*

Proof. The proof is by induction on j. Let $j = 1$. Then clearly $\mathrm{SE}[i]_1 \geqq r(j)$ for all i such that $\mathrm{SE}[i]_1$ is not empty (which in this case is all i such that $d'(i) \geqq d(1)$). Otherwise, BACKSEQUENCE would have halted at step (4). Furthermore, any d-sequence could not have the last job in the sequence start later than $\mathrm{SE}[i]_1 = d'(i) - 1$. Since $\mathrm{BR}[0] = \varnothing$, there are no bounded regions for $\mathrm{SE}[i]_1$ to avoid. This proves condition (i) for the base case.

If $\mathrm{BR}[1] = \varnothing$, then conditions (ii) and (iii) also follow. So assume that $\mathrm{BR}[1] \neq \varnothing$. Then $f_1 - r(1) < 1$ in step (5), where $f_1 = d(1) - 1$ (since $\mathrm{SE}[i]_1 = d'(i) - 1$ for $d'(i) \geqq d(1)$). Thus, $[r(1), f_1]$ is a 1-forced region, which by Lemma 0 implies the correctness of the $(m-1)$-bounded region $(f_1 - 1, r(1))$. Note that the latter region is of length no greater than one, since otherwise we would have $f_1 < r(j)$, once again causing the algorithm to halt at step (4). This proves the correctness of (ii) for the base case. Finally, BR$[1]$ is avoided since $\mathrm{SE}[i]_1 \geqq \mathrm{SE}[1]_1 \geqq r(j)$ for all nonempty SE$[i]$, which proves the correctness of (iii).

Assume the lemma holds for $j' - 1$; we now prove it holds for j'.

(i) If SE$[i]$ is unchanged in iteration j', then $j' \notin \Sigma(i, j')$, i.e., $d(j') > d'(i)$ and by the induction assumption (i) holds for $j = j'$. Otherwise, it follows from the induction assumption together with Lemma 2 that the value of S_0 computed by NEXTBACKWARD is at least as large as the smallest feasible start time for SE$[i]$. Furthermore, if one of the other start times of SE$[i]$ were not to satisfy condition (i), then we would have a contradiction to the induction assumption that was made for $j' - 1$. Finally, since by (iii) all regions of BR$[j'-1]$ are avoided by SE$[i]$ after iteration $j' - 1$, it follows from Lemma 2 that (i) holds for $j = j'$.

(ii) Let $(f_k - 1, r(j))$ be an $(m-k)$-bounded region. Because of the way bounded regions are constructed, we have $r(j) \leqq f_k$, $f_k - r(j) < 1$, and $r(j) - (f_k - 1) < 1$. In addition, for some i, $\mathrm{SE}[i]_1 \leqq \mathrm{SE}[i]_2 \leqq \cdots \leqq \mathrm{SE}[i]_k = f_k$. (Note that $r(j) \leqq \mathrm{SE}[i]_1$.) Consequently, by condition (i), $[r(j), f_k]$ is a k-forced region, which implies by Lemma 0 the correctness of the $(m-k)$-bounded region $(f_k - 1, r(j))$.

(iii) By induction we know that all regions of BR$[j'-1]$ are avoided by SE$[i]$ after iteration $j'-1$ for nonempty SE$[i]_1$. From Lemma 2 we know that if a new start time is computed in step (2), it avoids the regions BR$[j'-1]$. Thus, SE$[i]$ avoids BR$[j'-1]$ after iteration j'. Now, any new regions added to BR$[j']$ have their right endpoints at $r(j')$. But by step (4) of BACKSEQUENCE, the values of SE$[i]$ are at least as large as $r(j')$. Consequently, these new regions are also avoided by SE$[i]$. □

Assume BACKSEQUENCE does not halt in failure. Consider the sequence SE$[i]$ after the completion of BACKSEQUENCE. As the following 1-machine example illustrates, the sequences SE$[i]$ are not necessarily d-sequences for $\Sigma(i, n)$. Let $r(1) = 1.2$, $d(1) = 4$, $r(2) = 1$, and $d(2) = 2.5$. Then SE$[2] = \{2, 3\}$. But if $J(2)$ is started at either time 2 or time 3, it will not be completed by its deadline. However, the sequences SE$[i]$ are r-sequences for $\Sigma(i, n)$, as the following trivial algorithm demonstrates. If $J(j) \in \Sigma(i, n)$, i.e., if $d'(i) \geqq d(j)$, then schedule job $J(j)$ at time S_0 computed in step (2) of BACKSEQUENCE during the jth iteration of the global for loop. Clearly, $J(j)$ will start no earlier than $r(j)$ in any of the SE$[i]$ that is updated in step (2), since otherwise BACKSEQUENCE would have halted in step (4).

COROLLARY 1. *For all bounded regions created by* BACKSEQUENCE, $(\beta-1, \alpha)$ *is an* $(m-k)$*-bounded region if and only if* $[\alpha, \beta]$ *is a* k*-forced region.*

Proof. If $[\alpha, \beta]$ is a k-forced region, then by Lemma 0, $(\beta-1, \alpha)$ is an $(m-k)$-bounded region. So suppose that $(\beta-1, \alpha)$ is an $(m-k)$-bounded region. Since all bounded regions are created in step (5) of BACKSEQUENCE, it follows that for some value of i, SE$[i]$ has k jobs starting in the region $[\alpha, \beta]$. By condition (i) of Theorem 1, it follows that it is necessary for k jobs to begin in the region $[\alpha, \beta]$. Hence, $[\alpha, \beta]$ is a k-forced region. □

COROLLARY 2. *If* BACKSEQUENCE *does not halt in failure, then after the* j*th iteration of the global loop,* SE$[i]$ *does not violate any bounded regions of the final set of bounded regions computed by* BACKSEQUENCE.

Proof. By (iii) of Theorem 1, SE$[i]$ avoids BR$[j]$; because BACKSEQUENCE does not halt in failure, SE$[i]_1 \geqq r(j)$. Since all bounded regions computed at iteration j or later have a right endpoint no greater than $r(j)$, SE$[i]$, as it is computed at the jth iteration, avoids all of BR$[n]$. □

COROLLARY 3. *If* BACKSEQUENCE *halts in failure, then there is no rd-sequence for the problem instance.*

Proof. If the algorithm halts in failure for $i = i^*$, $j = j^*$, then it follows from BACKSEQUENCE together with Theorem 1 that $r(j^*) > f_1 = \text{SE}[i^*]_1 = S_0$ (as computed in step (2) of BACKSEQUENCE). Combining the fact that all jobs in $\Sigma(i^*, j^*)$ have release time at least $r(j^*)$ together with the correctness of (i) of Theorem 1, we get that there is no *rd*-sequence for $\Sigma(i^*, j^*)$ and hence for the entire problem instance. □

6. Regions may imply additional regions. If two bounded regions overlap and together cover an interval of length greater than one, then they imply a new bounded region. We call the set of regions created by BACKSEQUENCE *original regions.*

LEMMA 3. *Let* $(\beta-1, \alpha)$ *be an original* $(m-k)$*-bounded region, and let* $(\beta'-1, \alpha')$ *be an original* $(m-k')$*-bounded region such that* $\beta-1 < \alpha' < \beta' < \alpha$. *Then* $[\alpha', \beta]$ *is a* $(k+k')$*-forced region,* $(\beta-1, \alpha')$ *is an* $(m-k-k')$*-bounded region, and both regions have length less than* 1.

Proof. Note that the $(m-k)$-bounded region $(\beta-1, \alpha)$ strictly contains the k'-forced region $[\alpha', \beta']$. By Corollary 1, $[\alpha, \beta]$ and $[\alpha', \beta']$ are k- and k'-forced regions, respectively. Since $\beta' < \alpha$, $[\alpha, \beta]$ and $[\alpha', \beta']$ do not overlap. $\beta-1 < \alpha'$ implies that

$\beta - \alpha' < 1$. Therefore, $[\alpha', \beta]$ is a $(k+k')$-forced region that, by Lemma 0, implies the $(m-k-k')$-bounded region $(\beta-1, \alpha')$ (see Fig. 4).

To prove that the regions are well defined, we must show that $k+k' \leqq m$. Assume for contradiction that $k' > m-k$. Then it follows from BACKSEQUENCE that at the completion of the iteration in which the $(m-k')$-bounded region $(\beta'-1, \alpha')$ is created there is some value I such that there are at least k' elements of SE$[I]$ within the interval $[\alpha', \beta']$. But then at least one of these start times will violate the $(m-k)$-bounded region $(\beta-1, \alpha)$. (Since $\alpha > \alpha'$, we know that the $(m-k)$-bounded region $(\beta-1, \alpha)$ is computed prior to the processing of release time α'.) Therefore, at least one value of SE$[I]$ will be set to $\beta-1$. But $(\beta-1) < \alpha'$, so BACKSEQUENCE will declare infeasibility and halt. Consequently, the region $(\beta'-1, \alpha')$ would not have been declared. Finally, the fact that α' lies between $\beta-1$ and β trivially implies that the regions $(\beta-1, \alpha')$ and $[\alpha', \beta]$ have length less than one. □

The bounded regions implied by Lemma 3 are computed by the subroutine ADD (BR), presented below. Note that the only overlapping regions that are examined by ADD are regions that are in the set BR.

SUBROUTINE ADD (BR) (* Computes the additional bounded regions implied by Lemma 3 *)
ADDITIONAL := ∅;
For all pairs of original bounded regions $(\beta-1, \alpha)$ and $(\beta'-1, \alpha')$ of BR such that $(\beta-1, \alpha)$ is an $(m-k)$-bounded region, $(\beta'-1, \alpha')$ is an $(m-k')$-bounded-region, and $\beta - 1 < \alpha' = r(j) < \beta' < \alpha$ do

ADDITIONAL := ADDITIONAL ∪ {the $(m-k-k')$-bounded region $(\beta-1, \alpha')$}.

BR := BR ∪ ADDITIONAL;
end ADD.

We call the set of regions created by ADD *additional regions.* It follows from the statement of Lemma 3 together with ADD that additional bounded regions are precisely those bounded regions that Lemma 3 proves correct. Note that to incorporate ADD into the algorithm, we need only add the following instruction to the global for loop of step (1) of BACKSEQUENCE.

(6) Call ADD (BR);

A more efficient routine for computing the regions implied by Lemma 3 called FASTADD is given in § 8.

Remark. It is easy to see that Theorem 1 and Corollaries 1, 2, and 3 still hold after the insertion of step (6) in BACKSEQUENCE. Observe that when an additional bounded region $(\beta-1, r(j))$ is created, all entries in the sequences SE[.] are at least $r(j)$. Thus, the additional regions are never violated when they are created, and they are avoided during later iterations in the same manner that the original regions are avoided.

An obvious question is whether or not the additional regions are necessary. As Figs. 3 and 4 in § 11 demonstrate, NEXTFORWARD might return a start time that violates an additional bounded region if the additional bounded regions are not incorporated into the algorithm. Consequently, the resulting sequence is not a d-sequence. Figure 4 in § 11 is an rd-sequence that does not violate the additional

bounded region. Thus, additional bounded regions are necessary if BACKSEQUENCE is used to create the regions that are used to construct r-sequences.

7. Producing the final schedule. In this section we show how to produce an rd-sequence and a schedule using the set of regions created by BACKSEQUENCE and ADD.

SUBROUTINE SCHEDULE (BR) (* constructs an rd-sequence by iteratively computing the earliest possible start time and then assigning a job to the newest start time *)
$S := \emptyset$;
For $i = 1$ to n do
(1) Let r be the minimum release time of all unscheduled jobs;
(2) $S_i :=$ NEXTFORWARD (S, BR, r); $S := S \mid S_i$;
(3) Use the Earliest Deadline Rule to select the job that starts at time S_i on machine $i \bmod m$;
end SCHEDULE.

LEMMA 4. SCHEDULE *produces an r-sequence in which all bounded regions are avoided.*

Proof. The selection of r in step (1) guarantees that there is always some unscheduled job that has been released by the time returned by NEXTFORWARD. In addition, by Lemma 1 all bounded regions are avoided. □

Let S be a sequence, and let $J(i)$ be the job that is assigned start time S_j by the Earliest Deadline Rule. We say that $J(i)$ is in *slot* j of sequence S. We also say that slot j *contains* $J(i)$ in sequence S. When speaking of slots, we shall omit the sequence name if the reference is unambiguous.

LEMMA 5. *Let S be the r-sequence produced by* SCHEDULE, *and let* $S_0 = -\infty$. *If some job in the schedule produced by* SCHEDULE *is completed after its deadline, then there exist v, w, i, and j such that*

(1) *The jobs in slots* $v, v+1, \cdots, w$ *consist exactly of the set* $\Sigma(i, j)$;
(2) $S_w + 1 > d'(i)$, *i.e., some job of* $\Sigma(i, j)$ *is finished after its deadline*;
(3) $S_{v-1} < r(j) \leqq S_v$.

Proof. Let X be the job in S that is finished later than its deadline and is in the largest numbered slot, say w, of all such jobs. That is, $d(X) = d'(i)$ for some i, and $S_w + 1 > d'(i)$. It follows from the choice of X that all jobs with deadlines no greater than $d'(i)$ are in slots numbered no greater than w.

If all jobs in slots 1 through w have deadlines no greater than $d'(i)$, then by setting $v = 1$ and $j = n$, we get that conditions (1)-(3) of the lemma are satisfied. Otherwise, let $v-1$ be the largest numbered slot less than w that contains a job with deadline greater than $d'(i)$. Let r be the minimum release time of the jobs in slots v through w, and let j be the maximum such that $r = r(j)$. Note that $\Sigma(i, j)$ consists of all jobs with release time at least $r(j)$ and deadline at most $d'(i)$. Since S is an r-sequence, it follows that $r(j) \leqq S_v$. Furthermore, since slot $v-1$ contains a job with a deadline larger than the deadlines of the jobs in slots v through w, it follows from the Earliest Deadline Rule that none of the jobs in slots v through w was available at time S_{v-1}. Therefore, $S_{v-1} < r(j) \leqq S_v$, and condition (3) of the lemma is proved. All the jobs in slots v through w are in $\Sigma(i, j)$, i.e., they must have deadlines at most $d'(i)$ (from the choice of v) and release time at least $r(j)$ (from the choice of j). Also no jobs of $\Sigma(i, j)$ are scheduled in slots 1 through $v-1$, since they are not yet released. Similarly, no jobs of $\Sigma(i, j)$ appear in slots $w+1, \cdots, n$: the jobs of $\Sigma(i, j)$ have deadlines at most

$d'(i)$, and thus they would be late if they were scheduled in slots $w+1$ through n; but w is the last slot with a late job. This shows that condition (1) holds, and completes the proof of the lemma. □

LEMMA 6. *If v, w, i, j are chosen as in Lemma 5 with the additional constraint that n_{ij} is minimal, then the following condition holds*:

(4) *If a job from slots v through w starts at its release time, then it starts at S_v.*

Proof. Assume that Lemma 5 holds for v, w, i, and j, with n_{ij} being minimal, and that condition (4) does not hold. In other words, there is some slot v', $v < v' \leqq w$, such that the job in slot v' has release time $S_{v'}$. Let j' be maximum such that $r(j') = S_{v'}$. Now conditions (1), (2), and (3) of Lemma 5 hold for v', w, i, and j', and $n_{ij'} < n_{ij}$. This contradicts the minimality of n_{ij}. □

For the remainder of this section we assume that S is the sequence constructed by SCHEDULE and that some job in the schedule produced by SCHEDULE is completed after its deadline. Let v, w, i, and j be chosen to satisfy Lemmas 5 and 6. Recall that SE$[i]$ corresponds to deadline $d'(i)$ and that the jobs in $\Sigma(i, j)$ all have deadlines less than or equal to $d'(i)$.

To simplify the notation, let F_{q-v+1} denote S_q, $0 \leqq q \leqq n$. Then the relevant start times in S for the jobs in $\Sigma(i, j)$ are $F_1, F_2, \cdots, F_{n_{ij}}$ instead of $S_v, S_{v+1}, \cdots, S_w$. Condition (3) of Lemma 5 now states that $F_0 < r(j) \leqq F_1$. (Recall that $S_0 = -\infty$.)

We compare the start times F_q to the start times of the sequence SE$[i]$ computed by BACKSEQUENCE in iteration j. Again for the sake of notation, let $B_k = \text{SE}[i]_k$ after iteration j, $1 \leqq k \leqq n_{ij}$. It follows from the construction of SE$[i]$ and from the fact that BACKSEQUENCE did not halt at step (4) that $r(j) \leqq B_1 \leqq B_2 \leqq \cdots \leqq B_{n_{ij}} \leqq d'(i) - 1$.

LEMMA 7. $F_q \leqq B_q$, $1 \leqq q \leqq n_{ij}$.

Before we prove Lemma 7, observe that since its correctness implies that $F_{n_{ij}} \leqq B_{n_{ij}} \leqq d'(i) - 1$, we get an immediate contradiction to the assumption that job X, which is scheduled in slot w in S, is finished later than its deadline of $d'(i)$. Therefore, Lemmas 4 and 7 imply the following theorem.

THEOREM 2. SCHEDULE *produces an rd-sequence and a schedule.*

Proof of Lemma 7. Assume q is the minimum value such that $F_q > B_q$. Since the value of F_q is assigned either in step (1) or in step (2) of NEXTFORWARD and there are three different possible assignments for step (1), we break the analysis into four cases.

Case 1. $F_q = F_{q-1}$ ($S^0 := S^1$ in step (1) of NEXTFORWARD). Since condition (3) of Lemma 5 implies that $F_1 > F_0$, we have in this case that $q > 1$. But $F_q > B_q \geqq B_{q-1}$ leads to $F_{q-1} > B_{q-1}$, and this contradicts the minimality of q.

Case 2. $F_q = F_{q-m} + 1$ ($S^0 := S^m + 1$ in step (1) of NEXTFORWARD). If $q - m \geqq 1$, then $F_{q-m} \leqq B_{q-m} \leqq B_q < F_q = F_{q-m} + 1$. This implies that $B_q - B_{q-m} < 1$, which contradicts the fact that SE$[i]$ is a sequence (Lemma 2).

If $q - m < 1$, then by condition (3) of Lemma 5, $F_{q-m} \leqq F_0 < r(j)$. Now, $F_q > B_q$ and $F_q = F_{q-m} + 1$ imply that $B_q < r(j) + 1$. Since $r(j) \leqq B_1 \leqq \cdots \leqq B_q < r(j) + 1$, $[r(j), B_q]$ is a q-forced region and $(B_q - 1, r(j))$ is an original $(m - q)$-bounded region. Since $B_q - 1 < F_q - 1 = F_{q-m}$ and since $F_0 < r(j)$, we have that $F_{q-m}, F_{q-m+1}, \cdots, F_0$ start in the $(m - q)$-bounded region $(B_q - 1, r(j))$. Therefore, there are $m - q + 1$ start times in the $(m - q)$-bounded region $(B_q - 1, r(j))$, and this region is violated by S. This contradicts Lemma 4, which states that the sequence produced by SCHEDULE does not violate any bounded regions.

Case 3. The job started at F_q in S has release time F_q ($S^0 := t$ and $t = r =$ minimum release time of all unscheduled jobs when NEXTFORWARD is called by

SCHEDULE). By condition (4) of Lemma 6, q can only be 1. Therefore, when NEXTFORWARD is called, all jobs of $\Sigma(i, j)$ are unscheduled. Thus, $r = F_q = r(j)$. Now the assumption that $F_1 > B_1$ implies $B_1 < r(j)$, contradicting condition (i) of Theorem 1.

Case 4. F_q received its final value in step (2) of NEXTFORWARD (to avoid violating a bounded region). This implies both that there exists an original p-bounded region $(\beta - 1, F_q)$ and that S has p start times within that region. These start times are $F_{q-p}, F_{q-p+1}, \cdots, F_{q-1}$. If $q > p$, then $F_{q-p} \leqq B_{q-p} \leqq B_{q-p+1} \leqq \cdots \leqq B_q < F_q$. This implies that SE $[i]$ violates a bounded region, which contradicts Corollary 2.

Assume that $q \leqq p$ which implies that $\beta - 1 < F_{q-p} < r(j)$. Recall that $(\beta - 1, F_q)$ is a p-bounded region. Since all $B_1, B_2, \cdots, B_q$ lie within $[r(j), B_q]$, $(B_q - 1, r(j))$ is an original $(m - q)$-bounded region. Combining inequalities, we have $\beta - 1 < r(j) < B_q < F_q$. Therefore, the conditions of Lemma 3 hold, with $\alpha = F_q$, $m - k = p$, $\beta' = B_q$, $\alpha' = r(j)$, and $k' = q$. Thus, $(\beta - 1, r(j))$ is a $(p - q)$ additional bounded region. We know that the interval $[r(j), F_q)$ contains $q - 1$ start times from S, namely $F_1, \cdots, F_{q-1}$, and that $F_0 < r(j)$. Because the p-bounded region $(\beta - 1, F_q)$ contains p start times from S, it follows that $(\beta - 1, r(j))$ contains $p - q + 1$ start times from S, which violates the additional $(p - q)$-bounded region $(\beta - 1, r(j))$. □

COROLLARY 4 (to Theorem 2). *There is no rd-sequence that has a qth smallest time smaller than the qth smallest time of the sequence S produced by* SCHEDULE, $1 \leqq q \leqq n$.

Proof. Suppose for contradiction that S' is an *rd*-sequence with start times $S'_1, S'_2, \cdots, S'_n$, and let q be the minimum value such that $S'_q < S_q$. As in Lemma 7, we analyze the four cases for the assignment of the value of S_q.

Case 1. $S_q = S_{q-1}$. This is contradicted by $S_{q-1} \leqq S'_{q-1} \leqq S'_q < S_q$.

Case 2. $q > m$ and $S_q = S_{q-m} + 1$. Again we get a contradiction since $S_{q-m} \leqq S'_{q-m} \leqq S'_q < S_q = S_{q-m} + 1$ implies that $S'_q < S'_{q-m} + 1$.

Case 3. S_q is the minimum release time of all the jobs in slots q through n in S. None of the jobs in slots q through n in S can be scheduled in slots 1 through q of S' since $S'_q < S_q$. But this is impossible since it is not possible to place $n - q + 1$ jobs in slots $q + 1$ through n of S'.

Case 4. There exists a k-bounded region $(\beta - 1, S_q)$ and there are k jobs starting in $(\beta - 1, S_q)$ in S. It follows from the definition of q that $S_{q-k} \leqq S'_{q-k} \leqq S'_{q-k+1} \leqq \cdots \leqq S'_q < S_q$, and hence $(\beta - 1, S_q)$ is violated by S', contradicting the assumption that S' is an *rd*-sequence. □

In particular, Corollary 4 implies that the output of SCHEDULE has minimum makespan over all schedules that observe the release time and deadline constraints.

8. An $O(mn^2)$ implementation of BACKSEQUENCE. We first indicate how to speed up the running time of ADD. For each value of j for which BR (j) has been computed by BACKSEQUENCE, only the largest of each of the (at most) m regions is recorded in step (5). Thus, the total number of regions is $O(mn)$. If $(\beta - 1, r(j))$ is an original k-bounded region, then all possible additional regions that are implied by $(\beta - 1, r(j))$ can be computed in $O(mn)$ time. However, since there might be m original regions computed by BACKSEQUENCE for $r(j)$, the running time could be $O(m^2n^2)$ for the entire algorithm. This can be avoided by methodically checking for overlapping regions.

Let B-REGION be an $m \times n$ array. When an original $(m - k)$-bounded region $(f_k - 1, r(j))$ is created by BACKSEQUENCE, set B-REGION $(m - k, j) := f_k - 1$, $1 \leqq k \leqq m$. Note that B-REGION $(m - 1, j) \leqq B$-REGION $(m - 2, j) \leqq \cdots \leqq B$-REGION $(0, j)$.

SUBROUTINE FASTADD (j) (* j is the j-value that has just been processed by BACKSEQUENCE *)
(1) $q := j-1$; (* start by comparing with B-REGION (*, $j-1$) *)
(2) while $q>0$ and $0<r(q)-r(j)<1$ do (* B-REGION (*, q) is the next one to check *)
(3) Compute K and W such that:
(3a) $K :=$ minimum value of k such that B-REGION $(m-k, q)<r(j)$;
(3b) $W :=$ minimum value of w such that B-REGION $(m-w, j)<r(q)-1$;
(4) If K and W are both defined then (* the necessary condition for additional bounded regions *)
for $k := K$ downto 1 do
If $f_{(m-W-k)}-1>B$-REGION $(m-k, q)$ then replace the $(m-W-k)$-bounded region $(f_{(m-W-k)}-1, r(j))$ with (B-REGION $(m-k, q)$, $r(j)$)
(* update $f_{(m-W-k)}$ because of additional region *)
(5) $q := q-1$; (* compare with the next set of regions *)
end while statement;
end FASTADD.

LEMMA 8. *The total amount of time required by all the calls to* FASTADD *is* $O(mn^2)$.

Proof. There are $O(n)$ calls to FASTADD. For each call to FASTADD there are at most $O(n)$ iterations of the while loop with each iteration requiring $O(m)$ time for step (3) and $O(m)$ time for step (4). □

The speedup of FASTADD depends on the following observation.

(*) If $(\beta-1, \alpha)$ is both an $(m-k)$-bounded region and an $(m-k')$-bounded region, for $k' \leqq k$, then the $(m-k')$-bounded region is redundant.

The correctness of (*) follows immediately from the definition of bounded regions and implies that the $(m-k')$-bounded region can be discarded.

LEMMA 9. *The additional bounded regions created by* FASTADD *are correct. Furthermore, all additional bounded regions created by* ADD *that are not created by* FASTADD *are redundant.*

Proof. Lemma 3 implies that any $(m-W-K)$-bounded region created by FASTADD for $k=K$ is correct. Since B-REGION $(m-k, q) \leqq B$-REGION $(m-K, q)$ for $k \leqq K$, it follows that B-REGION $(m-K, q)<r(j)$ implies that B-REGION $(m-k, q)<r(j)$, $k \leqq K$. Since B-REGION $(m-W, j)<r(q)-1$, the conditions of Lemma 3 hold for $k \leqq K$, and therefore the $(m-W-k)$-bounded regions created by FASTADD are correct. By (*) any bounded region implied by B-REGION $(m-w, j)$ for $w \leqq W$ is redundant. The definition of W implies that there are no additional regions created by B-REGION $(m-w, j)$ for $w>W$. □

We remind the reader that ALGORITHM BOUNDED_REGION, referred to below, is the main procedure that is presented at the beginning of this paper.

THEOREM 3. *The running time of* ALGORITHM BOUNDED_REGION *is* $O(mn^2)$.

Proof. We first show how to implement SCHEDULE in $O(mn \log n)$ time. Steps (1) and (3) of SCHEDULE cost only $O(n \log n)$ overall to implement. To implement step (2) efficiently, we make m sorted lists of bounded regions with each list containing no more than n regions as follows. For $0 \leqq k \leqq m-1$, sort the $O(n)$ k-bounded regions by their left endpoints, with ties being broken by the right endpoint. This takes $O(mn \log n)$ time. The ordered lists of regions are examined in step (2) of NEXTFORWARD starting from the 0-bounded regions and ending with the $(m-1)$-bounded regions. A pointer associated with each list of regions denotes the region most recently

examined. This pointer is updated to the next region on the list whenever a slot is moved through a region. If the start time of S^0 is increased because of a k-bounded region, step (1) of NEXTFORWARD guarantees that all start times that are subsequently computed will be at least as large as S^0. Therefore, once the start time of a slot is moved through a region, that region will never again have to be examined. Consequently, the total number of times that all regions are examined by NEXTFORWARD is $O(mn)$, and each region is examined in $O(1)$ time.

We now show that BACKSEQUENCE can be implemented in $O(mn^2)$ time. This is easy to verify for all but the calls to NEXTBACKWARD. Note that we have already shown how to implement ADD efficiently (Lemma 9). When we use the same technique as was used to implement step (2) of SCHEDULE, all calls associated with SE$[i]$ can be processed in $O(mn)$ time. Note that the bounded regions are created according to the sorted right endpoint, which is the order required by NEXTBACKWARD. For each of the n sequences SE$[i]$ the list of bounded regions needs to be scanned only once. Thus, BACKSEQUENCE has running time $O(mn^2)$. □

9. NP-completeness results. The NP-completeness proofs are reductions from the following problem that has been shown to be strongly NP-complete in [6].[1]

1/1/1 SCHEDULING.

Instance: Set T of triplets that are to be executed in the time interval $[0, 3f)$, with $|T| = t$. A triplet consists of three unit-length jobs, each of which has an integer release time that is at most $3f-1$.

Question: Is it possible to schedule f triplets of T on one machine in the time interval $[0, 3f)$ such that all $3f$ jobs are started no earlier than their release times?

For the sake of completeness of this paper we present an alternative reduction for the 1/1/1-scheduling problem to that given in [6].

THEOREM 4. 1/1/1 *scheduling is strongly* NP*-complete.*

Proof. The reduction is from the strongly NP-complete 3-partition problem [4].

3-PARTITION.

Instance: A multiset Q of $3m$ natural numbers $\{n_i: 0 \leqq i \leqq 3m-1\}$ and a natural number B, with $\sum_{i=0}^{3m-1} n_i = mB$.

Question: Can Q be partitioned into m 3-element multisets $Q_1, Q_2, \cdots, Q_m$ such that the numbers in each Q_i sum to B?

Given an arbitrary instance of 3-partition as defined above, we assume without loss of generality that $n_{p-1} \leqq n_p$, for $1 \leqq p \leqq 3m-1$. We define Q' to be the set of all possible combinations of three elements of Q that sum to B. Each triple of numbers (n_i, n_j, n_k) of Q' corresponds to a triplet of jobs $(J(i), J(j), J(k))$ with release times i, j, and k, respectively. Let T be all such corresponding triplets of jobs. The instance of 1/1/1 scheduling consists of T and the interval $[0, 3m)$, with $f = m$.

Note that $|T| < (3m)^3$ and $r(i) \leqq 3m-1$, for all jobs of T. Thus, even with a unary encoding of the release times this reduction is polynomial.

Given a solution to the 3-partition problem, it is easy to produce a schedule for the 1/1/1-scheduling problem. Each Q_i in the solution to the 3-partition instance corresponds to a triplet of T. We simply schedule the jobs in the corresponding triplets at their release times. The resulting schedule maps the jobs onto the start times $\{0, \cdots, 3m-1\}$.

[1] The definition of 1/1/1 scheduling given in [6] uses deadlines instead of release times as is done here. However, the definitions are symmetric.

Given triplets $T_1, \cdots, T_m$ and a one-processor schedule SCH of the corresponding $3m$ jobs onto $\{0, \cdots, 3m-1\}$, we have

$$\sum_{i=0}^{3m-1} n_i = \sum_{k=1}^{m} \sum_{J(i)\in T_k} n_{s(i)} = mB.$$

Since no job starts before its release time and since $n_{p-1} \leqq n_p$, it follows that

$$\sum_{J(i)\in T_k} n_{s(i)} \geqq \sum_{J(i)\in T_k} n_{r(i)} = B \quad \text{for } 1 \leqq k \leqq m.$$

We conclude that $\sum_{J(i)\in T_k} n_{s(i)} = B$, for $1 \leqq k \leqq m$, and thus the m sets $\{n_{s(i)}: J(i) \in T_k\}$ constitute a solution to the instance of 3-partition. □

PROBLEM A. *Instance*: m identical machines and n jobs, each job having a release time of 0, an integer deadline, and length 1, 3, or q, for some integer q.

Question: Does there exist a valid schedule for the set of jobs?

THEOREM 5. *Problem* A *is strongly* NP-*complete.*

Proof. Let $T = \{T(i): 0 \leqq i \leqq t-1\}$ and f be an instance of 1/1/1 scheduling. Denote the triplet $T(i)$ as $(T(i, 1), T(i, 2), T(i, 3))$, and the release time of $T(i, j)$ as $r(i, j)$. Without loss of generality we assume that $r(i, 1) < r(i, 2) < r(i, 3)$. (If $r(i, j) = r(i, k)$, then one of these release times can be increased by one since the jobs are identical.)

The corresponding instance of Problem A consists of t machines, numbered $0, 1, \cdots, t-1$, and the following set of jobs (see Fig. 5 in § 11):

(1) A set of *filler jobs* $F = \bigcup_{i=1}^{4(t-1)} F(i)$. The subset $F(i)$ contains $t - \lceil i/4 \rceil$ filler jobs $F(i, j)$ each of which has length one such that $d(F(i, j)) = i$, $1 \leqq j \leqq t - \lceil i/4 \rceil$.

(2) A set of *chain jobs* $C = \bigcup_{i=0}^{t-1} C(i)$. The ith chain $C(i)$ corresponds to the triplet $T(i)$. It contains $3f+2$ chain jobs $C(i, j)$, $1 \leqq j \leqq 3f+2$, each of which has length $q = 4t$ such that

$$\begin{aligned} d(C(i, j)) &= 4i + jq && \text{for } 1 \leqq j \leqq r(i, 1) + 1, \\ &= 4i + jq + 1 && \text{for } r(i, 1) + 1 < j \leqq r(i, 2) + 1, \\ &= 4i + jq + 2 && \text{for } r(i, 2) + 1 < j \leqq r(i, 3) + 1, \\ &= 4i + jq + 3 && \text{for } r(i, 3) + 1 < j \leqq 3f + 2. \end{aligned}$$

(3) A set of *interval jobs* $I = \{I(j): 1 \leqq j \leqq 3f\}$ each of which has length 1, with $d(I(j)) = (j+1)q - 1$. $I(j)$ corresponds to the interval $[j-1, j)$ in the 1/1/1 schedule.

(4) A set of *pusher jobs* $P = \{P(i): 1 \leqq i \leqq t-f\}$ all of which have length 3, and deadline $(3f+2)q - 1$. The pusher jobs correspond to the unscheduled triplets.

Let T' be a solution to the instance of 1/1/1 scheduling, that is, T' is a subset of f triplets of T for which there exists a schedule $\mathrm{SCH}_{1/1/1}$ on one machine with all jobs appearing in the interval $[0, 3f)$. We construct a schedule $\mathrm{SCH}_{\mathrm{A}}$ for the corresponding instance of Problem A. (See Fig. 5 in § 11.)

Machine i, $0 \leqq i \leqq t-1$, contains $4i$ filler jobs in the interval $[0, 4i)$. The chain $C(i)$, for $0 \leqq i \leqq t-1$, is scheduled on machine i. We distinguish between two cases:

Case $T(i) \notin T'$. Job $C(i, j)$, for $1 \leqq j \leqq 3f+1$, is scheduled on machine i finishing at $4i + jq$. The only exception is the last chain job $C(i, 3f+2)$ of $C(i)$. It finishes at $4i + (3f+2)q + 3$ instead of three time units earlier so that a pusher job can be scheduled in the interval $[4i + (3f+1)q, 4i + (3f+1)q + 3)$ on machine i.

Case $T(i) \in T'$. In this case three interval jobs of length 1 are scheduled on machine i instead of one pusher job. Without loss of generality we can assume that $T(i, 1)$ precedes $T(i, 2)$ and $T(i, 2)$ precedes $T(i, 3)$ in $\mathrm{SCH}_{1/1/1}$, since $r(i, 1) < r(i, 2) < r(i, 3)$. Otherwise, we could swap the jobs of $T(i)$ to make them appear in the above order.

Assume that $T(i, 1)$, $T(i, 2)$, and $T(i, 3)$ are scheduled in the intervals $[j-1, j)$, $[k-1, k)$, and $[l-1, l)$, respectively, in $\mathrm{SCH}_{1/1/1}$. Schedule the interval jobs $I(j)$, $I(k)$, and $I(l)$ on machine i such that they finish at times $4i+jq+1$, $4i+kq+2$, and $4i+lq+3$, respectively. Schedule the chain jobs of $C(i, r)$ on machine i such that they finish at the following times:

$$4i+rq \quad \text{for } 1 \leqq r \leqq j,$$

$$4i+rq+1 \quad \text{for } j < r \leqq k,$$

$$4i+rq+2 \quad \text{for } k < r < 3f+2,$$

$$4i+rq+3 \quad \text{for } r = 3f+2.$$

This completes the construction of SCH_A. Note that all jobs of the instance of A are completed before or at their deadlines.

To show the reverse, suppose we are given a schedule SCH_A for the instance of A. We want to show that SCH_A determines a subset T' of f triplets of T such that there exists a schedule $\mathrm{SCH}_{1/1/1}$ for T' on one machine in the interval $[0, 3f)$.

CLAIM A. (i) The jobs $C(i, 1)$, for $0 \leqq i \leqq t-1$, are scheduled on different machines in SCH_A. Without loss of generality, assume $C(i, 1)$ is scheduled on machine i.

(ii) The filler jobs are scheduled in SCH_A such that the interval $[0, 4i)$ of machine i is occupied with filler jobs and $C(i, 1)$ is scheduled on machine i such that it finishes at $4i+q$, $0 \leqq i \leqq t-1$. (See Fig. 5 in § 11.)

(iii) All jobs of the chain $C(i)$ are scheduled on the machine i in SCH_A. The job $C(i,j)$ can have one of at most four different finishing times in SCH_A: $4i+jq$, $4i+jq+1$, $4i+jq+2$, or $4i+jq+3$.

(iv) Machine i, $0 \leqq i \leqq t-1$, contains either three interval jobs of length one, or one pusher job of length 3.

(v) The set $T' = \{T(i)$: machine i in SCH_A contains interval jobs$\}$ is a solution of the instance of 1/1/1 scheduling. Assume SCH_A is normalized, where by normalized we mean that no interval job can be moved to the right by *swapping* it with the chain job or interval job that is following it on the same machine. Then $I(j)$, for $1 \leqq j \leqq 3f$, is scheduled in the interval $[jq, (j+1)q-1)$ in SCH_A. We can construct a schedule $\mathrm{SCH}_{1/1/1}$ for T' as follows. If $I(j)$ is the kth interval job on machine i, $1 \leqq k \leqq 3$, then schedule job $T(i, k)$ in the interval $[j-1, j)$ of $\mathrm{SCH}_{1/1/1}$.

Proof of (i). Assume two jobs $C(k, 1)$ and $C(k', 1)$, $k < k'$, are scheduled on the same machine in SCH_A. Since chain jobs have length q, the earliest time both jobs can finish is time $2q$. This contradicts the assumption that there is a feasible schedule for Problem A since, for $0 \leqq i \leqq t-1$, $d(C(i, 1)) \leqq d(C(t-1, 1)) = 4(t-1)+q < 2q$.

Proof of (ii). The job $C(0, 1)$ has to start at time 0 and finish at its deadline q. Assume $1 \leqq j \leqq 4$. Then, since $|F(j)| = t-1$, and the jobs of $F(j)$ have deadline j, the intervals $[j-1, j)$ of machines 1 through $t-1$ of SCH_A must be occupied with the filler jobs of $F(j)$. This implies that $C(1, 1)$ is started at time 4 in SCH_A and finished at its deadline $4+q$. Simple induction on j implies that the jobs of $F(j)$ are scheduled on machine $\lceil j/4 \rceil$ through t and that $C(i, 1)$ starts at $4i$ and finishes at its deadline $4i+q$.

Proof of (iii). We define the lexicographic order $\leqq$ on the tuples (i, j) as follows. If $i < t-1$, then (i, j) immediately precedes $(i+1, j)$; otherwise, $i = t-1$ and $(t-1, j)$

immediately precedes $(0, j+1)$. We use this lexicographic order to do an induction on the index tuple (i, j) of $C(i, j)$.

The base case that (iii) holds for $(i, j) \leqq (t-1, 1)$ follows from claim (ii). Assume that (iii) holds for all jobs $C(i, j)$ such that $(i, j) \leqq (i', j')$. Let (i^*, j^*) be the next tuple after (i', j') in the lexicographic ordering. By induction we know that machine i is busy in the interval $[4i+(j-1)q+3, 4i+jq)$, for $(i, j) \leqq (i', j')$, since it is executing $C(i, j)$. In particular, all machines i such that $i \neq i^*$ are busy in the interval $[4i^*+(j^*-1)q+3, 4i^*+(j^*-1)q+4)$. Since $d(C(i^*, j^*)) \leqq 4i^*+j^*q+3$, we conclude that $C(i^*, j^*)$ has to be executed on machine i^*. Clearly, $C(i^*, j^*)$ has to be completed by its deadline, since $\mathrm{SCH_A}$ is a valid schedule. The fact that $C(i^*, j^*)$ has to be finished at time $4i^*+j^*q$ or later follows by induction, since $C(i^*, j^*-1)$ is finished at time $4i^*+(j^*-1)q$ or later. This completes the proof of (iii).

Proof of (iv). From (i) and (iii), we know that machine i of $\mathrm{SCH_A}$ receives $4i$ filler jobs of length 1 and $3f+2$ chain jobs of length q. The deadlines of the pusher jobs and interval jobs are no greater than $(3f+2)q-1$. Therefore, the last job on machine i, for $0 \leqq i \leqq t-1$, is the chain job $C(i, 3f+2)$. Since $d(C(i, 3f+2)) = 4i+(3f+2)q+3$, it follows that there are exactly three units available on machine i for interval and pusher jobs. This gives us $3t$ time units for interval and pusher jobs on all t machines. Since there are $3f$ interval jobs of length 1 and $t-f$ pusher jobs of length 3, we conclude that each machine i of $\mathrm{SCH_A}$ contains either three interval jobs or one pusher job.

Proof of (v). Assume $\mathrm{SCH_A}$ is normalized and assume by contradiction that $I(j)$ is an interval job that is scheduled in the interval $[lq, (l+1)q-1)$, for some $l<j$. If $I(j)$ is scheduled on machine i, then by (iii), it has to be scheduled between the chain jobs $C(i, l)$ and $C(i, l+1)$ in the interval $[4i+lq, 4i+lq+3)$. Since $d(C(i, l+1)) \leqq d(C(t-1, l+1)) \leqq (l+2)q-1 \leqq d(I(j))$, we can move $I(j)$ past $C(i, l+1)$ as follows. Iteratively swap $I(j)$ with the interval jobs between $I(j)$ and $C(i, l+1)$, and finally, swap $I(j)$ with $C(i, l+1)$. The fact that $I(j)$ could be swapped with a chain or interval job to its right contradicts the fact that $\mathrm{SCH_A}$ is normalized. We conclude that $l=j$ and that $I(j)$ is scheduled in the interval $[jq, (j+1)q-1)$ in $\mathrm{SCH_A}$.

To complete the proof of (v), we need to show that the constructed schedule $\mathrm{SCH}_{1/1/1}$ is a valid schedule for T', that is $r(i, k) \leqq j-1$, for $1 \leqq k \leqq 3$. By (iii), we know that $I(j)$ is scheduled between $C(i, j)$ and $C(i, j+1)$ in the interval $[4i+jq, 4i+jq+3)$ on machine i. Since $k-1$ interval jobs are scheduled before $I(j)$ and $3-k$ after $I(j)$ on machine i, it follows that $I(j)$ is executed in the interval $[4i+jq+k-1, 4i+jq+k)$ on machine i. This implies that $d(C(i, j+1)) \geqq 4i+(j+1)q+k$ and therefore by the definition of the deadlines of the chain jobs, we have $r(i, k)+1<j+1$, which is equivalent to $r(i, k) \leqq j-1$. This completes the proof of Claim A and the theorem. □

In Problem B instead of having jobs of length 3 as in Problem A we have arbitrary integer release times as well as arbitrary integer deadlines.

PROBLEM B. *Instance*: m identical machines and n jobs, each job having a release time and deadline that are arbitrary integers, and having length 1 or q, for some integer q.

Question: Does there exist a valid schedule for the set of jobs?

The reduction for B is similar to the reduction for A. The jobs of length 3 in the reduction of A are replaced in B by an additional set of jobs R, all of which have length 1 or q.

THEOREM 6. *Problem* B *is strongly* NP*-complete.*

Proof. In Problem A the length-3 jobs were used to guarantee that either three or zero interval jobs were scheduled on a machine. In the latter case the machine received

a pusher job of length 3. In the reduction below the same set of jobs is used as in Theorem 5 except that the length-3 pusher jobs are replaced by R.

The *slack* $\Delta(J)$ of job J is defined to be the difference $d(J)-r(J)-p(J)$. The jobs of R have release time at least $(3f+2)q$. Thus, for any machine i, $0 \leqq i \leqq t-1$, in a valid schedule all jobs of R follow all other jobs. The set R consists of the following sets of jobs:

(1) A set of *frame jobs* $A=\{A(i): 0 \leqq i \leqq t-1\}$, with $r(A(i))=(3f+t+3)q+4i+3$, $d(A(i))=r(A(i))+q$, and $p(A(i))=q$. Note that $\Delta(A(i))=0$.

(2) A set of *glider jobs* $G=\bigcup_{i=1}^{t} G(i)$, with $G(i)=\{G(i,j): j \neq t-i, 0 \leqq j \leqq t-1\}$, $r(G(i,j))=(3f+2+j)q+4i$, $d(G(i,j))=r(G(i,j))+q+3$, and $p(G(i,j))=q$. Note that $\Delta(G(i,j))=3$.

(3) A set of *early jobs* $E=\{E(i): 0 \leqq i \leqq t-1\}$, with $r(E(i))=(3f+2)q+4i$, $d(E(i))=(3f+t-i+3)q+4i+3$, and $p(E(i))=q$.

(4) A set of *late jobs* $L=\{L(i): 0 \leqq i \leqq t-1\}$ with $r(L(i))=r(E(i))+3$, $d(L(i))=d(E(i))-3$, and $p(L(i))=q$.

(5) A set of *stuffer jobs* $U=\{U(i): 1 \leqq i \leqq 3(t-f)\}$, with $r(U(i))=(3f+t+3)q$, $d(U(i))=(3f+t+4)q-1$, and $p(U(i))=1$.

The jobs of R can be scheduled such that $A(i)$, $G(i)$, $E(i)$, and $L(i)$, for $0 \leqq i \leqq t-1$, appear on machine i in one of the following two ways. (See Fig. 6 in § 11.)

Case 1. $E(i)$ starts at $(3f+2)q+4i$ on machine i. $E(i)$ is followed by $G(i,1)$, $G(i,2), \cdots, G(i,t-i-1)$, $L(i)$, $G(i,t-i+1), \cdots, G(i,t)$, three stuffer jobs, and $A(i)$.

Case 2. $L(i)$ starts at $(3f+2)q+4i+3$ on machine i. $L(i)$ is followed by $G(i,1)$, $\cdots, G(i,t-i-1)$, $E(i)$, $G(i,t-i+1), \cdots, G(i,t)$, $A(i)$.

Since $|U|=3(t-f)$, case 1 occurs exactly $t-f$ times and case 2 occurs f times.

To show that R has to be scheduled as outlined above, we prove the following claim (see also Fig. 6 in § 11). Note that the jobs of R have the same function as the pusher jobs of the previous reduction.

CLAIM B. For $0 \leqq i \leqq t-1$, (i) Machine i receives $t+2$ jobs of length q of R. The last job of R of length q on machine i is $A(i)$.

(ii) The vth job of R of length q on machine i starts either at time $(3f+2+v-1)q+4i$ or no more than three time units later. The job $G(i,j)$, for $j \neq t-i$, $1 \leqq j \leqq t$, is the $(j+1)$st job of R of length q on machine i.

(iii) The jobs $E(i)$ and $L(i)$ share the first and $(t-i+1)$st position of the length q jobs of R on machine i.

(iv) The first job of R is an early job on eactly $(t-f)$ machines, and a late job on the remaining f machines. In the case where the first job on machine i is $E(i)$, the start time for $E(i)$ is $(3f+2)q+4i$, and three stuffer jobs are scheduled before $A(i)$. In the case where the first job on machine i is $L(i)$, $L(i)$ begins at $(3f+2)q+4i+3$.

Proof of (i). Since all frame jobs $A(i)$ start at $(3f+t+3)q+4i+3$ and run for $q=4t$ units of time, exactly one of the t frame jobs has to be scheduled on each of the t machines. In statement (iii) of Claim A (of the reduction for problem A), it was shown that the last job not in R on machine i finishes either at $(3f+2)q+4i$ or no more than three time units later. Thus, if machine i were to run more than $t+2$ jobs of R of length q, then the $(t+3)$rd job could not be finished earlier than time $(3f+t+5)q+4i$. Since all deadlines of the jobs of R are smaller than $(3f+t+5)q$, we conclude that each machine runs at most $t+2$ jobs of R of length q. Because R contains t^2+2t jobs of length q and there are t machines, each machine receives exactly $t+2$ jobs of R of length q.

From part (iii) of Claim A, we know that the first job of R of length q on machine i can start no earlier than $(3f+2)q+4i$. Thus the $(t+2)$nd such job can start no earlier

than $(3f+t+3)q+4i$. By a simple induction, it can be shown that this implies that $A(i)$ is scheduled on machine i.

Proof of (ii). Since $r(G(i,j))=(3f+j+2)q+4i$ and $\Delta(G(i,j))=3$, the second part of statement (ii) is implied by the first part. To prove the first part, note that by statement (iii) of Claim A the first job of length q of R can start no earlier than $(3f+2)q+4i$. Additionally, by statement (i) above, machine i receives $t+2$ jobs of R of length q, with the last such job starting at time $(3f+t+3)q+4i+3$.

Proof of (iii). By (i) and (ii) above, we know that every job of E and L is scheduled as the first or $(t-i+1)$st job of R of length q on some machine i. It is easy to see that the first and tth job of R of length q of machine 0 are $E(0)$ and $L(0)$. No other job of E and L is released early enough to be the first, and no job of E or L has a deadline large enough to be the $(t+1)$st job of machine i. By a simple induction it can be shown that $E(i)$ and $L(i)$ are the first and the $(t-i+1)$st jobs of R of length q on machine i.

Proof of (iv). If $L(i)$ is the first job of machine i, then by (ii) above, it must start at its release time of $(3f+2)q+4i+3$. In this case, no stuffer jobs can be scheduled on machine i.

If $L(i)$ is the $(t-i+1)$st job on machine i, then by (ii) above it must finish at its deadline of $(3f+t-i+3)q+4i$. This implies that $E(i)$ is started at its release time of $(3f+2)q+4i$ and that no more than three stuffer jobs can be scheduled in the interval $[(3f+t+3)q+4i,\ (3f+t+3)q+4i+3)$. Since there are $3(t-f)$ stuffer jobs of length 1, $E(i)$ must be the first job of R on at least $t-f$ machines.

Using arguments similar to the reduction of Problem A, the existence of $3f$ interval jobs implies that there are at least f machines whose last job finishes later than $(3f+2)q+4i$. We conclude that exactly $t-f$ machines start with a job from $E(i)$, and the remaining f machines start with a job from $L(i)$.

This completes the proof of Claim B. It follows from the proof for Problem A that the f machines starting with a job from $L(i)$ will contain the interval jobs. □

10. Open problems. (1) We conjecture that ALGORITHM BOUNDED_REGION can be modified to obtain a running time of $O(mn\log n)$. We have already shown that SCHEDULE runs in $O(mn\log n)$ time. It is also possible, using a technique of [5] to avoid computing the additional regions. This technique determines whether or not start times being computed by BACKSEQUENCE *might* violate a newly computed bounded region by comparing the start times mod 1. If the answer is determined to be yes, then the start times are decremented accordingly. It can be shown that this process will not cause a feasible problem instance to be incorrectly designated infeasible. It is also straightforward to show that this precomputation corresponds to the additional regions.

In [5], it was shown that the running time for the single machine version of this problem is $O(n\log n)$. This running time was obtained by eliminating the need to compute all sequences for those jobs with deadline at least as great as the deadline of the job whose release time is currently being processed. We believe that similar techniques should work for the general problem, but we have not been able to solve the problem of how to capture the notion of different degrees of boundedness, i.e., the fact that we have regions with restrictions on the number of jobs that can start within them.

(2) Can the NP-completeness proofs be made tighter, or are there polynomial time algorithms for the more constrained problems? In particular, what can be said about variations of Problems A and B in which there is only a single machine or there

are a constant number of machines. There ia polynomial time algorithm for the single machine version of Problem B [2], but we have been unable to generalize the algorithm to the case where the running time of a job is one of two arbitrary integers, as opposed to being either one or an arbitrary integer.

11. Figures. We use the example below to illustrate the BOUNDED_REGION Algorithm.

Example $m = 2$, $n = 7$. The release time and deadline of a job are listed as an ordered pair after the job:

$A(0, 4.4)$, $B(0.2, 2.2)$, $C(0.3, 2.3)$, $D(0.5, 1.8)$, $E(1.6, 3.4)$, $F(2.4, 3.6)$, $G(2.4, 4.0)$.

The table constructed for the above example is given in Fig. 2. In column j we list only those entries that are changed in iteration j.

$BR[1] = ((1.6, 2.4), \varnothing)$ $BR[2] = ((1.6, 2.4), (2, 2.4))$
$BR[3] = ((1, 1.6), \varnothing)$ $BR[4] = ((-0.2, 0.5), \varnothing)$
$BR[5] = ((-0.2, 0.3), \varnothing)$ $BR[6] = ((-0.7, 0.2), (0.0, 0.2))$
$BR[7] = ((-0.7, 0.0), \varnothing)$

FIG. 1. *The original bounded regions computed by* BACKSEQUENCE.

release times / deadlines		F	G	E	D	C	B	A
		2.4	2.4	1.6	0.5	0.3	0.2	0.0
D	1.8				0.8			
B	2.2				1.2		1.0 [3]	
C	2.3				1.3	1.0 [2]	0.3	
E	3.4			2.4	2.4	1.4	1.0 [4]	
F	3.6	2.6		2.6	1.6	1.6	0.6	
G	4.0	3.0	3.0	2.0	1.6 [1]	1.0	0.6	
A	4.4	3.4	3.4	2.4	2.4	1.4	1.0 [4]	0.4

[1] was 2.0
[2] was 1.3
[3] was 1.2
[4] was 1.4

FIG 2. *The sequences computed by* BACKSEQUENCE.

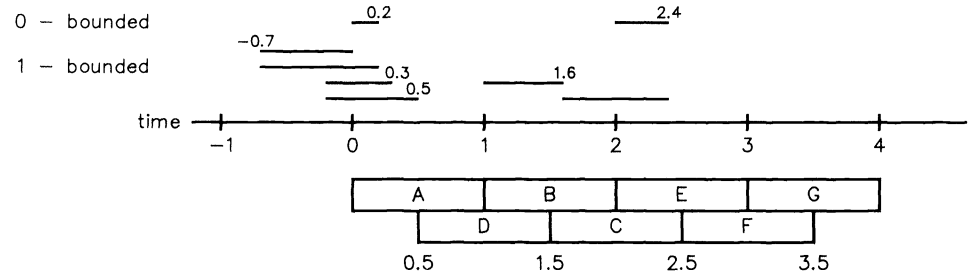


FIG 3. *Output of* SCHEDULE (BR), where BR *consists only of original regions.*

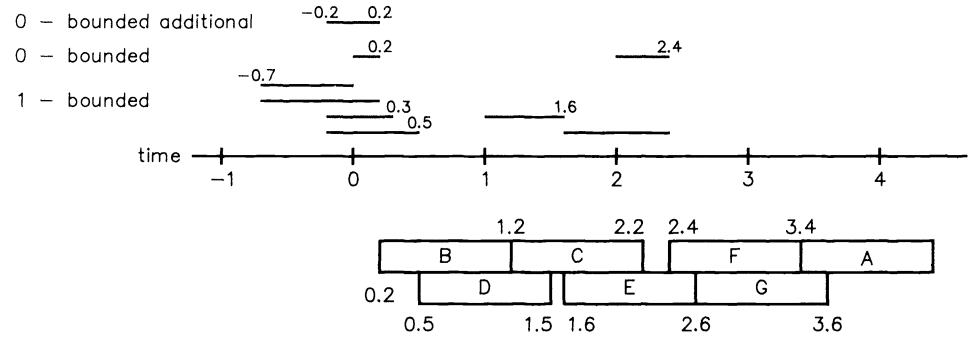
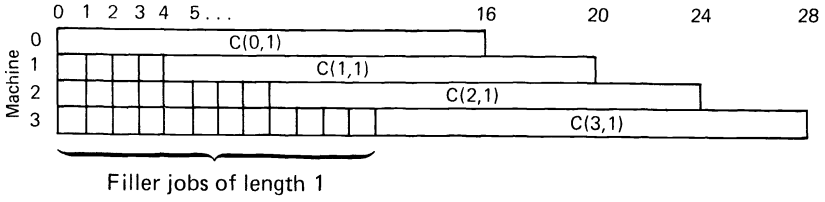
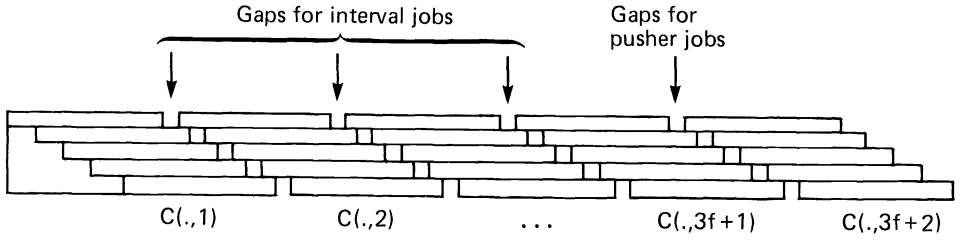


FIG. 4. *The final rd-sequence produced by* SCHEDULE (BR), *with additional regions included in* BR.

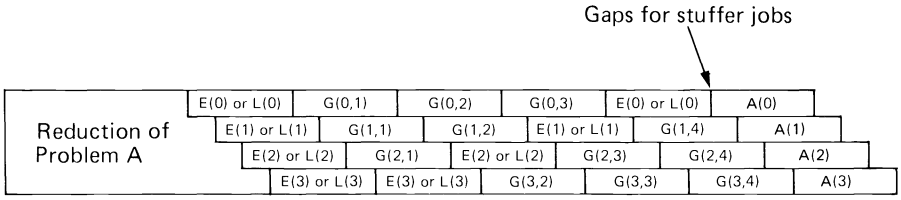


The beginning of a schedule for m = 4



Outline of SCH_A for m = 5

FIG. 5. SCH_A.



Outline of SCH_B for m = 4

FIG. 6. SCH_B.

The bounded regions computed at each iteration of the for loop of BACK-SEQUENCE are listed in Fig. 1. The first entry in the ordered pair is the 1-bounded region, the second is the 0-bounded region. The symbol $\varnothing$ is used when there is no bounded region.

Figure 3 shows how NEXTFORWARD avoids the original bounded regions. For the purpose of illustration, we ignore the additional regions. Observe that if only the original bounded regions listed above are used, C is completed later than its deadline, so the sequence is not a d-sequence. This follows because BR[4] and BR[6] satisfy the conditions of Lemma 3 and hence create the additional 0-bounded region $(-0.2, 0.2)$.

Acknowledgments. The authors are very grateful to Danny Dolev for the many hours he spent discussing the algorithm with us. We also are very appreciative of the comments and criticisms we received from the two very thorough referees.

REFERENCES

[1] J. CARLIER, *Problème à une machine dans le cas où les tâches ont des durées égales*, Tech. Report, Institut de Programmation, Université de Paris VI, Paris, France, 1979.

[2] D. DOLEV, B. SIMONS, AND M. WARMUTH, private communication.

[3] G. FREDERICKSON, *Scheduling unit-time tasks with integer release times and deadlines*, Inform. Process. Lett., 16 (1983), pp. 171-173.

[4] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, CA, 1978.

[5] M. R. GAREY, D. S. JOHNSON, B. B. SIMONS, AND R. E. TARJAN, *Scheduling unit-time tasks with arbitrary release times and deadlines*, SIAM J. Comput., 10 (1981), pp. 256-269.

[6] H. GABOW AND M: STALLMANN, *Scheduling multitask jobs with deadlines on one processor*, manuscript.

[7] H. 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.

[8] J. R. JACKSON, *Scheduling a production line to minimize maximum tardiness*, Res. Report 43, Management Science Research Project, University of California, Los Angeles, CA, 1955.

[9] J. K. LENSTRA, A. G. H. RINNOOY KAN, AND P. BRUCKER, *Complexity of machine scheduling problems*, Ann. Discrete Math., 1 (1977), pp. 343-362.

[10] B. B. SIMONS, *A fast algorithm for single processor scheduling*, in 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Long Beach, CA, 1980, pp. 246-252.

[11] ———, *Multiprocessor scheduling of unit length jobs with arbitrary release times and deadlines*, SIAM J. Comput., 12 (1983), pp. 294-299.

[12] J. D. ULLMAN, NP-*complete scheduling problems*, J. Comput. System Sci., 10 (1975), pp. 384-393.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 711-739, August 1989

006

MINIMUM-KNOWLEDGE INTERACTIVE PROOFS FOR DECISION PROBLEMS*

ZVI GALIL†¶, STUART HABER‡¶, AND MOTI YUNG§¶[1]

Abstract. Interactive communication of knowledge from the point of view of resource-bounded computational complexity is studied. Extending the work of Goldwasser, Micali, and Rackoff [*Proc. 17th Annual ACM Symposium on the Theory of Computing*, 1985, pp. 291-304; *SIAM J. Comput.*, 18 (1989), pp. 186-208], the authors define a protocol transferring the result of any fixed computation to be *minimum-knowledge* if it communicates no additional knowledge to the recipient besides the intended computational result. It is proved that such protocols may be combined in a natural way so as to build more complex protocols.

A protocol is introduced for two parties, a prover and a verifier, with the following properties:

(1) Following the protocol, the prover gives to the verifier a proof of the value, 0 or 1, of a particular Boolean predicate, which is (assumed to be) hard for the verifier to compute. Such a *deciding* "interactive proof-system" extends the interactive proof-systems of [*op. cit.*], which are used only to confirm that a certain predicate has value 1.

(2) The protocol is minimum-knowledge.

(3) The protocol is *result-indistinguishable*: an eavesdropper, overhearing an execution of the protocol, does not learn the value of the predicate that is proved.

The value of the predicate is a cryptographically secure bit, shared by the two parties to the protocol. This security is achieved without the use of encryption functions, all messages being sent in the clear. These properties enable one to define a cryptosystem in which each user receives exactly the knowledge he is supposed to receive, and nothing more.

Key words. zero knowledge, interactive proof systems, minimum knowledge cryptographic protocols, cryptography, security, probabilistic computations, factoring, quadratic residuosity, number theory

AMS(MOS) subject classifications. 11Z50, 68Q99, 94A60

1. Introduction.

Transfer and exchange of knowledge is the basic task of any communication system. Recently, much attention has been given to the process of knowledge exchange in the context of distributed systems and cryptosystems. In particular, several authors have concentrated on problems associated with the interactive communication of proofs [17], [1], [24].

In [17] Goldwasser, Micali, and Rackoff developed a computational-complexity approach to the theory of knowledge: a message is said to convey knowledge if it contains information that is the result of a computation that is intractable for the receiver. They introduce the notion of an *interactive proof-system* for a language L. This is a protocol for two interacting probabilistic Turing machines, whereby one of them, the *prover*, proves to the other, the *verifier*, that an input string x is in fact (with

* Received by the editors October 17, 1986; accepted for publication (in revised form) October 10, 1988. A preliminary version of the paper appeared as *A Private Interactive Test of a Boolean Predicate and Minimum Knowledge Public-Key Cryptosystems*, Proceedings of the 26th Annual IEEE Symposium on the Foundations of Computer Science, 1985, pp. 360-371. Most of this work was done while all three authors were at Columbia University, New York, New York 10027.

Editor's Note. This paper was originally scheduled to appear in the February 1988 Special Issue on Cryptography (SIAM J. Comput., 17 (1988)).

† Department of Computer Science, Columbia University, New York, New York 10027, and Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

‡ Bell Communications Research, Morristown, New Jersey 07960.

§ IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York 10598.

¶ The work of these authors was supported in part by National Science Foundation grants MCS-8303139 and DCR-851173.

[1] The work of this author was supported in part by an IBM graduate fellowship.

very high probability) an element of L. The verifier is limited to tractable (i.e., probabilistic polynomial-time) computations. We do not limit the computational power of the prover; in the cryptographic context, the prover may possess some secret information—for example, the factorization of a certain integer N. (This is analogous to the following model of a "proof-system" for a language L in NP: given an instance $x \in L$, an NP *prover* computes a string y and sends it to a deterministic polynomial-time *verifier*, which uses y to check that indeed $x \in L$.)

Goldwasser, Micali, and Rackoff called an interactive proof-system for L *zero-knowledge* if it releases no additional knowledge—that is, nothing more than the one bit of knowledge given by the assertion that $x \in L$ [17]. Extending their definition, we consider all two-party protocols for the purpose of transferring from one party to the other the result of a specified computation—$y = f(x)$, say—depending on the input x, and call any such protocol *minimum-knowledge* if it releases nothing more than the assertion that $y = f(x)$. Naturally, such interactive protocols are of particular interest in a cryptographic setting where distrustful users with unequal computing power communicate with each other.

After giving our definition of minimum-knowledge protocols, we prove that the concatenation of two minimum-knowledge protocols is minimum-knowledge. This suggests the importance of the minimum-knowledge property for the modular design of complex protocols. In fact, it is by serially composing several minimum-knowledge subprotocols that we formulate the more complex minimum-knowledge protocol that we introduce in this paper.

In this paper we extend the ability of interactive proof-system protocols from *confirming* that a given string x is in a language L to *deciding* whether $x \in L$ or $x \notin L$. That is, we give the first (nontrivial) example of a language L so that both L and its complement have minimum-knowledge interactive proof-systems for confirming membership, where both the proof of membership in L and the proof of nonmembership in L are by means of the *same* protocol, which releases no more knowledge than the value of the membership bit ($x \in L$).

Furthermore, by following the protocol, the *prover* demonstrates to the *verifier* either that $x \in L$ or that $x \notin L$ in such a way that the two cases are indistinguishable to an eavesdropping third party that is limited to feasible computations. In fact, the protocol releases *no knowledge at all* to such an eavesdropper. As usual, we assume that the eavesdropper knows both the prover's and the verifier's algorithms, and we allow him access to all messages passed during an execution of the protocol. In spite of the fact that our protocol makes no use of encryption functions, the eavesdropper receives *no knowledge* about whether he has just witnessed an interactive proof of the assertion that $x \in L$ or of the assertion that $x \notin L$. We call this property of our protocol *result-indistinguishability.*

The proof that our protocol is minimum-knowledge with respect to the verifier and result-indistinguishable with respect to the eavesdropper relies on no unproved assumptions about the complexity of a number-theoretic problem.

The work of [17], [1], [24] concentrates on the knowledge transmitted by a prover to an active verifier. Introducing a third party to the scenario, we analyze the knowledge gained both by an active verifier and by a passive eavesdropper.

If membership or nonmembership in L is an intractable computation, then a result-indistinguishable minimum-knowledge proof-system for L can be used as a tool in building a cryptographic system. After an execution of our protocol, the string x can serve as a cryptographically secure encoding—shared only by the prover and the verifier—of the membership-bit ($x \in L$). The use of x as an encoding of the membership-

bit exemplifies what we may call "minimum-knowledge cryptography": it is a probabilistic encryption with the property that neither its specification (i.e., the interactive proof of the value encoded by x) nor its further use in communication can release any compromising knowledge, either to the verifier or to an eavesdropper. The minimum-knowledge property ensures that each party receives exactly the knowledge he is supposed to receive and nothing more. A cryptosystem based on such a minimum-knowledge protocol has the strongest security against passive attack that we could hope to prove; in particular, it is secure against both chosen-message and chosen-ciphertext attack.

The predicate that our protocol tests is that of being a quadratic residue or nonresidue modulo N for a certain number N (whose factorization may be the prover's secret information). We note that the language for which we show membership and nonmembership is in NP $\cap$ co-NP. A conventional membership proof for these languages releases the factorization of N, while in the interactive proof-system presented below no extra knowledge (about the factorization or about anything else) is given either to the verifier or to an eavesdropper.

An important motivation in our work on this protocol comes from our desire to guarantee the security of cryptographic keys, especially in situations where the generation of new keys is very costly or is otherwise limited by the context. If the integer N is the prover's public key in a public-key cryptosystem, then N is not compromised by polynomially many executions of our protocol; a polynomially bounded opponent knows no more after witnessing or participating in these executions than he knew before the key was used at all.

2. Preliminaries.

2.1. Interactive Turing machines. We specify the model for which we describe our protocol; this is an extension of the model used in [17]. Two probabilistic Turing machines A and B form an *interactive pair of Turing machines* if they share a read-only *input tape* and a pair of *communication tapes*; one of the communication tapes is exclusive-write for A, while the other is exclusive-write for B. (The writing heads are unidirectional; once a symbol has been written on a communication tape, it cannot be erased.) We model each machine's probabilistic nature by providing it with a read-only *random tape* with a unidirectional read-head; the machine "flips a coin" by reading the next bit from its random tape. The two machines take turns being *active*. While it is active, a machine can read the communication tapes, perform computation using its own work tape and consulting its random tape, and send a message to the other machine by writing the message on its exclusive-write communication tape. In addition, B has a private *output tape*; whatever is written on this tape when A and B halt is the *result* of their computation.

In order to model the fact that the system is not memory-less, we also assume that each machine has a *history tape*, with a unidirectional write-head, on which the following records are automatically written:

- When the machine flips a coin, the bit it reads from its random tape is recorded on its history tape.
- At the beginning of each active turn, when the machine reads a new message from the other machine's exclusive-write communication tape, it records this message on its history tape.
- At the end of each active turn, when the machine writes a message to the other machine on its own exclusive-write communication tape, it records this message on its history tape.

• The result written on B's output tape is also recorded on B's history tape.

These records are written on the history tape sequentially in order according to the machine's computation; for example, when the machine flips a coin several times while computing its next message, these random bits are recorded on the history tape immediately before the message. The input tape and communication tapes are public, or shared by the two machines; each machine's random tape, history tape, and work tape are private, as is B's output tape. This is not the only way to model the situation we would like to describe, and some of the records written on the history tape are redundant, but without loss of generality we may assume this mode of operation.

When A and B begin their computation, an infinite bit-string is written on each of their random tapes. The choice of these two bit-strings, independently and uniformly at random from the set of all infinite strings, defines a probability measure on the set of possible computation histories of (A, B) that begin in any particular configuration.

For any strings x, h we say that the interactive pair of Turing machines (A, B) begins its computation with input x and B's *initial history* h if in their initial configuration x is written on the common input tape and h is the written portion of B's history tape. (Throughout this paper, we are not concerned with the contents of A's history tape.) We use $(A, B)[x, h]$ to denote the set of computations that begin in this configuration. In each of the protocols that we present in this paper, B never consults its history tape. However, in discussing the properties of these protocols, we must be concerned with an arbitrary Turing machine that may take the role of B in an interaction with A, and that may make use of its history tape.

In what follows, B is limited to expected running time that is polynomial in the length of the common input x, while we make no limiting assumption about A's computational resources. (For cryptographic applications, A is also limited to feasible computation but possesses some trapdoor information.) Their messages to each other are in cleartext, though these messages may depend on their private coin flips, which remain hidden. We assume that both the length of B's initial history, as well as the total length of the messages written on the two communication tapes, are polynomial in $|x|$. For any input string x, we introduce the notation $H_x = \{h \mid h \in \{0, 1\}^*, |h| = O(|x|^{O(1)})\}$ for the set of associated initial histories that we allow.

Our scenario also includes a third probabilistic Turing machine, C, limited to expected polynomial-time computation, that can read the input and communication tapes of A and B and knows their algorithms. A is the *prover*, B is the *verifier*, and C is the *eavesdropper*.

2.2. Ensembles of strings. In order to speak precisely of the knowledge transmitted by communicated messages, we need the following definitions [17], [27], [6]. Let $I \subseteq \{0, 1\}^*$ be an infinite set of strings, and for each $x \in I$, let $\pi[x]$ be a probability distribution on a set of bit-strings. We call $\Pi = \{\pi[x] \mid x \in I\}$ an *ensemble* of strings (usually suppressing any mention of I).

For example, if M is a probabilistic Turing machine, then any input string x defines a probability distribution, according to the coin-tosses (i.e., the random tape) of M's computation, on the set $M[x]$ of possible outputs of M on input x. Thus, for any I, $\{M[x] \mid x \in I\}$ is an ensemble.

As a second example, suppose that (A, B) is an interactive pair of Turing machines. For any strings x, h, let $VIEW_B\{(A, B)[x, h]\}$ denote the set of private "histories" that may be written on B's history tape during a computation that begins with input x and B's initial history h; each of these is B's private view of the protocol execution. This

set has a natural probability distribution according to the random tapes of A and B. Thus, for any set I, $\{VIEW_B\{(A, B)[x, h]\} \mid x \in I, h \in H_x\}$ is an ensemble of strings.

As another example, for any string x let $COM\{(A, B)[x]\}$ denote the set of possible ordered sequences of messages written on the communication tapes of A and B during a computation that begins with input x. Each of these is the public view, and in particular that of the eavesdropper C, of a protocol execution of A and B. This set also has a natural probability distribution. (We assume that the specified computations do not make use of previous private or public history.) Thus, for any set I, $\{COM\{(A, B)[x]\} \mid x \in I\}$ is an ensemble of strings that we call the *communications ensemble* produced by the interactive system (A, B).

A *distinguisher* is a family $D = \{D_x \mid x \in I\}$ of circuits with a single Boolean output; we assume that there is a constant c so that circuit D_x has $|x|^c$ input gates and one output gate. D is *polynomial-size* if there is a constant d so that D_x has at most $|x|^d$ nodes. Suppose that $\Pi = \{\pi[x] \mid x \in I\}$ and $\Pi' = \{\pi'[x] \mid x \in I\}$ are ensembles of strings, and that D is a distinguisher (all with respect to the same constant c). Let $p_D(\pi[x])$ be the probability that D_x outputs a 1 when it is given as input a single sample string of length $|x|^c$, randomly selected according to probability distribution $\pi(x)$; and let $p_D(\pi'[x])$, depending on the distribution $\pi'[x]$, be defined similarly. We call the two ensembles (computationally) *indistinguishable* if for any polynomial-size distinguisher D, for all n and sufficiently long x,

$$|p_D(\pi[x]) - p_D(\pi'[x])| < |x|^{-n}.$$

This condition holds, of course, if the two ensembles are exactly identical. In this case, for *any* distinguisher D the difference $|p_D(\pi[x]) - p_D(\pi'[x])|$ is equal to zero.

Let π and π' be two probability distributions on strings, and suppose that the number δ satisfies $0 \leqq \delta \leqq 1$. We say that π *approximates* π' *with error probability* δ if

$$\sum_s |\text{prob}(\pi[x] = s) - \text{prob}(\pi'[x] = s)| \leqq \delta$$

(where the sum is taken over all strings s in $\{0, 1\}^*$). This implies that the difference $|p_D(\pi[x]) - p_D(\pi'[x])| \leqq \delta$ for any distinguisher D, even if the definition of "distinguisher" is relaxed to allow as inputs to D_x a set of many samples randomly chosen either according to $\pi[x]$ or according to $\pi'[x]$.

2.3. Interactive proof-systems and transfer protocols. This paper is mainly devoted to a special sort of two-party protocol, that of interactively proving or disproving membership in a language L. A protocol that achieves this is called an *interactive proof-system* for L [17]. The prover A and the verifier B share a common input x, the string whose membership is in question. We assume that x belongs to a fixed set I, $I \supseteq L$, of input strings for (A, B). Depending on $k = |x|$, the length of the (binary) representation of the input string, we allow an error probability $\delta(k)$ that vanishes with increasing k. (In fact, all of the examples in this paper satisfy the stronger requirement of an error probability that is exponentially vanishing in k.)

Extending the definition of [17], we distinguish between a *confirming* proof-system for L, whose purpose is that the verifier confirm membership in L for the input string, and a *deciding* proof-system for L, whose purpose is that the verifier decide whether or not the input string is in L. At the end of a confirming protocol, the verifier may either *accept* the proof that $x \in L$, or *reject* the proof; at the end of a deciding protocol, the verifier may either *accept* a proof that $x \in L$, or *accept* a proof that $x \notin L$, or *reject* the proof. The execution ends normally when all of B's messages appear as if B is

following the protocol; if this is so, then A ends the execution in a *success* state. A may halt the execution of the protocol if it detects that B is not following the protocol, ending the execution in a *failure* state.

For any input string x, let $k = |x|$. We say that $(A, B)_k$ is a *confirming interactive proof-system* for L with inputs I and error probability $\delta(k)$ if:

(1) For any $x \in L$ given as input to (A, B), B accepts the proof with probability at least $1 - \delta(k)$.

(2) For any interactive Turing machine A*, and for any $x \in I - L$ given as input to (A*, B), B accepts the proof with probability at most $\delta(k)$.

We say that (A, B) is a *deciding interactive proof-system* for L with inputs I and error probability $\delta(k)$ if:

(1) For any $x \in I$ given as input to (A, B), B accepts the proof, halting with the correct value of the predicate $(x \in L)$ on its output tape, with probability at least $1 - \delta(k)$.

(2) For any interactive Turing machine A*, and for any $x \in I$ given as input to (A*, B), B accepts a proof of the incorrect value of the predicate $(x \in L)$ with probability at most $\delta(k)$.

As part of the definition, we require that these conditions should hold independently of the choice of the initial-history string (of length polynomial in k) that may be written on B's history tape at the beginning of the computation.

In the first definition, we require that (with high probability) B correctly accept the proof for strings $x \in L$, and that no cheating adversary, no matter how powerful, can convince B incorrectly to accept the proof for strings $x \notin L$ (except with vanishingly small probability). In the second definition, we require that (with high probability), given *any* input string $x \in I$, B correctly decide whether $x \in L$ or $x \notin L$, and that no adversary can convince B to accept an incorrect proof (except with vanishingly small probability). The probability is taken over all sequences of coin-tosses (i.e., over all possible random-tape bit-strings) used by the probabilistic computations of the two Turing machines.

The two definitions above describe correctness for protocols that transfer to B the computed value of a Boolean predicate that supplies one bit of "knowledge" about the input string. We can also study a more general sort of *transfer protocol* whose purpose is to transfer the result $F(x)$ of any specified computation depending on the input string x. For example, a deciding interactive proof-system for the language L is a transfer protocol for the function $F(x)$ taking the value 1 or 0 according to whether or not $x \in L$. Because the interacting machines are probabilistic, the intended result may take values in a probability distribution whose value $F(x, r)$ depends on x as well as on a random input string r. As in the case of an interactive proof-system, B may either *accept* or *reject* an execution of an interaction with another Turing machine. We say that a given protocol (A, B) is *correct* for a specified probability distribution of outputs if B's computed result, when it interacts with A, has the intended distribution (with very high probability), and no machine A*, no matter how powerful, can bias the distribution of B's outputs (except with vanishingly small probability).

In order to define "correctness" more precisely, we observe that the computations of any interactive pair of Turing machines (A, B) determine a partial function $f_{A,B}$ as follows. Given strings x, r_A, and r_B, we define $f_{A,B}(x, r_A, r_B)$ to be the result written on B's output tape at the end of an accepting computation of (A, B) that begins with input x, when their random-tape strings begin with r_A and r_B (respectively); this value is well-defined, as long as r_A and r_B are sufficiently long. Notice that the choice of r_A and r_B defines a probability distribution $f_{A,B}(x, \cdot, \cdot)$.

We say that (A, B) is a *correct transfer protocol* for the probability distribution $F(x, r)$, with inputs I and error probability $\delta(\cdot)$, if:

(1) For each $x \in I$, the distribution $f_{A,B}(x, \cdot, \cdot)$ of B's computed outputs approximates, with error probability $\delta(|x|)$, the distribution $F(x, \cdot)$ of intended results.

(2) Let A* be any interactive Turing machine. We require that for any $x \in I$ and for any $s \in \{0, 1\}^*$, the probability that B accepts the computation of (A*, B) on input x and writes out the string s as its output is bounded by the quantity prob $(F(x, \cdot) = s) + \delta(|x|)$.

Note that, according to the second part of this definition, it may be possible for a malicious adversary A* to bias the distribution of the set of conversations of (i.e., the set of sequences of messages exchanged by) A* and B on a particular input string x. But A* cannot significantly increase the probability that any given result string is accepted by B; in particular, A* cannot force B to accept an erroneous result (one that occurs with probability zero in the distribution $F(x, \cdot)$) except with probability $\delta(|x|)$.

Observe that the probability threshold δ occurs twice in the above definition. In general, there may be protocols for which it makes sense to define correctness with two different δ's. In all our examples, the function $\delta(k)$ is exponentially vanishing in k; therefore, for simplicity, we use the same δ in both places.

3. Knowledge. In the setting of complexity theory, what do we mean by "knowledge"? Informally, a message conveys knowledge if it communicates the result of an intractable computation. A message that consists of the result of a computation that we can easily carry out by ourselves does not convey knowledge. In particular, a string of random bits—or a string of bits that is "indistinguishable" from a random string (as defined above)—does not convey knowledge, since we can flip coins by ourselves.

3.1. Minimum knowledge. Suppose that (A, B) is a confirming interactive proof-system for a language L, taking inputs from the set I. Following the definition in [17], we say that the system (A, B) is *minimum-knowledge* if, given any expected polynomial-time probabilistic Turing machine B*, there exists another probabilistic Turing machine M_{B^*}, running in expected polynomial time, such that the ensembles $\{M_{B^*}[x, h] \mid x \in L, h \in H_x\}$ and $\{VIEW_{B^*}\{(A, B^*)[x, h]\} \mid x \in L, h \in H_x\}$ are (computationally) indistinguishable. If the ensembles are identical, we say that the proof-system is *perfectly minimum-knowledge.*

The output of M_{B^*}, on input $x \in L$ and initial history h, is a simulation of B*'s view of the computation that A and B* would have on the same input and the same initial history. Note that, in this definition, we are not concerned with inputs that do not belong to L. When it takes part in a successful execution of the protocol with input x, B* learns that (with high probability) the predicate of language-membership associated with the protocol, $x \in L$, is true; however, it gains no more knowledge that this. Note that in our examples, B (the machine that acts according to the protocol specifications) does not use its initial history string at all; however, when we worry about the "knowledge" that a cheating machine B* may try to extract from A we have to consider the fact that B* can use its history string.

The authors of [17] called a confirming proof-system satisfying the above properties "zero-knowledge." We now show how to extend this definition so as to be able to say when a more general sort of protocol—for example, a two-party protocol whose purpose is to transfer to one of the parties the result of a hard computation—should be called "minimum-knowledge."

Let (A, B) be an interactive pair of Turing machines which constitute a correct transfer protocol for the probability distribution $F(x, r)$, with inputs I and error probability δ. We say that (A, B) is *minimum-knowledge* if, given any expected polynomial-time probabilistic Turing machine B*, there exists another probabilistic Turing machine M_{B^*}, running in expected polynomial time, such that:

(1) M_{B^*} has one-time access to an F-oracle, as follows. Given any input x and initial history h, M_{B^*} queries the oracle with input x; the oracle returns a value distributed according to $F(x, \cdot)$.

(2) The ensembles $\{M_{B^*}[x, h] | x \in I, h \in H_x\}$ and $\{VIEW_{B^*}\{(A, B^*)[x, h]\} | x \in I, h \in H_x\}$ are indistinguishable.

If the ensembles are identical, we say that the proof-system is *perfectly minimum-knowledge.*

In order to motivate this definition, we recall that we are trying to formalize the notion of the amount of knowledge transmitted by a sequence of messages. Speaking informally, one gains no knowledge from a message which is the result of a feasible computation that one could just as well have carried out by oneself. If the purpose of a protocol followed by two interacting parties A and B is that A transmit to B a value v chosen according to the probability distribution $F(x, r)$, we would like to be able to say exactly when the protocol transmits no more knowledge than this value. We might also demand that the protocol accomplish this even if B somehow tries to cheat—that is, even if the Turing machine B is replaced by another (polynomial-time, but possibly "cheating") machine B*. The simple transmission of the value v can be modelled by a single oracle query (with input x). If the provision of this oracle query makes it possible, by means of a feasible computation, to simulate B*'s view of the "conversation" that A and B* would have had on input x, then we can say that when A and B* actually have a conversation (i.e., follow the protocol) with the same input, there is no *additional* knowledge transmitted to B* besides the value v.

Note that if F is computable in expected polynomial time, then the F-oracle adds no power to the machine M_{B^*}. In this case M_{B^*} can compute F without the assistance of A.

In all our examples, the simulating machine M_{B^*} uses the program of B* as a subprogram or subroutine. This subprogram makes use of the simulator's input tape (containing the input string x), a virtual history tape (which is initialized to contain the given initial history h), a virtual random tape, a virtual work tape, two virtual communication tapes, and a virtual output tape. Without loss of generality we supply the probabilistic machine M_{B^*} with two random tapes; one of these is B*'s virtual random tape. On its output tape—which is also the virtual history tape for the subprogram B*—the simulator uses the subprogram to write records that correspond to B*'s view of the simulated protocol execution.

While carrying on its computation, the machine M_{B^*} may back up a few steps in the simulated protocol and restore a previous machine configuration: It recovers the old state of B* and the old content of the virtual work tape, and resets both the virtual read-head of B*'s random tape and the write-head of its own output tape (the virtual history tape) to where they had been earlier; then it proceeds with its simulation, starting again from the old configuration but "flipping new coins" in its probabilistic computation.

The virtual communication tapes are used to simulate the communication activities of the simulated protocol. The simulator "sends" a message to B* by writing it on the appropriate virtual communication tape and then activating the subprogram. The subprogram operates for (the simulation of) one active turn and then writes a message

on the other virtual communication tape; this is the next message "received" from B*. Just as in the interaction of B* with A, the simulator's subprogram B* records random bits, messages read and written, and the computed result on the virtual history tape. The operation of the subprogram B* during a simulated active turn, beginning in a certain state with a certain configuration of the virtual tapes, is identical to the operation of the interactive Turing machine B* during an active turn, beginning in the same state with the same configuration of the actual tapes, of an actual protocol execution with A. This matter of the difference in B*'s operation, either as a subprogram of the simulator or as a Turing machine interacting with A, is discussed further in the remark at the end of the next section.

3.2. Concatenation of protocols. Next, we investigate how protocols may be concatenated in order to achieve modularity in protocol design and how properties of the resulting protocol can be derived from the properties of its subprotocols. The protocol presented in this paper is an example of such a modular design.

We write $s \cdot s'$ for the concatenation of the two strings s and s'.

Suppose that we are given two protocols $P_1 = (A_1, B_1)$ and $P_2 = (A_2, B_2)$. We define the *concatenation* of the two protocols, denoted $P = P_1; P_2$, to be the following two-stage protocol: Its first stage is P_1. If at the end of this stage A_1 is not in a failure state and B_1 has not rejected, the protocol continues with P_2; otherwise the protocol halts. We write $A_1; A_2$ and $B_1; B_2$ for the interacting machines of the concatenated protocol. At the end of an execution, the history tape of $B_1; B_2$ contains the initial history-string, followed by B_1's private view of the execution of P_1, followed by B_2's private view of the execution of P_2.

Assume that P_1 and P_2 are two transfer protocols for the probability distributions F_1 and F_2, respectively, both taking inputs from the set I. Then the concatenated protocol, on input $x \in I$, transfers to $B_1; B_2$ the combined result $[F_1(x, \cdot), F_2(x, \cdot)]$. As a special case, suppose that P_1 is a confirming interactive proof-system for L_1 with inputs I, and that P_2 is a confirming interactive proof-system for L_2 with inputs L_1. Then the concatenated protocol is a confirming interactive proof-system for $L_1 \cap L_2$, with inputs I.

It may not be surprising that the concatenation of two correct protocols gives the correct combined result. The more important observation is that, as we prove below, the concatenated protocol is minimum-knowledge if P_1 and P_2 are both minimum-knowledge.

LEMMA. *Given two protocols P_1 and P_2 as above, with error probabilities $\delta_1(k)$ and $\delta_2(k)$, respectively. Then the concatenation $P = P_1; P_2$ is a correct transfer protocol for the combined result $[F_1(x, \cdot), F_2(x, \cdot)]$ with error probability $\delta(k) = \delta_1(k) + \delta_2(k) + \delta_1(k) \cdot \delta_2(k)$. Furthermore, if P_1 and P_2 are both minimum-knowledge (or, respectively, both perfectly minimum-knowledge), then so is their concatenation.*

Proof. First we show that correctness of protocols is preserved by concatenation. It is clear that if the output distribution of (A_1, B_1) approximates the intended distribution F_1 with error probability δ_1, and the output of (A_2, B_2) approximates the intended distribution F_2 with error probability δ_2, then (A, B) approximates $[F_1, F_2]$ with error probability at most $\delta_1 + \delta_2$.

To show that the second correctness condition holds, let $x \in I$ and let s_1, s_2 be a pair of possible output strings, occurring with probabilities p_1 and p_2 in probability distributions $F_1(x, \cdot)$ and $F_2(x, \cdot)$, respectively. The pair (s_1, s_2) occurs with probability $p_1 p_2$ in the combined result distribution $[F_1(x, \cdot), F_2(x, \cdot)]$. Let us write $\delta_1 = \delta_1(|x|)$ and $\delta_2 = \delta_2(|x|)$. By the correctness of protocols P_1 and P_2, if A* is any interactive

Turing machine that interacts with $B = B_1; B_2$, then the probability that B writes out (s_1, s_2) is at most

$$(p_1 + \delta_1)(p_2 + \delta_2) = p_1 p_2 + \delta_1 p_2 + \delta_2 p_1 + \delta_1 \delta_2 \leqq p_1 p_2 + (\delta_1 + \delta_2 + \delta_1 \delta_2),$$

as required.

Next we show that concatenation maintains the minimum-knowledge property. Assume that P_1 and P_2 are both minimum-knowledge, and let B^* be any probabilistic interactive Turing machine, running in expected polynomial time, that interacts with $A_1; A_2$. We may write $B^* = B_1^*; B_2^*$ to denote the two parts of B^*'s program. For convenience, let us write $V_1[x, h] = VIEW_{B_1^*}\{(A_1, B_1^*[x, h]\}$ and $V_2[x, h] = VIEW_{B_2^*}\{(A_2, B_2^*)[x, h]\}$. Thus, for any input string x and any initial history h, we have $VIEW_{B^*}\{(A, B^*)[x, h]\} = \{v_1 \cdot v_2 \mid v_1 \in V_1[x, h],\ v_2 \in V_2[x, h \cdot v_1]\}$.

To show that the concatenated protocol P is minimum-knowledge we have to show the existence of a simulating expected polynomial-time probabilistic Turing machine $M = M_{B^*}$ whose output ensemble $\{M[x, h] \mid x \in I,\ h \in H_x\}$ is indistinguishable from the ensemble $\{VIEW_{B^*}\{(A, B^*)[x, h]\} \mid x \in I,\ h \in H_x\}$.

Our hypothesis on P_1 implies that, given B_1^*, there is a simulating machine M_1, running in expected polynomial time, with access to an F_1-oracle, so that the ensembles $\{M_1[x, h] \mid x \in I,\ h \in H_x\}$ and $\{V_1[x, h] \mid x \in I,\ h \in H_x\}$ are indistinguishable. Similarly, our hypothesis on P_2 implies that, given B_2^*, there is a simulating machine M_2, running in expected polynomial time, with access to an F_2-oracle, so that the ensembles $\{M_2[x, h] \mid x \in I,\ h \in H_x\}$ and $\{V_2[x, h] \mid x \in I,\ h \in H_x\}$ are indistinguishable. We specify M to be the machine that operates as follows, given any input string $x \in I$ and initial history $h \in H_x$. First, M runs machine M_1 on (x, h) to produce an output h_1. Second, if h_1 is the simulation of a successful execution of P_1, then M runs M_2 on (x, h_1) to produce its final output; otherwise, M simply writes out h_1.

For any $x \in I$, $h \in H_x$ we define the sets of strings

$$E_1[x, h] = VIEW_{B^*}\{(A, B^*)[x, h]\} = \{v_1 \cdot v_2 \mid v_1 \in V_1[x, h],\ v_2 \in V_2[x, h \cdot v_1]\}, \text{ and}$$

$$E_2[x, h] = M[x, h] = \{m_1 \cdot m_2 \mid m_1 \in M_1[x, h],\ m_2 \in M_2[x, h \cdot m_1]\}.$$

(As usual, the choices of the bit-strings that are written on these probabilistic machines' random tapes define a probability distribution on both of these sets.) We need to show that the ensembles $\bar{E}_1 = \{E_1[x, h] \mid x \in I,\ h \in H_x\}$ and $\bar{E}_2 = \{E_2[x, h] \mid x \in I,\ h \in H_x\}$ are indistinguishable. For this purpose, we introduce the intermediate ensemble $\bar{E}_3 = \{E_3[x, h] \mid x \in I,\ h \in H_x\}$, where

$$E_3[x, h] = \{v_1 \cdot m_2 \mid v_1 \in V_1[x, h],\ m_2 \in M_2[x, h \cdot v_1]\}.$$

Assume that $\bar{E}_1$ and $\bar{E}_2$ are not computationally indistinguishable. Then there is a polynomial-size distinguisher $D = \{D_{x,h} \mid x \in I,\ h \in H_x\}$ that distinguishes between the two ensembles. In other words, in the notation of § 2.2, for some n and for infinitely many pairs (x, h) (with $x \in I$, $h \in H_x$)

$$|p_D(E_1[x, h]) - p_D(E_2[x, h])| \geqq |x|^{-n}.$$

This implies, by the triangle equality, that at least one of the inequalities

$$|p_D(E_2[x, h]) - p_D(E_3[x, h])| \geqq \tfrac{1}{2}|x|^{-n} \tag{1}$$

$$|p_D(E_1[x, h]) - p_D(E_3[x, h])| \geqq \tfrac{1}{2}|x|^{-n} \tag{2}$$

holds infinitely often, i.e., that the circuit-family D distinguishes either between $\bar{E}_2$ and $\bar{E}_3$ or between $\bar{E}_1$ and $\bar{E}_3$ (or both.) We next show that either of these possibilities leads to a contradiction.

First, we show that if D distinguishes between $\bar{E}_2$ and $\bar{E}_3$ then we can construct a distinguisher D_1 that distinguishes between the ensembles $\{M_1[x, h] | x \in I, h \in H_x\}$ and $\{V_1[x, h] | x \in I, h \in H_x\}$, contradicting the hypothesis that P_1 is minimum-knowledge. Let I_1 be the infinite set of pairs (x, h) for which inequality (1) holds. Given as input a string s, chosen either from $M_1[x, h]$ or from $V_1[x, h]$, let $C_{x,h}$ be the (probabilistic) circuit that does the following: It simulates the operation of M_2 on input (x, s) for a suitable multiple of its expected running time to produce either a string m_2 or (for those few sequences of coin-flips which may cause M_2 to run too long) a null output, then passes $h \cdot m_2$ to the circuit $D_{x,h}$, which gives its output. Since the simulation of M_2 is polynomial in length, and D is polynomial-size, the circuit-family C is also polynomial-size. Inequality (1) shows that for all pairs $(x, h) \in I_1$, the circuit $C_{x,h}$ distinguishes between $M_1[x, h]$ and $V_1[x, h]$. Therefore, $C_{x,h}$ can be converted into a deterministic polynomial-size circuit $(D_1)_{x,h}$ that distinguishes between the same two sets.

Second, we show that if D distinguishes between $\bar{E}_1$ and $\bar{E}_3$ then we can construct a distinguisher D_2 that distinguishes between the ensembles $\{M_2[x, h] | x \in I, h \in H_x\}$ and $\{V_2[x, h] | x \in I, h \in H_x\}$, contradicting the hypothesis that P_2 is minimum-knowledge. Let I_2 be the infinite set of pairs (x, h) for which inequality (2) holds, and consider the infinite set $I_2' = \{(x, v) | v \in V_1[x, h], (x, h) \in I_2\}$. We define $(D_2)_{x,h}$ to be the circuit whose output, on input s (chosen either from $M_2[x, h]$ or from $V_2[x, h]$) is the output of $D_{x,h}$ on input $h \cdot s$. Since D is polynomial-size, it is clear that D_2 is polynomial-size, too. Inequality (2) shows that $(D_2)_{x,h}$ distinguishes between $M_2[x, h]$ and $V_2[x, h]$ for infinitely many pairs (x, h)—namely, for all pairs $(x, v) \in I_2'$.

We therefore conclude that the concatenated protocol is minimum-knowledge. Analogous arguments show that the concatenated protocol is perfectly minimum-knowledge if the same is true of both component protocols. □

REMARK. We mention here a special case of the above lemma that we use implicitly throughout the proofs in §§ 5 and 6. Suppose that a protocol (A, B) is given, and consider a certain point in the protocol execution when A has just sent a message and B is about to perform its next active turn. Let P_1 be the protocol up to this point, and let P_2 consist just of B's next active turn. The lemma implies that if P_1 and P_2 are minimum-knowledge, then so is the given protocol through the end of B's next turn. This allows us to specify a machine M_{B^*} for our proofs below, simply by having the machine activate a subprogram B* as explained at the end of the previous section: As long as the subprogram, when activated, has access to a virtual history tape whose contents are indistinguishable from the history tape of an actual protocol execution carried on with A, its operation within M_{B^*} is identical to its operation during an actual interaction.

3.3. Result indistinguishability. Next we introduce the eavesdropper C, as described above. Recall that $COM\{(A, B)[x]\}$ is the set of possibilities for C's view of the computation of A and B on input x. In all our examples of interactive pairs of Turing machines (A, B), neither machine uses its history tape. Thus, without loss of generality we can assume that A and B begin their computation with their history tapes initially empty.

We call an interactive pair of Turing machines (A, B) result-indistinguishable if an eavesdropper that has access to the communications of A and B on input x gains

no knowledge. More precisely, the system (A, B) is *result-indistinguishable* if there exists a probabilistic polynomial-time Turing machine M such that the ensembles $\{M[x] \mid x \in I\}$ and $\{COM\{(A, B)[x]\} \mid x \in I\}$ are indistinguishable. If the ensembles are exactly identical, we say that the proof-system is *perfectly result-indistinguishable.*

Suppose that (A, B) is a transfer protocol for the probability distribution $F(x, r)$. Observe that unlike the simulating machine in the definition of the minimum-knowledge property, this machine M does not have access to an oracle for F. In other words, M can simulate the communications of A and B on input x, regardless of the value $F(x, r)$ (even if computing F is intractable). Since this simulation is by means of a feasible computation that an eavesdropping adversary could carry out for itself, the adversary gains no knowledge if it is given the text of a "conversation" belonging to the set $COM\{(A, B)[x]\}$.

We remark that if two protocols are result-indistinguishable, then so is their concatenation. The simulating machine for the concatenated protocol is simply the concatenation of the two simulators for the component protocols; neither the interacting parties nor the simulator makes any computation that depends on the history tapes.

4. Specification of the language

4.1. Preliminaries. We assume that the reader is familiar with the following notions from elementary number theory. (See, for example, [19], [23] for the number theory, and [21] for a computational point of view.) We will be working in the multiplicative group $\mathbf{Z}_N^*$ of integers relatively prime to N. Any element $z \in \mathbf{Z}_N^*$ is called a *quadratic residue* if it is a square mod N (i.e., if the equation $x^2 \equiv z \bmod N$ has a solution); otherwise, z is a *quadratic nonresidue* mod N. Given N and $z \in \mathbf{Z}_N^*$, the quantity called the *Jacobi symbol* of z with respect to N, denoted $(\frac{z}{N})$, can be efficiently computed (in time polynomial in $\log N$) and takes on the values $+1$ and -1. If $(\frac{z}{N}) = -1$, then z must be a quadratic nonresidue mod N. On the other hand, if $(\frac{z}{N}) = +1$, then z may be either a residue or a nonresidue. Determining which is the case, without knowing the factorization of N, appears to be an intractable problem, namely the *quadratic residuosity* problem. (However, given the prime factorization of N, it is easy to determine whether or not z is a quadratic residue.) Several cryptographic schemes have been proposed that base their security on the assumed difficulty of distinguishing between residues and nonresidues modulo an integer N that is hard to factor [16], [3], [22].

We also make use of Bernstein's law of large numbers [25], [21]: Suppose that the event E occurs with probability p, and let $F_k(E)$ denote the frequency with which E occurs in k independent trials. Then for any $k \geqq 1$ and any positive $\varepsilon \leqq p(1-p)$,

$$\text{Prob}\{|F_k(E) - p| \geqq \varepsilon\} \leqq 2\, e^{-k\varepsilon^2}.$$

4.2. The language. The protocol introduced in [17] is a minimum-knowledge confirming interactive proof-system for the language

$$\{(N, z) \mid z \in \mathbf{Z}_N^*,\ z \text{ a quadratic nonresidue mod } N\}.$$

The protocol that we present below is a deciding interactive proof-system, which is both minimum-knowledge and result-indistinguishable, for a language based on the same problem.

We use the notation $\nu(N)$ to represent the number of distinct prime factors of an integer N.

Our protocol is concerned with integers of a special form, namely those with prime factorization $N = \prod_{i=1}^{k} p_i^{e_i}$ such that for some i, $p_i^{e_i} \equiv 3 \bmod 4$. Let ***BL*** (for Blum, who pointed out their usefulness in cryptographic protocols) denote the set of such integers. There are two equivalent formulations of membership in ***BL***: (1) $N \in \boldsymbol{BL}$ if

and only if for any quadratic residue mod N, half its square roots (mod N) have Jacobi symbol $+1$ and half its square roots have Jacobi symbol -1. (2) $N \in \boldsymbol{BL}$ if and only if there exists a quadratic residue mod N which has two square roots with different Jacobi symbols [2].

The special integers that we require form a subset of $\boldsymbol{BL}$, namely

$$\boldsymbol{N} = \{N \mid N \in \boldsymbol{BL},\ N \equiv 1 \bmod 4,\ \nu(N) = 2\}.$$

It is not hard to see that this set may be defined equivalently as

$$\boldsymbol{N} = \{p^i q^j \mid p \neq q \text{ prime},\ i, j \geqq 1,\ p^i \equiv q^j \equiv 3 \bmod 4\}.$$

Finally, we define the languages

$$I = \{(N, z) \mid N \in \boldsymbol{N},\ z \in \mathbf{Z}_N^*,\ (\tfrac{z}{N}) = +1\} \quad \text{and}$$

$$L = \{N, z) \in I \mid z \text{ a quadratic residue mod } N\}.$$

Taking I as the set of inputs, this paper gives a deciding interactive proof-system for L. Notice that $I - L = \{(N, z) \in I \mid z \text{ a quadratic nonresidue mod } N\}$.

4.3. Outline of the protocol. Our protocol is the concatenation of two subprotocols. The first part is a confirming interactive proof-system for I. If the first part is completed successfully (i.e., if A proves to B that the input string is in I), then A and B perform the second part of the protocol. The second part, taking inputs from the set I, is a deciding interactive proof-system for the language L; A proves to B either that the input string is in L or that it is not in L. Both parts are minimum-knowledge, and the second part is result-indistinguishable as well. The eavesdropper learns that, with high probability, the input is in I. But he gains no more knowledge than this—in particular, he gains no computational advantage in deciding whether the input is in L or in $I - L$, i.e., whether or not z is a quadratic residue mod N.

The confirmation that an input string (N, z) belongs to I in turn requires three stages, each of which confirms a property of N or of z; these stages are carried out in the following order:

(1) $N \equiv 1 \bmod 4$, $\nu(N) > 1$, $z \in \mathbf{Z}_N^*$, and $(\tfrac{z}{N}) = +1$.

(2) $N \in \boldsymbol{BL}$.

(3) $\nu(N) \leqq 2$.

While proving that our protocol has the properties that we desire, we make no limiting assumption about the computational power of Turing machine A. However, we remark that the protocol can be performed by a probabilistic polynomial-time Turing machine A which is given the factorization of the relevant integers N. (In the cryptographic applications that we discuss later, the party that performs A's role in our protocol chooses N along with its prime factorization.)

We now give the details of our protocol: the confirming first part in § 5, and the deciding second part in § 6.

5. Interactive confirmation of the input language. In each of the protocols that we describe, we use the notation "A → B: · · · " to indicate the transmission of a message from A to B.

5.1. Blum's coin-flip protocol. Our confirmation protocol requires that A and B jointly generate a sequence of bits. The verifier B has to be sure that A cannot bias these bits. They do this by following a protocol due to Blum [2].

An integer $N \in \boldsymbol{BL}$, $N \equiv 1 \bmod 4$, is given.

A and B generate a random bit b:

1. B chooses $u \in \mathbf{Z}_N^*$ at random, and computes $v := u^2 \bmod N$;
   B → A: v

2. A chooses $\sigma := +1$ or -1 at random, its guess for $(\frac{u}{N})$;
   $A \to B$: σ
3. $B \to A$: u
4. if $v \neq u^2 \bmod N$, then A halts the protocol in the FAILURE state;
   otherwise, if $\sigma = (\frac{u}{N})$ then $b := 1$ else $b := 0$

The message triple (v, σ, u) may be regarded as an encoding of the bit $b = \frac{1}{2} + \frac{1}{2}(\frac{u}{N})\sigma$.

This protocol is correct: Since B picks u at random and A picks the sign σ at random, the bit b chosen by the protocol is random. Furthermore, the first alternate characterization of ***BL*** (§ 4.2) implies that no interactive Turing machine A*, no matter what its computational power, can bias the bit produced, since it cannot guess the Jacobi symbol of the square root of v chosen by B with probability greater than $\frac{1}{2}$.

We remark that a cheating Turing machine B* could bias the bit solely by using its ability to produce two numbers u and u', both square roots (mod N) of v, with opposite Jacobi symbols; this capacity would enable B* to factor N simply by computing the greatest common divisor $(u - u', N)$.

The protocol is perfectly minimum-knowledge. The reason is that A's only task is to transmit a guess, $\sigma = +1$ or -1, for a sign, a task that may easily be carried out by a simulator interacting with B*. We formalize this argument below.

5.2. The confirmation protocol. This is a minimum-knowledge confirming interactive proof-system by which A proves to B that the input (N, z) is in the language I defined above. It consists of the concatenation of three subprotocols, each of which takes, as legal input, a string that has been confirmed (with high probability) by the preceding subprotocol. Let k denote the length of the input string.

First Stage: The easy properties of N and z

This stage involves no communications between A and B. Given (N, z) as input, B checks that $N \equiv 1 \bmod 4$, that N is not a prime power, and that $(\frac{z}{N}) = +1$. Each of these is easily accomplished in time polynomial in $\log N$ [21]. If any one of these conditions does not hold, then B REJECTS the proof (and halts the entire protocol).

Second Stage: N belongs to ***BL***

The following protocol is due to Blum [2]. The error probability of this proof-system is $\delta_2(k) = 2^{-k}$. This stage does not concern itself with z at all. The integer N must satisfy $N \equiv 1 \bmod 4$; this condition holds if the first stage has been completed successfully.

1. repeat k times:
   1.1 A chooses a quadratic residue $r \in \mathbf{Z}_N^*$ at random;
   $A \to B$: r
   1.2 B chooses $\sigma := +1$ or -1 at random;
   $B \to A$: σ
   1.3 if $\sigma \notin \{-1, +1\}$, then A halts the protocol in the FAILURE state;
   otherwise, A computes s such that $s^2 \equiv r \bmod N$ and $(\frac{s}{N}) = \sigma$;
   $A \to B$: s
   1.4 B checks to make sure that s satisfies the above conditions; if not, then B REJECTS the proof (and halts the entire protocol).
2. B ACCEPTS the proof.

Third Stage: N has two prime factors

This stage also does not concern itself with z.

Let us use $\mathbf{Z}_N^*(\pm 1)$ to denote the set of elements of $\mathbf{Z}_N^*$ with Jacobi symbol ± 1 (respectively). This protocol relies on the fact that if N has exactly i prime factors (i.e., $\nu(N) = i$), then exactly $1/2^{i-1}$ of the elements of $\mathbf{Z}_N^*(+1)$ are quadratic residues. A and B jointly pick random elements of $\mathbf{Z}_N^*(+1)$. If A can show that about half of them are residues (e.g., by producing their square roots mod N), then B should be convinced that $\nu(N) \leqq 2$. Since N is not a prime power, $\nu(N)$ must be equal to 2.

In order to pick a list of random elements of $\mathbf{Z}_N^*(+1)$, A and B follow Blum's coin-flip protocol, which requires that $N \in \boldsymbol{BL}$ and $N \equiv 1 \bmod 4$. These conditions hold (with very high probability) if the second stage has been completed successfully.

This proof-system has error probability $\delta_3(k) = 2e^{-k(1/8)^2}$.

1. A and B use Blum's coin-flip protocol to generate k random elements $r_1, \cdots, r_k \in \mathbf{Z}_N^*(+1)$:
   $i := 0$;
   do until $i = k$:
   a. generating it bit by bit using Blum's coin-flip protocol, A and B choose a number a, $0 < a < N$
   b. if g.c.d. $(a, N) \neq 1$ (which happens with vanishingly small probability) then HALT the protocol
   c. if $(\frac{a}{N}) = +1$ then $i := i + 1$; $r_i := a$
2. for each $i = 1, \cdots, k$ such that r_i is a quadratic residue, A computes s_i such that $r_i \equiv s_i^2 \bmod N$;
   A $\rightarrow$ B: (i, s_i)
3. B checks that at least $\frac{3}{8}$ of the r_i are quadratic residues; if so then B ACCEPTS the proof (and otherwise B REJECTS the proof and halts the entire protocol).

THEOREM 1. *This protocol is a perfectly minimum-knowledge confirming interactive proof-system for the language* $I = \{(N, z) \mid N \equiv 1 \bmod 4,\ N \in \boldsymbol{BL},\ \nu(N) = 2,\ (\frac{z}{N}) = +1\}$.

Proof. We treat each of the three subprotocol stages separately. As a consequence of the lemma of § 3.2, it then follows immediately that the concatenation of the three has the required properties.

First Stage

The first stage is, trivially, a confirming proof-system for the language

$$I_1 = \{(N, z) \mid N \equiv 1 \bmod 4,\ \nu(N) > 1,\ z \in \mathbf{Z}_N^*,\ (\tfrac{z}{N}) = +1\},$$

since each of these conditions is validated by B in polynomial time without interacting with A at all.

Second Stage

Given an integer $N \equiv 1 \bmod 4$ (in particular, given input that has been confirmed in the first stage), the second stage is a perfectly minimum-knowledge confirming interactive proof-system for the language $I_2 = \{(N, z) \mid N \in \boldsymbol{BL}\}$ with error probability $\delta_2(k) = 2^{-k}$.

This stage requires $O(k)$ communication rounds, during which $O(k^2)$ bits are exchanged.

The correctness of this stage depends on the alternate characterizations of membership in $\boldsymbol{BL}$ (§ 4.2). If $N \in \boldsymbol{BL}$, then each quadratic residue r sent by A has at least one square root mod N with Jacobi symbol $+1$ and at least one square root mod N with Jacobi symbol -1; no matter which sign σ B chooses, A can respond with a square root of the appropriate sign. B accepts the proof with probability 1. On the other hand, if $N \notin \boldsymbol{BL}$ then no quadratic residue mod N has two square roots with Jacobi symbols of opposite sign. In this case, it is very likely that there is some i for which A will be

unable to send an appropriate s_i, and B will halt the protocol. The only way for a cheating A* to convince B that $N \in \boldsymbol{BL}$ (by sending the appropriate elements s_i) is by guessing the entire sign-sequence $\sigma_1, \cdots, \sigma_k$; the probability that such a guess will be correct is at most $2^{-k} = \delta_2(k)$. Thus, this protocol is indeed a confirming interactive proof-system for ***BL***.

To prove the perfect minimum-knowledge property, choose any interactive Turing maching B*; we have to specify the computation of a Turing machine M_{B^*} whose output, on input $N \in \boldsymbol{BL}$ and initial history h, is a simulation of B*'s view of the computation that A and B* would have performed on the same input. This view includes a message-history that consists of triples (r, σ, s) satisfying the conditions implicitly defined by the specification of the protocol. As described above in § 3.1, M_{B^*} uses the program of B* as a subroutine. After initializing B*, M_{B^*} operates as follows:

1. repeat k times:
   1.1 save the current configuration of B*;
   1.2 choose $s \in \mathbf{Z}_N^*$ at random, compute $r := s^2 \bmod N$, "send" r to the simulated B^*, and "receive" σ in return;
   1.3 if $\sigma \notin \{-1, +1\}$ then append HALT to A's message-record in B*'s virtual history, write out the updated virtual history, and halt;
   otherwise if $(\frac{s}{N}) \neq \sigma$ then restore the saved configuration of B* and go back to step 1.1;
   (else $(\frac{s}{N}) = \sigma$ and the most recent exchange of messages recorded in B*'s virtual history is the triple (r, σ, s))
2. write out B*'s virtual history.

For each of the k iterations, the expected number of times the loop has to be repeated is 2, since for any value of r the probability that $(\frac{s}{N}) = \sigma$ is exactly $\frac{1}{2}$; thus the expected running time of M_{B^*} is polynomial in k.

The simulated messages "sent" to B* are drawn from the same probability distribution as the messages sent by A in an actual execution of the protocol, and the random communications triples (r, σ, s) produced by M_{B^*} satisfy the conditions $s^2 \equiv r \bmod N$ and $(\frac{s}{N}) = \sigma$. As explained in § 3.2, these messages are interleaved on the virtual history tape with the random-tape bits used by B*, exactly as they would be in an actual interaction with A. Therefore the sets $M_{B^*}[N, h]$ and $VIEW_{B^*}\{(A, B^*)[N, h]\}$ are identical. This completes the proof for the second stage.

Third Stage

Given an integer from the set

$$\{N \mid N \in \boldsymbol{BL},\ N \equiv 1 \bmod 4,\ \nu(N) > 1\}$$

(in particular, given input that has been confirmed in the first and second stages), the third stage is a perfectly minimum-knowledge confirming interactive proof-system for the language $I_3 = \{(N, z) \mid \nu(N) = 2\}$ with error probability $\delta_3(k) = 2e^{-k(1/8)^2}$.

This stage requires $O(k^2)$ communication rounds, during which $O(k^3)$ bits are exchanged.

During the third stage, A and B together choose random elements of $\mathbf{Z}_N^*(+1)$. Since they do this by means of Blum's coin-flip protocol, and no Turing machine A* can bias the bits produced by Blum's procedure, these elements are indeed produced at random. In order to prove that this stage is a proof-system, consider the experiment of choosing a random element of $\mathbf{Z}_N^*(+1)$, where the experiment is a success if the chosen element is a quadratic residue mod N; let $F_k(N)$ denote the frequency of

successes in k independent trials. Recall that B accepts N if the frequency $F_k(N) \geqq \frac{3}{8}$. As mentioned above, the probability of success in one trial is exactly $(\frac{1}{2})^{\nu(N)-1}$. (Since N is known to have at least two prime factors, this probability is at most $\frac{1}{2}$.) If $\nu(N)$ is exactly 2, then the probability that B does not accept N is, by Bernstein's law of large numbers,

$$\text{Prob}\{F_k(N) < \tfrac{3}{8}\} \leqq \text{Prob}\{|F_k(N) - \tfrac{1}{2}| \geqq \tfrac{1}{8}\} \leqq 2e^{-k(1/8)^2} = \delta_3(k).$$

On the other hand, if N has more than two prime factors, the probability of success in one trial is at most $\frac{1}{4}$, and thus the probability that B incorrectly accepts N (when interacting with a cheating A*) is

$$\text{Prob}\{F_k(N) \geqq \tfrac{3}{8}\} \leqq \text{Prob}\{|F_k(N) - \tfrac{1}{4}| \geqq \tfrac{1}{8}\} \leqq 2e^{-k(1/8)^2} = \delta_3(k).$$

To prove the minimum-knowledge property, given an interactive Turing machine B* we have to specify the computation of a simulating Turing machine M_{B^*}. The ensemble that M_{B^*} must simulate includes a sequence of Blum coin-flips, so we begin by showing that Blum's coin-flip protocol is perfectly minimum-knowledge. To prove this, we must specify the computation of a probabilistic polynomial-time Turing machine M_{coin}, whose output, on input N (satisfying $N \in \boldsymbol{BL}$ and $N \equiv 1 \bmod 4$), and initial history h, provides a simulation of the ensemble $VIEW_{B^*}\{(A, B^*)[N, h]\}$, which includes a message triple (v, σ, u) encoding a bit as described in § 5.1 above. Modelling the result oracle for the protocol, M_{coin} is given as additional input a (presumably random) bit b.

Given any bit b, M_{coin} (initializes B* and) proceeds as follows:

a. execute the protocol with B*:
   1. let B* "send" v (simulating step 1)
   2. save the current configuration of B*
   3. simulate A's action in step 2 by choosing $\sigma := \pm 1$ at random and "sending" it to B*
   4. let B* "send" u (simulating step 3)

b. if the bit encoded by (v, σ, u) is b, then write out B*'s virtual history (which includes the triple (v, σ, u)) and halt; otherwise:
   1. restore the saved configuration of B*
   2. simulating step 2 again, "send" $-\sigma$ (instead of σ) to B*
   3. let B* "send" u'
   4. write out B*'s virtual history (which includes the triple $(v, -\sigma, u')$)

Note that if B* does not follow the protocol, it may happen that the numbers u and u' are not the same; if their Jacobi symbol is the same the outcome of the protocol is the same random bit b and this has no effect on the output distribution of M_{coin} (since B*, when interacting with A, can decide to send either u or $-u$). On the other hand, if they have opposite Jacobi symbols mod N, then the outcome bit $1-b$ has been determined by B* and not chosen at random. As noted above, this can only happen if B* can factor N, in which case it indeed has the ability to dictate the outcome of the protocol, regardless of whether it is interacting with A or acting as a subroutine for M_{coin}.

Whether the virtual history of B* written out by M_{coin} was generated in step a or step b of the simulation, the distribution of its possible values (and thus the probability distribution of the bit encoded by the message triple) is identical to that of $VIEW_{B^*}\{(A, B^*)[N, h]\}$. Thus the coin-flip protocol is perfectly minimum-knowledge.

Next we describe the simulation by M_{B^*} of the third stage of our protocol. The set $VIEW_{B^*}\{(A, B^*)[N, h]\}$ that M_{B^*} must simulate begins with a sequence of Blum

coin-flips, which are used to generate random elements of $\mathbf{Z}_N^*$. This simulation can be performed by following the program M_{coin} as just described; the difficulty for M_{B^*}, a polynomial-time machine that may not be able to factor N, is that those elements which are quadratic residues must be randomly generated along with their square roots.

Given as input an integer N that has been confirmed by the first two stages and that satisfies $\nu(N)=2$, and given an initial history h, M_{B^*} proceeds as follows:

1. $i := 0$; $\Lambda :=$ the empty list
2. do until $i = k$:
   choose a random number a, $0 < a < N$;
   if g.c.d. $(a, N) \neq 1$ (which happens with vanishingly small probability) then FLAG the number a, adjoin it to Λ, and go to step 3;
   else:
   choose a random bit b (to decide the Jacobi symbol of the next element generated);
   if $b = 0$ then adjoin to Λ a random element of $\mathbf{Z}_N^*(-1)$;
   else:
   a. $i := i+1$
   b. choose $s_i \in \mathbf{Z}_N^*$ at random
   c. choose a random bit b_i (to decide whether the next element generated should be a quadratic residue);
      if $b_i = 0$ then $r_i := s_i^2 \bmod N$ (a random residue in $\mathbf{Z}_N^*(+1)$)
      else $r_i := -s_i^2 \bmod N$ (a random nonresidue in $\mathbf{Z}_N^*(+1)$)
   d. adjoin r_i to Λ
3. (simulate as many executions as needed of Blum's coin-flip in order to generate the sequence of bits in the list Λ)
   for each bit b in the representation of each number in Λ:
   follow the procedure above for M_{coin} (using B* as a subroutine), recording the numbers u (and possibly u') "sent" by B*;
   if the outcome of the coin-flip simulation is indeed b, then continue with the next bit in Λ;
   otherwise B* has "forced" the complementary outcome $1-b$ by "sending" u and u' with $(\frac{u}{N}) \neq (\frac{u'}{N})$, in which case:
   a. use u and u' to factor N
   b. discard the rest of Λ
   c. repeatedly execute Blum's coin-flip with B* (as originally specified, without backtracking to restore previous configurations of B*) in order to choose elements of $\mathbf{Z}_N^*$, bit by bit, until the resulting list contains k elements ($r_1, \ldots, r_k$, say) with Jacobi symbol $+1$; again let Λ denote the new list
4. if the last number in Λ is FLAGGED then halt
5. discard the elements in Λ with Jacobi symbol -1
6. if B* has not "forced" the outcome of any of the coin-flip simulations of step 3, then for each r_i in Λ such that $b_i = 0$ "send" (i, s_i) to B*;
   otherwise, use the factorization of N to test each r_i in Λ to see whether it is a quadratic residue; if it is, then compute s_i such that $r_i \equiv s_i^2 \bmod N$ and "send" (i, s_i) to B*
7. write out the virtual history of B*.

If $\nu(N)=2$, then a randomly chosen element of $\mathbf{Z}_N^*$—in particular, one that has been chosen by A interacting with a machine B* that does not "force" the outcome

of any Blum coin-flips—will have Jacobi symbol +1 with probability $\frac{1}{2}$; among these, quadratic residues will occur with probability $\frac{1}{2}$. If B*, as a subprogram of the simulator M_{B^*}, does not "force" any (simulated) Blum coin-flips, then the simulator generates elements of $\mathbf{Z}_N$ with exactly the same probabilities, and then perfectly simulates the generating coin-flips; on the other hand, if B* does "force" a coin-flip, then M_{B^*} simply performs with it a sequence of Blum coin-flips, exactly as in the specification of the protocol. Either way, M_{B^*} generates lists of elements of $\mathbf{Z}_N$ with the same distribution as do A and B*, and B* makes identical use of bits from its random tape, so that the sets $VIEW_{B^*}\{(A, B^*)[N, h]\}$ and $M[N, h]$ are identical. This completes the proof for the third stage.

Finally, to conclude the proof of Theorem 1, we see by the concatenation lemma that, given any input string at all, the concatenation of the three stages is a perfectly minimum-knowledge confirming interactive proof-system for the language $I_1 \cap I_2 \cap I_3 = I$. □

6. Interactive decision of quadratic residuosity. If the confirming part of our protocol has been successfully completed, then with high probability the input string (N, z) is in the language I. In particular, we know that $\nu(N) = 2$, that $z \in \mathbf{Z}_N^*$, and that $(\frac{z}{N}) = +1$; these are the properties that are required of the inputs to the next part of the protocol.

This part is a deciding interactive proof-system for L, taking inputs from I. The proof-system is both perfectly minimum-knowledge and perfectly result-indistinguishable. As noted above, a pair (N, z) that is known to belong to I either is or is not also a member of L according to whether or not z is a quadratic residue mod N.

To make the exposition clearer, we present three successive versions of our protocol.

Let $y \equiv -1 \bmod N$. Everything that follows holds for any nonresidue $y \in \mathbf{Z}_N^*$ that has Jacobi symbol +1. As long as $N \in \boldsymbol{BL}$ and $N \equiv 1 \bmod 4$, we can take $y = -1$. (*Remark.* If another nonresidue y is desired, A can prove to B, as a preliminary subprotocol stage, that y is a nonresidue by following the minimum-knowledge interactive proof-system of [17].)

Let us fix some notation. For any $x \in \mathbf{Z}_N^*$ we define the predicate:

$$\mathrm{RES}_N(x) = \begin{cases} 0 & \text{if } x \text{ is a quadratic residue mod } N, \\ 1 & \text{otherwise.} \end{cases}$$

Recall that $\mathbf{Z}_N^*(+1)$ denotes the set of elements of $\mathbf{Z}_N^*$ with Jacobi symbol +1. Since $\nu(N) = 2$, half of these are quadratic residues mod N, and half of them are nonresidues.

Our protocol relies on the fact that if $r \in \mathbf{Z}_N^*$ is chosen at random, then $r^2 \bmod N$ is a random quadratic residue in the set $\mathbf{Z}_N^*(+1)$ and $yr^2 \bmod N$ is a random quadratic nonresidue in $\mathbf{Z}_N^*(+1)$; similarly, $zr^2 \bmod N$ is either a random residue or a random nonresidue in $\mathbf{Z}_N^*(+1)$ according to whether or not z is a residue mod N.

This interactive proof-system has error probability $\delta(k) = 2e^{-4k/81}$.

First version: A deciding proof-system

i. Repeat k times:

1. B chooses $r \in \mathbf{Z}_N^*$ and $c \in \{1, 2, 3\}$ at random, and computes CASE c of:
   1: $x := r^2 \bmod N$
   2: $x := yr^2 \bmod N$
   3: $x := zr^2 \bmod N$
   B → A: x

2. A computes $b := \mathrm{RES}_N(x)$;
   A → B: b
3. B checks that if $c = 1$ then $b = 0$, if $c = 2$ then $b = 1$, and if $c = 3$ then b is consistent with any previous case-3 iterations; if not then B REJECTS the proof and halts the protocol

ii. B ACCEPTS the proof that $\mathrm{RES}_N(z)$ is equal to the consistent value of b for case-3 iterations.

As explained above, if z is a quadratic residue then x's constructed in case 1 are indistinguishable from x's constructed in case 3. If A acts as specified, then when the protocol finishes B will be convinced that z is a residue. The only way that a cheating A* can convince B that z is not a residue is by correctly guessing, among all iterations during which B has sent a residue x, which of these were constructed in case 1 and which of them in case 3; if there are ck such iterations in a particular execution of the protocol, then the probability of successful cheating is 2^{-ck}. Since c is very likely to be close to $\frac{2}{3}$, a simple calculation using Bernstein's law of large numbers shows that the probability of successful cheating is at most $2e^{-4k/81}$. Similarly if z is a quadratic nonresidue. Hence the above version is a deciding interactive proof-system for L.

However, this version is not result-indistinguishable. An observer of an execution of the protocol can easily tell whether he is watching an interactive proof that $\mathrm{RES}_N(z) = 1$ or a proof that $\mathrm{RES}_N(z) = 0$ by keeping a tally of the bits b sent by A in step 2 of each iteration.

Second version: A result-indistinguishable proof-system

A simple modification of the above protocol does hide the result from an eavesdropper. The only change is that at the beginning (before step i), A flips a fair coin in order to decide whether to use $R(x) = \mathrm{RES}_N(x)$ or $R(x) = 1 - \mathrm{RES}_N(x)$ as the bit b to be sent to B in step 2 of each iteration throughout the protocol. $R(x)$ can be regarded as an encoding, chosen at random for the entire protocol, of $\mathrm{RES}_N(x)$.

In step 3, B checks for consistency in the obvious way: B should receive the same bit b in all case-1 iterations and the complementary bit in all case-2 iterations; B should receive a consistent bit b in all case-3 iterations, and its value indicates to B whether or not z is a quadratic residue. As before, if in step 3 of any iteration B finds that the value of b is not consistent then B halts the protocol, REJECTing the proof.

With this modification, the protocol is still—arguing as above—a deciding interactive proof-system for L. Furthermore, it is result-indistinguishable. An eavesdropper expects to overhear one bit about $\frac{2}{3}$ of the time during step 2 of each iteration and the complementary bit the remaining $\frac{1}{3}$ of the time; whether the majority bit in a particular execution of the protocol is 0 or 1 gives him no knowledge. A formal proof of result-indistinguishability of the full protocol is presented below.

However, the version so far presented is not minimum-knowledge. For example, a cheating B* that wanted to find out whether a particular number—17, say—is a quadratic residue mod N could, during one of the iterations, send $x = 17$ in step 1 instead of an element x constructed at random according to B's program. A's response in step 2 will convey to B* the value $\mathrm{RES}_N(17)$, which is something that B* could not have computed by itself.

Third version: A minimum-knowledge result-indistinguishable proof-system

We can make this a minimum-knowledge protocol by refining step 1 of the version just presented; the refinement consists of several interactive substeps by which B gives to A what amounts to a minimum-knowledge proof that the element x that it sends

was constructed in one of the three ways specfied (without giving A any knowledge about which of the three ways). The rest of the protocol is unchanged.

1.0 B chooses $r \in \mathbf{Z}_N^*$ and $c \in \{1, 2, 3\}$ at random, and computes CASE c of:
1: $x := r^2 \bmod N$
2: $x := yr^2 \bmod N$
3: $x := zr^2 \bmod N$
B → A: x

1.1 B chooses $s_i \in \mathbf{Z}_N^*$ at random ($i = 1, \cdots, 4k$) and computes:
$T_1 = \{t_1, \cdots, t_k \mid t_i \equiv s_i^2 \bmod N\}$,
$T_2 = \{t_{k+1}, \cdots, t_{2k} \mid t_i \equiv ys_i^2 \bmod N\}$,
$T_3 = \{t_{2k+1}, \cdots, t_{3k} \mid t_i \equiv zs_i^2 \bmod N\}$,
$T_4 = \{t_{3k+1}, \cdots, t_{4k} \mid t_i \equiv yzs_i^2 \bmod N\}$;
taking this to be matrix of 4 rows $[T_1, T_2, T_3, T_4]$ and k columns (where column j contains the elements t_j, t_{k+j}, t_{2k+j}, t_{3k+j}), B randomly permutes each column of the matrix, resulting in a matrix T;
B → A: T

1.2 A chooses $S \subseteq \{1, \cdots, k\}$ at random (a *query* indicating a random subset of T's columns);
A → B: S

1.3 for each $j \in S$, for each $t_i \in$ column j, B → A: s_i
(These numbers show A that B has correctly computed the jth column of T for each $j \in S$ and convince A that it is very likely that at least one other column of T was also computed correctly.)

1.4 A verifies (for each such t_i) that $t_i \equiv$ either s_i^2, ys_i^2, zs_i^2, or $yzs_i^2 \bmod N$, with each congruence being satisfied once in each column $j \in S$;
if not, then A halts the protocol in the FAILURE state

1.5 for each $j \notin S$, for each $t_i \in$ column j, B computes w_i according to Table 1: if x was chosen as case c of step 1.0 and $t_i \in T_l$, then $w_i :=$ the table-entry in the lth row and cth column;

TABLE 1. (Step 1.5.)
(*All computations of table entries are modulo N.*)

	$x = \cdots$		
$t_i = \cdots$	r^2 ($c = 1$)	yr^2 ($c = 2$)	zr^2 ($c = 3$)
$s_i^2 \in T_1$	$rs_i = \sqrt{(xt_i)}$	$yrs_i = \sqrt{y(xt_i)}$	$zrs_i = \sqrt{z(xt_i)}$
$ys_i^2 \in T_2$	$yrs_i = \sqrt{y(xt_i)}$	$yrs_i = \sqrt{(xt_i)}$	$yzrs_i = \sqrt{yz(xt_i)}$
$zs_i^2 \in T_3$	$zrs_i = \sqrt{z(xt_i)}$	$yzrs_i = \sqrt{yz(xt_i)}$	$zrs_i = \sqrt{(xt_i)}$
$yzs_i^2 \in T_4$	$yzrs_i = \sqrt{yz(xt_i)}$	$yzrs_i = \sqrt{z(xt_i)}$	$yzrs_i = \sqrt{y(xt_i)}$

(For each $j \notin S$, these four numbers show A that if B has correctly computed the jth column of T, then B has correctly computed x.)
for each such t_i, B → A: w_i.

1.6 A verifies (for each such t_i) that $w_i^2 \equiv$ either (xt_i), $y(xt_i)$, $z(xt_i)$, or $yz(xt_i) \bmod N$, with each congruence being satisfied once in each column $j \notin S$;
if not, then A halts the protocol in the FAILURE state.

The protocol now continues as before. A sends $b = R(x)$ to B (step 2), and B checks b for consistency (step 3); and then they continue with step 1 of the next iteration. Note that it is in A's "interest" to choose S at random in step 1.2, so that with overwhelming probability both S and $\{1, \cdots, k\} - S$ are reasonably large (and thus the probability that any particular column of T will be queried is close to $\frac{1}{2}$).

The idea is that any machine playing the role of B (and desiring that the protocol succeed) must follow the protocol, because if it tries to cheat during any iteration—either by sending a number x in step 1.0 for which it does not "know" the corresponding number r, or by sending numbers t_i in step 1.1 for which it does not "know" the corresponding numbers s_i—then A will, with overwhelming probability, detect its cheating either in step 1.6 or in step 1.4. This is formalized in the following proof.

THEOREM 2. *Given input belonging to* $I = \{(N, z) \mid N \equiv 1 \bmod 4,\ N \in \mathbf{BL},\ \nu(N) = 2,\ (\frac{z}{N}) = +1\}$, *this protocol is a perfectly minimum-knowledge and perfectly result-indistinguishable deciding interactive proof-system for* $L = \{(N, z) \in I \mid z$ *a quadratic residue* $\bmod N\}$.

Proof. First we prove that the protocol is a deciding proof-system for L. Since we have already shown that the second version presented above is a proof-system, it suffices to show that the refinement of step 1 preserves this property.

Suppose that z is a quadratic residue. The question is whether a cheating A*—even if it does not choose S at random in step 1.2—can use the numbers sent by B during step 1 to distinguish correctly between case-1 iterations ($x = r^2 \bmod N$ for a random r) and case-3 iterations ($x = zr^2 \bmod N$). Since B has chosen them at random, A* is unable to distinguish between residues t_i of the form s_i^2 and residues t_i of the form zs_i^2. Table 2, the subtable of Table 1 corresponding to these four possibilities, has rows that are permutations of each other, and thus A* is not able to tell whether B is using column $c = 1$ or column $c = 3$ of the whole table.

TABLE 2
A subtable of Table 1.

		$c = 1$	$c = 3$
$t_i = \cdots$	s_i^2	$\sqrt{(xt_i)}$	$\sqrt{z(xt_i)}$
	zs_i^2	$\sqrt{z(xt_i)}$	$\sqrt{(xt_i)}$

Similarly for nonresidues t_i of the form ys_i^2 or yzs_i^2. A like analysis holds if z is a nonresidue mod N. Hence the protocol is indeed a deciding interactive proof-system for L.

In order to prove the minimum-knowledge property, we choose an interactive Turing machine B* that runs in expected polynomial time; we must describe the computation of a simulating machine $M = M_{B^*}$.

M has one-time access to an oracle for the result of the protocol, as explained in § 3.1. M begins by querying the oracle on the input string (N, z), and learns (with very high probability) the value of $\mathrm{RES}_N(z)$. The rest of the simulation is similar to that of the proof that the protocol of [17] is minimum-knowledge.

As its next step, M flips a coin to simulate A's choice of whether to compute $R(x) = \mathrm{RES}_N(x)$ or $R(x) = 1 - \mathrm{RES}_N(x)$ during the protocol.

In each iteration, M carries on the protocol through the end of (the refinement of) step 1 in a straightforward manner: M uses B* to perform its own version of B's role, and M easily simulates A's role, choosing a random query S in step 1.2 and

checking several congruences mod N in steps 1.4 and 1.6. If these congruences do satisfy the check, the difficulty comes in simulating A's communication in step 2, which consists of the bit $R(x)$; how can M quickly calculate the correct value of $\mathrm{RES}_N(x)$? M accomplishes this by following the EXTRACTION procedure described below.

After B* has performed its computations in the simulation of step 1.1 and "sent" the matrix T, M saves the current configuration C_0 of B*. At this point, given C_0 (which includes a fixed random-tape string) and any fixed query-set $S \subseteq \{1, \cdots, k\}$ that A might choose in step 1.2, the lists of numbers that B* would "send" in steps 1.3 and 1.5 in answer to the query S are determined. Let us call S a *satisfiable* query if these answers would satisfy A's verifying checks of steps 1.4 and 1.6, causing the protocol to continue with step 2. (A query that is not satisfiable would cause A to halt the protocol in its failure state.) It is easy to check whether or not a query S is satisfiable, by setting B*'s configuration to C_0, "sending" S to B*, and checking the numbers that B* "sends" in return.

In its simulation, M makes use of an auxiliary matrix T' that contains two data fields for each entry t_i of the matrix T, one for the number s_i and one for the number w_i (where s_i and w_i are related to t_i as in the specification of the protocol). Note that if M succeeds in filling both fields for any single entry t_i, then M can easily deduce the value $R(x)$ that it needs in order to simulate step 2: M can use s_i to see how t_i was computed in step 1.1 (i.e., which set T_j contains t_i, and hence which row of the table B* must use in step 1.5); next M can use w_i to see which column c of the table B* must use; and then the choice of column gives M the value of $\mathrm{RES}_N(x)$, and hence of $R(x)$.

Next we describe the EXTRACTION procedure that M performs in each iteration following the simulation of step 1.1.

1. save the current machine configuration C_0
2. choose a query $S \subseteq \{1, \cdots, k\}$ at random, store it, and "send" it to B* (simulating step 1.2)
3. let B* "send" its answers to S (the numbers s_i of step 1.3 and $w_{i'}$ of step 1.5), and check the congruences of steps 1.4 and 1.6;
   if the congruences do not check, then halt the simulation;
   otherwise, store B*'s answers in the auxiliary matrix T' and repeat the following two loops concurrently until *success*:
   a. sampling the query space (without repetition):
      i. restore configuration C_0
      ii. choose a new query $S' \subseteq \{1, \cdots, k\}$ at random that has not already been chosen (if there is one; if none exists, then halt the sampling loop);
      store S' and "send" it to B*
      iii. let B* "send" its answers to S'
      iv. for each $j = 1 \cdots k$ if B*'s answers for column j of the matrix T satisfy the congruences of 1.4 (if $j \in S'$) or of 1.6 (if $j \notin S'$), then enter them in the auxiliary matrix T';
      if any of these new entries is an s_i for which T' already contains w_i or vice versa, then (as explained above) use s_i, w_i to compute $R(x)$ and set *success* := TRUE
   b. use any factoring algorithm F to factor N:
      i. until (*success* or {F has successfully factored N}) do the next step of F
      ii. use the prime factors of N to compute $\mathrm{RES}_N(x)$ and $R(x)$, and set *success* := TRUE

4. restore configuration C_0 and "send" to B* the original query S
5. let B* "send" its answers and update its history tape (exactly as it did the first time it received the query S)

Simulating step 2, M sets $b := R(x)$ (as computed either in a or b of the last inner loop) and "sends" b to B*, which performs its version of step 3 of the protocol. If B* is following the instructions of step 3, then it is indeed "expecting" the computed value of b, and the simulation continues with the next iteration.

We need to show that M's expected running time is polynomial (in k, the length of the input) and that its output ensemble is identical to B*'s view. To bound the running time, it suffices to prove a polynomial bound on the expected time required by each of the k iterations of M's program. First observe that the outer loop of the EXTRACTION procedure takes polynomial time. The same is true for any single execution of the inner loop: queries may be stored in a lexicographically ordered tree (so that choosing a new one costs $O(k)$); the rest of the sampling loop is polynomial-time, and in each inner loop only one step of the factoring algorithm is performed. Therefore, it is enough to show that, for any fixed configuration C_0, the expected number of repetitions of the inner loop is polynomial in k.

In fact, we show that this number is constant. In configuration C_0, let p $(0 \leqq p \leqq 2^k)$ be the number of queries that are satisfiable. When M performs the EXTRACTION procedure, with probability $1 - p/2^k$ its first query S will not be satisfiable, in which case the inner loop is not executed at all. If $p = 0$, we have no other case to consider; so assume $p \geqq 1$. With probability $p/2^k$, S is satisfiable, and the inner loop is repeated until *success* is set to TRUE (either in the sampling process or after factoring N). Each sampling loop begins with the choice of a new random query. Of the $2^k - 1$ possible queries, at least the $p - 1$ satisfiable queries (besides S) lead to *success*; that is, the probability of a successful inner loop is at least $(p-1)/(2^k - 1)$. Hence if $p > 1$, the expected number of attempts after choosing a satisfiable original query is at most $(2^k - 1)/(p-1)$; overall, the expected number of repetitions of the inner loop is at most $(p/2^k) \cdot (2^k - 1)/(p-1) \leqq 2$. We consider below the special case $p = 1$.

Next, we show that the sets $VIEW_{B^*}\{(A, B^*)[(N, z), h]\}$ and $M[(N, z), h]$ are identical; following the Remark of § 3.2, it suffices to show that this is so for any one of the k iterations of the protocol. Consider, therefore, an iteration—either of an actual protocol execution by A and B* or of the simulation by M—at the beginning of which B* sends the matrix T, and let p $(0 \leqq p \leqq 2^k)$ be the number of satisfiable queries. With probability $1 - p/2^k$ a randomly chosen query is not satisfiable, causing either the protocol execution or the simulation to halt; in this case, the actual history and the virtual history are identical. If $p = 0$, then this is the only case that occurs. Otherwise, with probability $p/2^k$, the original query S is satisfiable, and both the actual protocol and the simulation continue with step 2. As long as $p \geqq 2$, there is at least one other query that leads to *success* in the inner loop of M's EXTRACTION procedure, enabling M to "send" in its simulation of step 2 the correct value of $b = R(x)$, the same one that A would send during an actual execution. The factoring algorithm may be faster then the sampling process, in which case the correct value of b is computed directly. Either way, the actual history and the virtual history are identical.

If $p = 1$, then the probability that M's original query is satisfiable is only 2^{-k}. In this rare case, the sampling process in the inner loop of the EXTRACTION procedure might never lead to *success*; the inner loop might not terminate until after N has been factored. Since the cost of factoring N is less than $O(2^k)$, the total expected number of repetitions of the inner loop when $p = 1$ is less than $2^{-k} \cdot 2^k = 1$. In this case, as

before, the actual history and the virtual history are identical. This concludes the proof that the protocol is perfectly minimum-knowledge.

In order to prove that the protocol is result-indistinguishable, we must specify the computation of a probabilistic Turing machine M′, running in expected polynomial time, that simulates the communications ensemble $COM\{(\mathrm{A}, \mathrm{B})[N, z]\}$. (Recall that M′ does not have access to any oracle.) M′ begins by flipping a coin to decide whether to simulate the choice $R(z)=0$ or the choice $R(z)=1$. Then in each iteration M′ simulates the specified computations of A and B, except for the following changes. In (simulated) step 1.0, M′ chooses $x := zr^2 \bmod N$ with probability $\frac{2}{3}$ and $x := yzr^2 \bmod N$ with probability $\frac{1}{3}$. In (simulated) step 2, M′ outputs $b = R(z)$ if $x = zr^2$ and $b = 1 - R(z)$ if $x = yzr^2$. (Here the simulation of step 2 is much simpler than in the minimum-knowledge proof above, since M′ "knows" how each x was constructed.) In (simulated) step 1.5, M′ outputs w_i computed according to Table 3.

TABLE 3

	$x = \cdots$	
$t_i = \cdots$	zr^2	yzr^2
s_i^2	$zrs_i = \sqrt{z(xt_i)}$	$yzrs_i = \sqrt{yz(xt_i)}$
ys_i^2	$yzrs_i = \sqrt{yz(xt_i)}$	$yzrs_i = \sqrt{z(xt_i)}$
zs_i^2	$zrs_i = \sqrt{(xt_i)}$	$yzrs_i = \sqrt{y(xt_i)}$
yzs_i^2	$yzrs_i = \sqrt{y(xt_i)}$	$yzrs_i = \sqrt{(xt_i)}$

The numbers x output by M′ have the same distribution as the numbers x output by B; the same is true of the s_i and the w_i. Hence, as required, the set of outputs $\mathrm{M}'[N, z]$ is identical to the set $COM\{(\mathrm{A}, \mathrm{B})[N, z]\}$, so the protocol is perfectly result-indistinguishable.

As presented, the protocol takes $O(k)$ communication rounds during which $O(k^3)$ bits are exchanged. However, all k iterations of the main loop can be performed in parallel, taking $O(1)$ rounds. The simulator M can perform in parallel all k iterations of its main loop, and its expected running time is still polynomial in k. Similarly, M′ can operate in parallel. Thus the parallelized version of the protocol is also perfectly minimum-knowledge and perfectly result-indistinguishable. This concludes the proof of Theorem 2. □

We note that there is another modification of the first version of our protocol that also achieves result-indistinguishability. A can always respond in step 2 with the true value of $\mathrm{RES}_N(x)$ if B computes each x in step 1 according to a random choice among four varieties: to the types r^2, yr^2, and $zr^2 \bmod N$ we add the fourth type $yzr^2 \bmod N$. If the protocol is to be minimum-knowledge as well, we can refine step 1 as in the third version of our protocol, adding an appropriate fourth column to the table used to compute w_i.

7. Cryptographic applications. In all our applications, we let N be the public key of a user A who knows its factorization. Within the set $\boldsymbol{N}$, it is most advantageous to

A to choose N to be of the form $N = pq$, with p and q of approximately the same size. A can follow our confirming protocol in order to prove to any other user that $N \in \boldsymbol{BL}$ and $\nu(N) = 2$. For these applications, we assume that the residuosity problem is intractable.

When A communicates with another user B, any element $z \in \mathbf{Z}_N^*(+1)$ can serve as an encoding of the bit $\mathrm{RES}_N(z)$, as soon as A has used our protocol to prove to B the value of this bit. According to need, z can be chosen by A or by both A and B together (say, by flipping coins). Because of the result-indistinguishability of the protocol, this encoding is cryptographically secure.

In contrast, the conventional approach to hiding knowledge from an eavesdropper is to use encryption. (For example, given two different protocols, one for membership in a language L and the other for nonmembership in L, one could "pad" the protocols so that they both caused messages of the same length to be sent at each round of communications, and then encrypt all messages.) However, in this approach, proving a theorem about the security of the protocol against eavesdroppers usually requires an assumption about the security of the encryption scheme used.

The result-indistinguishability of our protocol suggests two different ways that it can be used in a public-key encryption scheme that is secure against chosen-ciphertext attack.

(1) A sequence of random elements $z_1, z_2, \cdots$ can serve as a probabilistic encryption [16] of the bit-sequence $\mathrm{RES}_N(z_1), \mathrm{RES}_N(z_2), \cdots$, which in turn can be used as a one-time pad to encode a message sent either from A to B or from B to A.

(2) Instead of using the z_i directly to encrypt the bits $\mathrm{RES}_N(z_i)$, we can define a much more efficient scheme for probabilistic encryption by using a short bit-sequence $\mathrm{RES}_N(z_1), \mathrm{RES}_N(z_2), \cdots$ as the random seed for a cryptographically secure pseudorandom bit generator [5], [27], [6] whose security may be based on the unknown factorization of N (e.g., [3], [4]). Sharing the seed, A and B can efficiently generate polynomially many bits and use them as a (very long) one-time pad with which to send messages back and forth. The pad bits alone are secure against any polynomially bounded adversary; furthermore, the adversary gains no computational advantage in guessing any pad bit when he is given probabilistic encryptions of the bits of the seed, nor when he is allowed to overhear the protocol interactions that define these encrypted bits. Because our protocol is only used in order to initialize the system, this scheme has low amortized cost.

Whether the bits $\mathrm{RES}_N(z_i)$ are used directly or to form the seed of a pseudorandom bit generator, the resulting schemes have the minimum-knowledge property with respect to B as well as with respect to an eavesdropper C. In particular, they are provably secure against both chosen-message and chosen-ciphertext attack. For precise definitions of levels of cryptographic security, and for further study of the power that interaction seems to add to public-key cryptography, see [13], [10].

Another application of our protocol gives a new private unbiased coin-flip, generated jointly by A and B. The two users simply choose z at random—for example, choosing its bits by means of Blum's coin-flip. Note that the bits of z are public; it is $\mathrm{RES}_N(z)$, the result of the coin-flip, which is private.

In certain applications we can omit the confirming proof that N is of the required form. Suppose in fact that N has more than two prime factors. For any $z \in \mathbf{Z}_N^*(+1)$, A can carry out the deciding protocol as before. Now, however, if y and z—both quadratic nonresidues in $\mathbf{Z}_N^*(+1)$—have different quadratic character modulo several

of the prime factors of N, then A can distinguish numbers of the form r^2 from numbers of the form $yr^2 \bmod N$ and can distinguish each of these from numbers of the form $zr^2 \bmod N$. (This is not true if $\nu(N) = 2$; recall that for such N any nonresidue in $\mathbf{Z}_N^*(+1)$ is a nonresidue modulo both prime factors of N.) Thus A can, at will, use our deciding protocol to "prove" to B either that z is a residue or that z is a nonresidue. In either case, the interactively proved value of $\mathrm{RES}_N(z)$—whether or not it is the true value—is cryptographically secure. This value gives B no knowledge whatever. The "proof" only convinces B that A can distinguish between numbers with different quadratic characters mod N, without releasing to B any information about the quadratic character mod N of any particular number. (This can be formalized in terms of a simulator $\mathrm{M} = \mathrm{M}_{\mathrm{B}^*}$ for any given verifier B*. Note that at the beginning of the program for M given in the proof of Theorem 2, we can replace the oracle query for $\mathrm{RES}_N(z)$ with a simple coin-flip; then exactly as in that proof, the two sets

$$VIEW_{\mathrm{B}^*}\{(\mathrm{A}, \mathrm{B})[(N, z), h]\}$$

and

$$\mathrm{M}[(N, z), h]$$

are identical.) Thus, we may say that in this case, the protocol is result-indistinguishable even with respect to B.

In this situation, when N has more than two prime factors, we can define the following game: A picks a random nonresidue z with quadratic character different from that of y. A then "proves" to user B_1 that $\mathrm{RES}_N(z) = b_1$, and "proves" to user B_2 that $\mathrm{RES}_N(z) = b_2$. The "proven" value of $\mathrm{RES}_N(z)$ in each execution of the protocol is shared only by the prover A and the verifier B_1 or B_2. In fact, user B_1 has absolutely no computational advantage in deciding whether or not $b_1 = b_2$, and neither does user B_2.

8. Conclusions. Approaching knowledge from the point of view of computational complexity, we have studied the interactive transmission of computational results. The protocol that we introduce gives a proof of the value, 0 or 1, of a number-theoretic predicate, $\mathrm{RES}_N(\cdot)$. In a sense that we make precise (extending the definitions of [17]), the verifier gains no more knowledge from an execution of the protocol than this value; this is the "minimum-knowledge" property of the protocol. Furthermore, we are able to analyze the difference between the knowledge gained by the active verifier and that gained by a passive eavesdropper of equal computational power; the protocol is "result-indistinguishable," in that an eavesdropper gains no knowledge at all by overhearing the messages passed during an execution.

Recent work on minimum-knowledge protocols has taken several different directions. Feige, Fiat, and Shamir adapted the result-indistinguishable protocol of this paper (originally presented in [11]) and the protocols of [17] in order to give an efficient minimum-knowledge (and therefore cryptographically secure) identification scheme [9]. Their paper proposes a formalization, similar to that of Tompa and Woll [26], of the notion that a protocol can supply a "proof" that the prover knows some fact or possesses some computational ability, while completely hiding this piece of knowledge. (For example, in case N has more than two prime factors, our deciding proof-system for $\mathrm{RES}_N(\cdot)$ may be regarded as demonstrating the prover's ability to distinguish between numbers with different quadratic characters mod N; see § 7.)

Goldreich, Micali, and Wigderson proved that, under the assumption that one-way functions exist, every language in NP has a minimum-knowledge confirming interactive proof-system; this result has important consequences for the design of cryptographic

protocols [14]. Under the assumption that certain number-theoretic computations are infeasible, a similar result was proved by Brassard and Crepeau, both for prover and verifier as described in this paper [7], and for the dual situation in which a resource-bounded prover interacts with a verifier of unlimited computational power [8]. (Our formalization of the requirement that a two-party transfer protocol be minimum-knowledge applies to protocols that depend on such "cryptographic assumptions;" and under the appropriate assumption, the concatenation lemma of § 3.2 holds in the cryptographic setting.) Impagliazzo and Yung gave a construction for the direct minimum-knowledge transfer of the result of any given computation (both for the usual and for the dual model of the computational power of the prover and the verifier); the dual protocol is implemented under the more general assumption that any of a large class of one-way functions exist [20]. Their construction applies to probabilistic as well as deterministic computations, and in particular it provides a minimum-knowledge interactive proof-system for any language possessing a confirming interactive proof-system at all (i.e., for any language in the complexity class IP [1], [18]).

In a recent paper, instead of considering only protocols for transferring a computational result from one party to another, Yao studied a broad class of two-party protocols for what may be called "crytographic computation," in which the (polynomially bounded) users combine their private inputs in order to compute private outputs in a minimum-knowledge fashion, preserving the privacy of these inputs and outputs and hiding partial computational results as much as possible; it may also be required that both users compute their final results simultaneously [28]. Under the asssumption that factoring is hard, Yao showed how to design such a protocol for any given cryptographic computation problem. Continuing this work, Goldreich, Micali, and Wigderson proved similar results for multiparty protocols, assuming that one-way trapdoor functions exist, and showed how such protocols could be made to tolerate faults [15]. Galil, Haber, and Yung simplified and extended these constructions for cryptographic computation, giving new methods for the design of fault-tolerant multiparty cryptographic protocols [12].

In summary, the complexity-theoretic approach to measuring and controlling the knowledge transmitted in various distributed and cryptographic settings has proved to be a useful tool in protocol design.

Acknowledgments. We would like to thank Silvio Micali and Charles Rackoff for their helpful discussions, and Paul Beame, Gilles Brassard, Joan Feigenbaum, Shafi Goldwasser, David Lichtenstein and Adi Shamir for their insightful remarks.

Note added in proof. The concatenation lemma of § 3.2 was proved independently in Y. Oren, *On the cunning power of cheating verifiers: Some observations about zero-knowledge proofs*, Proc. 28th Annual IEEE Symposium on the Foundations of Computer Science, 1987, pp. 462–471, and in [26].

REFERENCES

[1] L. BABAI, *Trading group theory for randomness*, Proc. 17th Annual ACM Symposium on the Theory of Computing, 1985, pp. 421–429.

[2] M. BLUM, *Coin flipping by phone*, Proc. IEEE COMPCON, 1982, pp. 132–137.

[3] L. BLUM, M. BLUM, AND M. SHUB, *A simple unpredictable pseudo-random number generator*, SIAM J. Comput., 15 (1986), pp. 364–383.

[4] M. BLUM AND S. GOLDWASSER, *An efficient probabilistic public-key encryption scheme which hides all partial information*, in Proc. Crypto '84, Springer-Verlag, New York, Berlin, 1984, pp. 289–299.

[5] M. BLUM AND S. MICALI, *How to generate cryptographically strong sequences of pseudo-random bits*, SIAM J. Comput., 13 (1984), pp. 850–864.

[6] R. B. BOPPANA AND R. HIRSCHFELD, *Pseudorandom generators and complexity classes*, in Advances in Computer Research, Volume on Randomness and Computation, JAI Press, to appear.

[7] G. BRASSARD AND C. CREPEAU, *Zero-knowledge simulation of Boolean circuits*, Proc. Crypto '86, Springer-Verlag, New York, Berlin, 1987, pp. 223–233.

[8] ———, *Non-transitive transfer of confidence: A perfect zero-knowledge interactive protocol for* SAT *and beyond*, in Proc. 27th Annual IEEE Symposium on the Foundations of Computer Science, 1986, pp. 188–195.

[9] U. FEIGE, A. FIAT, AND A. SHAMIR, *Zero knowledge proofs of identity*, Proc. 19th Annual ACM Symposium on the Theory of Computing, 1987, pp. 210–217.

[10] Z. GALIL, S. HABER, AND M. YUNG, *Symmetric public-key encryption*, Proc. Crypto '85, Springer-Verlag, New York, Berlin, 1985, pp. 128–137.

[11] ———, *A private interactive test of a Boolean predicate and minimum-knowledge public-key cryptosystems*, Proc. 26th Annual IEEE Symposium on the Foundations of Computer Science, 1985, pp. 360–371.

[12] ———, *Cryptographic computation: Secure fault-tolerant protocols and the public-key model*, Proc. Crypto '87, Springer-Verlag, New York, Berlin, 1988, pp. 135–155.

[13] ———, *Symmetric public-key cryptosystems*, 1989, submitted.

[14] O. GOLDREICH, S. MICALI, AND A. WIGDERSON, *Proofs that yield nothing but their validity and a methodology of cryptographic protocol design*, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, pp. 174–187.

[15] ———, *How to play any mental game*, Proc. 19th Annual ACM Symposium on the Theory of Computing, 1987, pp. 218–229.

[16] S. GOLDWASSER AND S. MICALI, *Probabilistic encryption*, JCSS, 28 (1984), pp. 270–299.

[17] S. GOLDWASSER, S. MICALI, AND C. RACKOFF, *The knowledge complexity of interactive proof systems*, Proc. 17th Annual ACM Symposium on the Theory of Computing, 1985, pp. 291–304; SIAM J. Comput., 18 (1989), pp. 186–208.

[18] S. GOLDWASSER AND M. SIPSER, *Private coins versus public coins in interactive proof systems*, Proc. 18th Annual ACM Symposium on the Theory of Computing, 1986, pp. 59–68.

[19] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Oxford University Press, London, 1954.

[20] R. IMPAGLIAZZO AND M. YUNG, *Direct minimum-knowledge computations*, Proc. Crypto '87, Springer-Verlag, New York, Berlin, 1988, pp. 40–51.

[21] E. KRANAKIS, *Primality and Cryptography*, John Wiley, New York, 1986.

[22] M. LUBY, S. MICALI, AND C. RACKOFF, *How to simultaneously exchange a secret bit by flipping a symmetrically-biased coin*, Proc. 24th Annual IEEE Symposium on the Foundations of Computer Science, 1983, pp. 11–22.

[23] I. NIVEN AND H. S. ZUCKERMAN, *An Introduction to the Theory of Numbers*, John Wiley, New York, 1972.

[24] C. H. PAPADIMITRIOU, *Games against nature*, Proc. 24th Annual IEEE Symposium on the Foundations of Computer Science, pp. 446–450.

[25] A. RENYI, *Foundations of Probability*, Holden-Day, New York, 1970.

[26] M. TOMPA AND H. WOLL, *Random self-reducibility and zero knowledge interactive proofs of possession of information*, Proc. 28th Annual IEEE Symposium on the Foundations of Computer Science, 1987, pp. 472–482.

[27] A. C. YAO, *Theory and applications of trapdoor functions*, Proc. 23rd Annual IEEE Symposium on the Foundations of Computer Science, 1982, pp. 80–91.

[28] ———, *How to generate and exchange secrets*, Proc. 27th Annual IEEE Symposium on the Foundations of Computer Science, 1986, pp. 162–167.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 740-747, August 1989

007

AN OPTIMAL SYNCHRONIZER FOR THE HYPERCUBE*

DAVID PELEG† AND JEFFREY D. ULLMAN‡

Abstract. The synchronizer is a simulation methodology introduced by Awerbuch [*J. Assoc. Comput. Math.*, 32 (1985), pp. 804-823] for simulating a synchronous network by an asynchronous one, thus enabling the execution of a synchronous algorithm on an asynchronous network. In this paper a novel technique for constructing network synchronizers is presented. This technique is developed from some basic relationships between synchronizers and the structure of a *t-spanning subgraph* over the network. As a special result, a synchronizer for the hypercube with optimal time and communication complexities is obtained.

Key words. distributed computation, networks, synchronization, spanners

AMS(MOS) subject classifications. 68R10, 68M10

1. Introduction. Algorithms for synchronous networks are easier to design, debug, and test than similar algorithms for asynchronous networks. Consequently, it is desirable to have a uniform methodology for transforming an algorithm for synchronous networks into an algorithm for asynchronous networks. This tool will enable one to design an algorithm for a synchronous network, test it and analyze it in that simpler environment, and then use the standard methodology to transform the algorithm into an asynchronous one, and use it in the asynchronous network.

This general approach for handling asynchrony was introduced by Awerbuch in [A1], and referred to as a *synchronizer.* His paper proposes several specific methods for implementing a simulation of this type, and studies their complexities. Later Awerbuch [A2] demonstrated the surprising fact that despite the inevitable overheads involved in such a simulation, asynchronous algorithms designed in this way are sometimes more efficient than any previously known. This is mainly due to the inherent difficulty in reasoning about an asynchronous network, which sometimes makes it hard to reach an optimal solution for a problem directly in such an environment. This phenomenon was demonstrated on several graph problems, such as breadth-first-search and maximum flow [A2], for which algorithms obtained by combining a standard synchronous algorithm with a synchronizer yield an asynchronous algorithm with better performance (in terms of time and/or number of messages) than all previously known ones.

It is clear that every synchronizer incurs some time and communication costs for the synchronization of every round. Let us denote the time and communication requirements added by a synchronizer ν for each pulse (timestep) of the synchronous algorithm by $T(\nu)$ and $C(\nu)$, respectively. Clearly, an efficient synchronizer should keep these costs as low as possible. In [A1], Awerbuch presents three synchronizers, α, β, and γ. These synchronizers demonstrate a certain trade-off between their communication and time requirements.

* Received by the editors May 18, 1987; accepted for publication (in revised form) August 26, 1988. An extended abstract of this paper has appeared in the *Proceedings of the 6th Annual* ACM *Symposium on Principles of Distributed Computing*, August 1987.

† Department of Computer Science, Stanford University, Stanford, California 94305. Present address, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel. The work of this author was partially supported by a Weizmann fellowship and a grant from Stanford University's Center for Integrated Systems.

‡ Department of Computer Science, Stanford University, Stanford, California 94305. The work of this author was partially supported by the Office of Naval Research contract N00014-C-85-0731.

The first two simple synchronizers represent the two endpoints of this trade-off. Synchronizer α is time optimal, i.e., $T(\alpha) = O(1)$, but its communication complexity is $C(\alpha) = O(|E|)$. On the other hand, synchronizer β achieves an optimal number of messages (i.e., $C(\beta) = O(|V|)$), but its time requirement may be high, namely, $T(\beta) = O(D)$. Here V, E, and D denote the set of vertices, the set of edges, and the diameter of the network, respectively. (Note that these complexities are crucial since they represent the overhead *per pulse* of the synchronous algorithm.) The third, more involved synchronizer γ is a combination of the two previous ones, which achieves some reasonable middle points on this scale. In particular, it is possible to achieve $C(\gamma) = O(k|V|)$ and $T(\gamma) = O(\log_k |V|)$, where k is a parameter taken from the range $2 \leqq k < |V|$.

Awerbuch also proves some lower bounds which demonstrate that for some networks, the best possible improvements are within constant factors from synchronizer γ [A1], [A3]. However, this lower bound is not global, and for various networks one can do better. For example, note that for the class of bounded-degree networks, synchronizer α is optimal in both time and messages, since $|E| \in O(|V|)$. This covers many common architectures proposed for parallel computing, such as meshes, butterflies and cube-connected cycles, rings, etc. The same holds also for the class of trees and planar graphs in general. Likewise, for bounded-diameter networks, synchronizer β is optimal. Thus the problem remains interesting only for graph classes that are in between.

A notable example of a network for which the current solutions are not satisfactory is the hypercube (cf. [P], [U]). It is possible to construct synchronizers of type α, β or γ for the hypercube using the constructions of [A1], but the resulting complexities are not optimal. Furthermore, in the sequel we note that the hypercube has an even better synchronizer of type γ, which can be obtained by direct construction (rather than by using the algorithm of [A1]) and whose complexities are $T = O(\log \log V)$ and $C = O(V)$, which is still not optimal. The main result of this paper is the construction of an optimal synchronizer (with $T = O(1)$ and $C = O(|V|)$) for the hypercube.

On our way toward this goal, we introduce a novel general methodology for a synchronizer (that we term δ, for purposes of consistency). This methodology is developed by exploiting the close connection between synchronizers and the structure of a *t-spanner* on a network. Given a network $G = (V, E)$, a subgraph $G' = (V, E')$ is a *t-spanner* of G if for every $(u, v) \in E$, the distance between u and v in G' is at most t.

In the sequel we derive the following basic connections between t-spanners and synchronizers.

THEOREM 1. (1) *If the network G has no t-spanner with at most m edges, then every synchronizer ν for G requires either $T(\nu) \geqq t+1$ or $C(\nu) \geqq m+1$.*

(2) *If the network G has a t-spanner with m edges, then it has a synchronizer δ with $T(\delta) = O(t)$ and $C(\delta) = O(tm)$.* □

In fact, the lower bounds of [A1], [A3] implicitly use part (1) of Theorem 1.

We then proceed to prove the existence of a 3-spanner with a linear ($O(2^n)$) number of edges for the hypercube of dimension n (henceforth, the n-cube). This yields the desired optimal synchronizer for the hypercube.

THEOREM 2. *For every $n \geqq 0$, the n-cube has a synchronizer of type δ with optimal time and communication complexities $T(\delta) = O(1)$ and $C(\delta) = O(2^n)$.* □

Spanners turn out to have other applications in the area of communication networks, e.g., in designing memory-efficient routing strategies (cf. [PU]). Additional results on the existence of spanners for various graph classes can be found in [PS].

The rest of the paper is organized as follows. Section 2 contains the definitions of the synchronous and asynchronous models, and § 3 gives an overview of the basic structure of a synchronizer. Finally, in § 4 we discuss the relationships between synchronizers and t-spanners and prove Theorem 1, and in § 5 we construct an optimal synchronizer of type δ for the hypercube and thus prove Theorem 2.

2. The model. We consider the standard model of an asynchronous point-to-point communication network (e.g., [A1], [GHS]). The network is described by an undirected graph $G=(V, E)$. The nodes of the graph represent the processors of the network and the edges represent bidirectional communication channels between the processors.

All the processors have distinct identities. There is no common memory, and algorithms are event-driven (i.e., processors cannot access a global clock in order to decide on their action). Messages sent from a processor to its neighbor arrive within some finite but unpredictable time. Each message contains a fixed number of bits, and therefore carries only a bounded amount of information.

A synchronous network is a variation of the above model in which all link delays are bounded. More precisely, each processor keeps a local clock, whose pulses must satisfy the following property. A message sent from a processor v to its neighbor u at pulse p of v must arrive at u before pulse $p+1$ is generated by u.

Our complexity measures are defined as follows. The *communication complexity* of an algorithm A, C_A, is the worst-case total number of messages sent during the run of the algorithm. The *time complexity* of an algorithm A, T_A, is defined as follows. For a synchronous algorithm, T_A is the number of pulses generated during the run. For an asynchronous algorithm, T_A is the worst-case number of time units from the start of the run to its completion, assuming that each message incurs a delay of at most one time unit. This assumption is used only for performance evaluation, and does not imply that there is a bound on delay in asynchronous networks.

Next, let us formally define the hypercube of dimension n, $H_n=(V_n, E_n)$. This network is defined by $V_n=\{0, 1\}^n$ and

$$E_n=\{(x, y)\,|\,x, y\in V_n, x \text{ and } y \text{ differ in exactly one bit}\}.$$

The network has $|V_n|=2^n$ nodes, $|E_n|=n\cdot 2^{n-1}$ edges, and diameter n.

Finally, let us define the *Cartesian product* of two graphs. Let $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$. The Cartesian product of G_1 and G_2, denoted $G_1\times G_2$, is defined as

$$G_1\times G_2=(V_1\times V_2, E)$$

where

$$E=\{((u_1, v_1), (u_2, v_2))\,|\,(u_1=u_2 \text{ and } (v_1, v_2)\in E_2) \text{ or } (v_1=v_2 \text{ and } (u_1, u_2)\in E_1)\}.$$

Hence $G_1\times G_2$ is constructed by substituting a copy of G_2 for each vertex in G_1 and drawing in edges between corresponding nodes of adjacent copies.

The H_n hypercube can be viewed as the Cartesian product graph $H_n=H_{n-k}\times H_k$, for any $0\leqq k\leqq n$. Later, we make use of this characterization of the cube.

3. Synchronizers. Th synchronizer is intended to enable any synchronous algorithm to run on any asynchronous network. The goal of the simulation is to generate a sequence of local clock pulses at each processor of the network, satisfying the following property: pulse number p is generated by a processor only after it received all the messages of the algorithm sent to it by its neighbors during their pulse number $p-1$.

This property is easy to guarantee as long as we restrict our attention to synchronous algorithms with complete communication, i.e., algorithms that require every processor to send messages to every neighbor at every time pulse. The obvious problem with

partial communication algorithms is that in the case where processor v does not send any message to its neighbor u at a certain pulse, u is obliged to wait forever for a message, as link delays in the asynchronous network are unpredictable.

The conceptual solution proposed in [A1] consists of two additional phases of communication. The first phase simply requires every processor receiving a message from a neighbor to send back an acknowledgment. This way, every processor learns, within finite time, that all the messages it sent during a particular pulse have arrived. Such a processor is said to be *safe* with respect to that pulse. Note that introducing this phase does not increase the message complexity or the time complexity of the algorithm by more than a constant factor.

A processor may generate a new pulse whenever it learns that all its neighbors are safe with respect to the current pulse. Thus the second and main phase of the synchronizer involves delivering this "safety" information. This phase is thus responsible for the additional time and message requirements, denoted earlier by $C(\nu)$ and $T(\nu)$, for a synchronizer ν.

The complexities of a synchronous algorithm S are related to those of the asynchronous algorithm A resulting from combining S with a synchronizer ν by $C_A = C_S + T_S \cdot C(\nu)$ and $T_A = T_S \cdot T(\nu)$. (We ignore, for the purposes of the present discussion, any additional costs of an initialization phase that may be needed for setting up the synchronizer.)

Let us demonstrate these ideas by giving a brief description of the three synchronizers introduced in [A1]. Synchronizer α is the simplest. After the execution of a certain pulse, when a processor learns that it is safe, it simply reports this fact directly to all its neighbors. Thus the behavior and complexity of this synchronizer boil down to those of an algorithm in which every processor sends messages to every neighbor in every pulse, so $C(\alpha) = O(|E|)$ and $T(\alpha) = O(1)$.

For synchronizer β we assume the existence of a rooted spanning tree in the network. After the execution of a certain pulse, the safety information is collected "bottom-up" on the tree, by means of a communication pattern termed *convergecast*; whenever a processor learns that it and all its descendants in the tree are safe, it reports this fact to its parent. Eventually the root learns that all the processors in the network are safe, and then it broadcasts this fact along the tree, letting the processors start a new pulse. Since the process is carried out on the tree, the complexities of synchronizer β are $C(\beta) = O(|V|)$ and $T(\beta) = O(H)$, where H is the height of the tree. In the worst case $H = O(|V|)$ too.

The last synchronizer, γ, ahieves a certain trade-off between the previous ones. For synchronizer γ we assume that the network is partitioned into *clusters*. The partition is defined by a rooted spanning forest of the graph, where each tree in the forest defines a cluster. Between any two neighboring clusters there is one designated *preferred link* that serves for communication between them.

The safety information is handled by synchronizer γ in three steps. In the first step, each cluster separately applies the synchronizer β. By the end of this step, every processor knows that its cluster is safe (i.e., every processor in the cluster is safe). In the second step, every processor incident to a preferred link sends a message to the other cluster, saying that its cluster is safe. Finally, the third step is a repetition of the first, except the convergecast performed in each cluster carries a different information: whenever a processor learns that all the clusters neighboring it or any of its descendants are safe, it reports this fact to its parent. Again, when the root learns that all the neighboring clusters are safe, it broadcasts this fact along the tree, letting the processors of the cluster start a new pulse.

The complexities of synchronizer γ are $C(\gamma) = O(E_p)$ and $T(\gamma) = O(H_p)$, where E_p is the number of all tree links and all preferred links in a partition P, and H_p is the maximum height of a tree in the forest of P. Thus our goal is to find partitions with small values of E_p and H_p.

Let us now describe a simple partition for product graphs, and state the resulting complexities of a synchronizer γ based on this partition. Partition a product graph $G_1 \times G_2$ by defining each copy of G_2 as a cluster. Let D_2 denote the diameter of G_2. Then G_2 has a spanning tree of depth D_2. This spanning tree can be used to define the whole forest, as all the clusters have identical structure. Between every two clusters there are $|V_2|$ edges, but only one of them need be chosen as a preferred link. Thus an overall of $|E_1|$ preferred links are used. The number of edges used in the trees is $|V_1|(|V_2|-1)$. Thus, the time and communication complexities of the resulting synchronizer of type γ are $T = O(D_2)$ and $C = O(|E_1| + |V_1| \cdot |V_2|)$, respectively. (Alternatively, noting the symmetric role of G_1 and G_2 in the definition of the Cartesian product, we may reverse our view of the two graphs and obtain another partition, yielding the complexities $T = O(D_1)$ and $C = O(|E_2| + |V_1| \cdot |V_2|)$.)

The construction described above immediately suggests a way to handle the hypercube. For any chosen k, $0 \leqq k \leqq n$, view H_n as the product $H_n = H_{n-k} \times H_k$ and partition H_n as described earlier. This yields a synchronizer of type γ for H_n with time and communication complexities $T = O(k)$ and $C = O((n-k)2^{n-k} + 2^n)$, respectively. In particular, selecting $k = \lfloor \log n \rfloor$, we get $T = O(\log \log |V_n|)$ and $C = O(|V_n|)$. This complexity is already better than that of a synchronizer γ based on any partition obtained by using the partitioning algorithm of [A1]. However, it is still not optimal. To achieve an optimal synchronizer we need to develop a new technique, discussed in the next section.

4. Synchronizers and t-spanners. In this section we prove the two parts of Theorem 1. This amounts to showing two complementary relationships between t-spanners and synchronizers. On the one hand, the nonexistence of a t-spanner of a certain size implies a lower bound on the complexities of any synchronizer for the network. On the other hand, the existence of a t-spanner can be used for constructing a synchronizer of a new type, δ.

THEOREM 1 (Part (1)). *If the network G has no t-spanner with at most m edges, then every synchronizer ν for G requires either $T(\nu) \geqq t+1$ or $C(\nu) \geqq m+1$.*

Proof. Let us assume that the network $G = (V, E)$ has no t-spanner of m edges, and yet it has a synchronizer ν with $C(\nu) \leqq m$, i.e., using m or fewer messages per pulse. The argument is similar to that of the lower bounds of [A1], [A2]. The requirement from the synchronizer is to ensure that a processor does not produce a new pulse before it gets all the messages sent to it in the previous pulse. Thus, between every two consecutive pulses there must be some transfer of information between each pair of neighbors in the network. Otherwise, the completely asynchronous nature of the network will force these neighbors to wait forever for a message that may still be on its way. Consider the set E' of edges through which the messages of the synchronizer were sent. The information flow between every pair of neighbors in G has to go through the edges of E'. Since only m or fewer messages were sent, the number of these edges is at most m. Therefore by hypothesis, the subgraph $G' = (V, E')$ is not a t-spanner. This implies that there is an edge $(u, v) \in E$ such that the distance between u and v in G' is at least $t+1$. Thus, the information flow between u and v may require $t+1$ time units, so the time complexity of the synchronizer ν is at least $t+1$. □

THEOREM 1 (Part (2)). *If the network G has a t-spanner with m edges, then it has a synchronizer δ with $T(\delta) = O(t)$ and $C(\delta) = O(tm)$.*

Proof. Assume that the network $G=(V, E)$ has a t-spanner $G'=(V, E')$ of m edges. The synchronizer δ for G is constructed as follows. The safety information is transmitted over the spanner in rounds which may be viewed as "subpulses." When a processor v learns that it is safe, it sets a counter c to 0 and sends the message "safe" to all its neighbors *in the spanner.* Then it waits to hear a similar message from all these neighbors. Upon receiving a "safe" message from all neighbors (again—in the spanner), it increases its counter and repeats the process (i.e., it sends the message "safe" again, and then waits for similar messages and so on). This is done for t rounds. When $c=t$, the processor v may generate its next pulse. □

LEMMA 4.1. *When v holds $c=i$, every processor u at distance i or less from v in G' is safe.*

Proof. By induction on i. For $i=0$ the claim is immediate, as v reaches this stage of the synchronizer only after it is safe itself. Now consider the time when v increases c to $i+1$. This is done after v received $i+1$ "safe" messages from every neighbor in G'. These neighbors each sent the $(i+1)$st message only after having $c=i$. Thus, by the inductive hypothesis, for every such neighbor u, every processor w at distance i or less from u in G' is safe. Thus every processor w at distance $i+1$ or less from v in G' is safe too. □

COROLLARY 4.2. *When v holds $c=t$, every neighbor of v in G is safe.*

Proof. By the lemma, when v holds $c=t$, every processor u at distance i from v in G' is safe. By the definition of t-spanners, every neighbor of v in G is at distance t or less from v in G'. Thus every such neighbor is safe. □

It is clear that the time delay of synchronizer δ is at most $O(t)$, and the number of additional messages is $O(mt)$. Thus the proof is complete. □

The synchronizers of [A1] do not use spanners explicitly, and their formulations are quite different from our synchronizer δ. Nevertheless it is interesting to note that their underlying structure is strongly related to the notion of spanners. For instance, synchronizer α is based essentially on the fact that every graph is its own 1-spanner, and the construction algorithm of [A1] for synchronizer γ establishes the fact that every graph has an $O(\log_k |V|)$-spanner with $O(k|V|)$ edges, where $1<k<|V|$ is a parameter.

5. An optimal synchronizer δ for the hypercube. In this section we show how to construct a 3-spanner for the hypercube H_n with a linear $[O(2^n)]$ number of edges. This in turn implies an optimal synchronizer δ and thus proves our Theorem 2.

A *dominating set* for a graph G is a subset U of vertices with the property that for every vertex v of G, U contains either v itself or some neighbor of v.

To construct dominating sets, we make use of the notion of a Hamming code, a well-known idea from coding theory (cf. [H]). Since the Hamming code is so central to what we do, we review its definition and properties. The reader familiar with Hamming codes can skip directly to Lemma 5.1.

A string of 0's and 1's is a *word.* The *product* of two code words of the same length is their bitwise logical "and;" their *sum* is their bitwise exclusive "or." For example, the product of 0011 and 0101 is 0001, and their sum is 0110. The *weight* of a code word is the number of 1's in the word.

For any integer r, the Hamming code $HC(r)$ is defined by making reference to a matrix of 0's and 1's with r rows and 2^r-1 columns. The columns are all the binary integers from 1 up to 2^r-1. For example, here is the matrix used to define $HC(3)$:

$$M_3=\begin{bmatrix}0&0&0&1&1&1&1\\0&1&1&0&0&1&1\\1&0&1&0&1&0&1\end{bmatrix}.$$

The *code words* in $HC(r)$ are all those words of length 2^r-1 whose product with each of the rows of the defining matrix M_r has even weight. For example, 0000000 and 0101010 are code words of $HC(3)$. Word 0000000 has a product with each row of M_r that has weight 0, while 0101010 has products of weight 2, 2, and 0, respectively.

The *distance* between two words is the weight of their sum. Put another way, the distance is the number of positions in which the two words differ. For example, each pair of rows of M_r has distance 4.

An important and easily proved fact about Hamming codes is that the sum of two code words is also a code word. As a consequence, the Hamming codes have minimum distance 3; that is, no two code words have distance less than 3. In proof, let u and v be code words, and let w be their sum, which is also a code word. If the weight of w were 1, then since the product of w with every row of M_r has even weight, those products would have to have weight 0; i.e., the one column in which w has a 1 would have all 0's in M_r. Since there is no such column, w cannot have weight 1. Similarly, if w had weight 2, we could argue that two columns of M_r would have to be identical, which is not the case.

Now, let the *neighborhood* of a code word be that word plus all words of distance 1. Then in $HC(r)$, all neighborhoods have 2^r members. No word can be in the neighborhood of two code words, or else those code words would be of distance 2.

The last fact we need about Hamming codes is that $HC(r)$ has exactly 2^{2^r-1-r} members. The argument is that the rows of M_r can be thought of as independent vectors in the vector space of 2^r-1 dimensions over the field of two elements. $HC(r)$ is thus the null space of the vector space (of dimension r) whose basis is the rows of M_r, and therefore $HC(r)$ is of dimension 2^r-1-r.

The number of words in the neighborhood of some code word of $HC(r)$ is thus $2^r \times 2^{2^r-1-r}$, or 2^{2^r-1}. This is exactly the number of binary words of length 2^r-1. Thus, every binary word of length 2^r-1 is in the neighborhood of exactly one code word of $HC(r)$. Put another way, $HC(r)$ is a dominating set for the hypercube H_{2^r-1}.

LEMMA 5.1. *For every $n \geqq 1$, the n-cube has a dominating set of at most $2^{n+1}/n$ nodes.*

Proof. Let us first consider the case of $n = 2^r-1$ for some $r \geqq 1$. Let U be the Hamming code $HC(r)$. As we have argued above, U is a dominating set for the n-cube, and $|U| = 2^n/(n+1) \leqq 2^{n+1}/n$.

Now consider an arbitrary $n \geqq 1$. Let r be the integer satisfying $2^r-1 \leqq n < 2^{r+1}-1$ and let $d = 2^r-1$. Note that $n/2 \leqq d$. Let U_d be a dominating set for the d-cube. View the n-cube as the Cartesian product $H_n = H_{n-d} \times H_d$, and let $U = \{(x, y) \mid x \in V_{n-d}, y \in U_d\}$, where V_{n-d} is the set of vertices of H_{n-d}. Clearly U is a dominating set for the n-cube, and its size is

$$2^{n-d}|U_d| = 2^{n-d}\frac{2^d}{d+1} = \frac{2^n}{d+1} \leqq \frac{2^{n+1}}{n}. \qquad \square$$

We finally construct a spanner for the n-cube. The cases $n = 1, 2$ can easily be verified directly, so assume $n \geqq 3$. Our construction uses dominating sets for some of the subcubes of our hypercube. Consider an n-cube H_n, and let $p = \lfloor n/2 \rfloor$ and $q = \lceil n/2 \rceil$. View H_n as the Cartesian product $H_p \times H_q$. Let U_1 and U_2 be minimum-size dominating sets for H_p and H_q, respectively.

We define the subgraph $G' = (V_n, E')$ by choosing the following sets of edges:

(1) Every edge $((x, y), (x, y'))$ subject to $(y, y') \in E_q$ and $y' \in U_2$.
(2) Every edge $((x, y), (x', y))$ subject to $(x, x') \in E_p$ and $x' \in U_1$.
(3) Every edge $((x, y), (x, y'))$ subject to $(y, y') \in E_q$ and $x \in U_1$.
(4) Every edge $((x, y), (x', y))$ subject to $(x, x') \in E_p$ and $y \in U_2$.

LEMMA 5.2. *G' has fewer than $7 \cdot 2^n$ edges.*

Proof. By the previous lemma $|U_1| \leqq 2^{p+1}/p$ and $|U_2| \leqq 2^{q+1}/q$. Therefore the number of edges $(y, y') \in E_q$ with $y' \in U_2$ is at most $q|U_2| \leqq 2^{q+1}$, and so the number of edges of the first type is at most $2^p 2^{q+1} = 2^{n+1}$. Similarly the number of edges of the second type is at most 2^{n+1}. For the third type we get the bound $|U_1| \cdot |E_q| \leqq (2^{p+1}/p) \cdot q \cdot 2^{q-1} \leqq (q/p) \cdot 2^n$, and similarly of the fourth type there are at most $|U_2| \cdot |E_p| \leqq (p/q) \cdot 2^n$ edges. (This simple counting ignores multiple occurrences of certain edges in the various types, which in fact implies a slightly smaller bound.) Since $n \geqq 3$, $p/q + q/p < 3$, so overall we get a bound of fewer than $7 \cdot 2^n$ edges ($6 \cdot 2^n$ for even n). □

LEMMA 5.3. *G' is a 3-spanner of the n-cube.*

Proof. Consider two neighboring vertices u, v in H_n. Let $u = (x, y)$ where $x \in \{0, 1\}^p$. Then either $v = (x, y')$ for some neighbor y' of y in H_q or $v = (x', y)$ for some neighbor x' of x in H_p. Suppose the first case holds. If $x \in U_1$ then G' contains the edge (u, v) itself (type 3). Otherwise, G' contains the following edges:

(1) $e_1 = ((x, y), (x', y))$ for some $x' \in U_1$ (type (2)).

(2) $e_2 = ((x', y), (x', y'))$ (type (3)).

(3) $e_3 = ((x', y'), (x, y'))$ (type (2)).

These three edges constitute a path of length 3 in G' connecting u and v. The second case is handled similarly, using edges of types (1) and (4). □

COROLLARY 5.4. *For every $n \geqq 0$ the n-cube has a 3-spanner of fewer than $7 \cdot 2^n$ edges.* □

Finally, using Theorem 1 together with the last corollary, we have Theorem 2, which we restate below.

THEOREM 2. *For every $n \geqq 0$, the n-cube has a synchronizer of type δ with optimal time and communication complexities $T(\delta) = O(1)$ and $C(\delta) = O(2^n)$.* □

We note that the relatively large constant involved in the construction implies that the new synchronizer improves on the simple synchronizer α only for large n. A more careful analysis of Lemma 5.2 reveals improvements for some values of n beginning around $n \geqq 18$.

Acknowledgments. We wish to thank Alex Schäffer for helpful discussions.

REFERENCES

[A1] B. AWERBUCH, *Complexity of network synchronization*, J. Assoc. Comput. Mach., 32 (1985), pp. 804–823.

[A2] ———, *Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization*, Networks, 15 (1985), pp. 425–437.

[A3] ———, *Communication-time trade-offs in network synchronization*, in Proc. 4th ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, 1985, pp. 272–276.

[GHS] R. G. GALLAGER, P. A. HUMBLET, AND P. M. SPIRA, *A distributed algorithm for minimum weight spanning trees*, ACM Trans. Program. Languages Systems, 5 (1983), pp. 66–77.

[H] R. HILL, *A First Course in Coding Theory*, Oxford Applied Mathematics and Computing Science Press, Oxford, 1986.

[P] N. C. PEASE, *The indirect binary n-cube microprocessor array*, IEEE Trans. Comput., 6 (1977), pp. 458–473.

[PS] D. PELEG AND A. SCHÄFFER, *Graph spanners*, Res. Report RJ 6171, IBM Almaden, San Jose, CA, April 1988; J. Graph Theory to appear.

[PU] D. PELEG AND E. UPFAL, *A tradeoff between space and efficiency for routing tables*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 43–52.

[U] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, 1984.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 748–758, August 1989

008

SPACE-TIME TRADE-OFFS FOR ORTHOGONAL RANGE QUERIES*

PRAVIN M. VAIDYA†

Abstract. This paper investigates the question of (storage) space-(retrieval) time trade-off for orthogonal range queries on a static data base. Each record in the data base consists of a key that is a d-tuple of integers, and a data value that is an element in a commutative semigroup G. An orthogonal range query is specified by a d-dimensional parallelepiped (box). Two types of response to such a query are considered: one where the output is the semigroup sum of the data values whose keys are located in the query parallelepiped, and the other where the output is a list of all the records whose keys lie in the query parallelepiped. This paper studies two models, the arithmetic model and the tree model and obtains lower bounds on the product of retrieval time and storage space in both models.

Key words. range queries, space-time trade-offs, lower bounds

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U05

1. Introduction.

Consider a data base that contains a collection of records, each with a key and a number of data fields. Given a range query, which is specified by a set of constraints on the keys, the data base system is expected to return the set of records, or a function of the set of records whose keys satisfy all the constraints. If the data base is static the collection of records may be preprocessed to achieve a balance between the storage utilized and the time required to answer a query. There is an extensive literature [1], [2], [4], [8], [9], [11], [12] on algorithms for range query, and the space and time requirements have traditionally been used as performance measures for such algorithms. In this paper, we investigate the question of (storage) space-(retrieval) time trade-off for orthogonal range queries on a static data base.

Let G be a commutative semigroup with an addition operation $+$. Let d be a fixed positive integer. Let $N=\{1, 2, \cdots, n\}$ and let N^d denote the set of all d-tuples of positive integers less than or equal to n. A record $(k, f(k))$ is a pair of key $k \in N^d$ and datum $f(k) \in G$. The data base consists of n such records. Let $k=(k_1, k_2, \cdots, k_d)$. An orthogonal range query is specified by a $2d$-tuple $(x_{11}, x_{12}, x_{21}, x_{22}, \cdots, x_{d1}, x_{d2})$ of positive integers satisfying $x_{i1} < x_{i2}$, $1 \leqq i \leqq d$. Alternately, the query region for an orthogonal range query is a parallelepiped (box) b, defined by the product $[x_{11}, x_{12}) \times [x_{21}, x_{22}) \times \cdots \times [x_{d1}, x_{d2})$ of d-semiclosed intervals with positive integer endpoints. A key k is said to be located in a box $b=[x_{11}, x_{12}) \times [x_{21}, x_{22}) \times \cdots \times [x_{d1}, x_{d2})$ if and only if $x_{i1} \leqq k_i < x_{i2}$, $1 \leqq i \leqq d$. We consider two types of response to such a query, one where the output is the sum of the data $f(k)$ whose keys k are located in the query parallelepiped (box) b, and the other where the output is a list of all the records whose keys lie in the query parallelepiped b.

Let $Q(b)$ denote the input $2d$-tuple corresponding to query box b, and let K denote the set of keys in the data base. As we shall be studying space-time requirements for orthogonal range query only, we shall assume that the set of query regions is fixed to be the set of boxes.

A space-time trade-off seeks to answer questions such as what is the minimum amount of storage needed to ensure a certain query time? The trade-off between space

* Received by the editors August 18, 1986; accepted for publication (in revised form) November 6, 1988. This research was supported by a Shell Foundation fellowship. A preliminary version of this paper appeared in the Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 1985 [10].

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. Present address, AT & T Bell Laboratories, Murray Hill, New Jersey 07974.

and time is dependent on the model of data structure and the set of records in the data base. We fix the data structure model and then try to obtain a set of records that makes this trade-off as bad as possible. Thus the lower bounds on space-time products are to be interpreted as worst-case bounds, i.e., there exists a set of n records whose space-time product has the said bounds.

We study two models. In Model A, we work in the general framework defined by Fredman [5]-[7], and consider only data structures and manipulation algorithms that are independent of the choice of the semigroup G. So the set K of keys in the data base together with the set of query regions completely specifies the problem. Given a query box b, the query answering algorithm is expected to return the semigroup sum of the data values whose keys are located in b. Model A is an arithmetic model with unit cost for each arithmetic operation but no cost for memory retrieval. In this model, we show that for orthogonal range query on a static database with n records, there is a space-time trade-off $(\log T)^{d-1}TS \geqq \Omega(n(\log n)^{d-\theta})$, where $\theta = 1$ for $d = 2$, and $\theta = 2$ for $d \geqq 3$. Space-time trade-offs for circular range query and interval query in this model are studied by Yao in [13] and [14]. We note that for $d = 2$ the results of Yao [14] are considerably stronger for this model; specifically he shows that for a restricted type of range query $T = \Omega(\log n/(\log(S/n) + \log\log n))$ for $d = 2$. The complexity of dynamic range queries in this model is discussed by Fredman in [5]-[7].

In Model B (tree model), we study a broad class of tree data structures. In this model, a data structure is a rooted tree, and with each edge in the tree is associated a condition. Given a query, the query answering algorithm starts with the root, and visits a vertex v if and only if the given query satisfies the conjunction of the conditions on the path from the root to v. The output corresponding to a given query is a function of the data associated with the visited vertices. Several standard data structures, such as linked lists, range trees, etc. [1], [2], [4], [8], [9], [11], [12], fit into this model. In Model B, we investigate the orthogonal reporting problem where the response to a query is a list of all the records in the data base whose keys are located in the query box. Since the output size is query dependent, the time required to answer a query is not the correct measure of the overhead involved in producing the desired response to the query. So we define a scaled query time T' that measures the overhead for producing one unit of output. For the orthogonal reporting problem on a static data base with n records, we show that there is a space-time trade-off $(\log T' + \log\log n)^{d-1}T'S \geqq \Omega(n(\log n)^{d-\theta})$, where $\theta = 1$ for $d = 2$, and $\theta = 2$ for $d \geqq 3$.

The results in this paper for Model A (arithmetic model) have been significantly strengthened by Chazelle using a different technique [3].

2. An overview.

2.1. Model A. Arithmetic model. In this model [5], [6], [7], a data structure is an infinite array Z of variables $z_0, z_1, z_2, \cdots$, that stores elements from the commutative semigroup G. Given any input query, the query answering algorithm chooses a collection of at most T variables in the array and returns their semigroup sum as the response to the query. Since only arithmetic operations are charged, the query time is T. The data structure is assumed to be independent of the specific semigroup G and so the mapping between elements in G and variables in Z is determined solely by the set K of n keys in the data base. With each variable z_i in Z is associated a subset h_i of K, and the data value $\sum_{k \in h_i} f(k)$ is stored in z_i, where $f(k)$ is the data value associated with k. Let $H \subseteq 2^K$ such that every set in H is associated with some variable in Z and every variable in Z is associated with some set in H. Let R be the set of all possible query boxes, and let $P(K, H, T)$ be the property that for each $b \in R$, $b \cap K$ is expressible

as the disjoint union of at most T sets in H. (The lower bounds in this paper are valid even if disjoint union is replaced by union in the definition of $P(K, H, T)$.) The query answering algorithm works correctly if and only if $P(K, H, T)$ is satisfied. The storage space S is defined by

$$S = \max_{K} \min_{H \text{ satisfying } P(K,H,T)} |H|.$$

The following theorem summarizes the results in this model.

THEOREM 1. *In Model* A, *for orthogonal range query on a static database with n records, there is a space-time trade-off* $(\log T)^{d-1} TS \geq \Omega(n(\log n)^{d-\theta})$, *where* $\theta = 1$ *for* $d = 2$, *and* $\theta = 2$ *for* $d \geq 3$.

The proof is based on Lemma 1 given below. The lemma asserts that there exists a set K of n keys and a large enough set $B(T, n)$ of query parallelepipeds such that the subsets of K induced by members of $B(T, n)$ satisfy certain intersection conditions. The proof of Lemma 1 is given in § 4.

LEMMA 1. *There is a set K of n keys and a set $B(T, n)$ of boxes satisfying the following properties*:

(1) $(\log T)^{d-1} T|B(T, n)| = \Omega(n(\log n)^{d-\theta})$, *where* $\theta = 1$ *for* $d = 2$, *and* $\theta = 2$ *for* $d \geq 3$.

(2) *For distinct* b_1, b_2, *in* $B(T, n)$, $|b_1 \cap b_2 \cap K| < 1/T \min\{|b_1 \cap K|, |b_2 \cap K|\}$.

Using property (2) in Lemma 1, we show that for any H satisfying $P(K, H, T)$, we must have $|H| \geq |B(T, n)|$. Then Theorem 1 follows from the lower bound on $|B(T, n)|$ given by property (1) in Lemma 1. Let b_1, b_2, be distinct boxes in $B(T, n)$. As $b_1 \cap K$ is expressible as the union of at most T sets in H, there exists $h_1 \in H$ such that $|h_1| \geq (1/T)|b_1 \cap K|$ and $h_1 \subseteq (b_1 \cap K)$. Since $|(b_1 \cap K) \cap (b_2 \cap K)| < (1/T)|b_1 \cap K|$, h_1 cannot appear in the decomposition of $b_2 \cap K$ as the union of members of H. So with each b_i in $B(T, n)$ we can associate a distinct h_i in H.

2.2. Model B. Tree model. In the case of the reporting problem the output size is dependent on the given query, and so the arithmetic model is not suitable for investigating this problem. So we study the tree model for data structures. In this model, the data structure is assumed to be a rooted tree. With each vertex v is associated a set of data items and we let $data(v)$ denote the set of data items associated with vertex v. With each edge in the tree is associated a condition. Given an input query in the form of a tuple of numbers, the query answering algorithm first visits the root. A vertex v is visited if and only if it is a son of some vertex u that has already been visited and the input tuple satisfies the condition associated with edge uv. We define $cond(v)$ to be the conjunction of all the conditions on the path from the root to vertex v. Thus for any query box b, on being given the corresponding tuple $Q(b)$ as input, the query answering algorithm visits vertex v if and only if $Q(b)$ satisfies $cond(v)$. The response to a query is a function of the data at the visited vertices.

In the tree model, we investigate the orthogonal reporting problem where the response to a given query is a list of all the records in the data base whose keys lie in the query box. Let $\hat{G}$ be a semigroup consisting of a single element. We shall restrict the universe of records so that the data in each record is the unique element from $\hat{G}$. We shall only consider sets of records which are such that no two records in a set have the same key. Then the set K of keys completely specifies the set of records, and the orthogonal reporting problem is to produce a list of all the keys in K that lie in the given query box. Note that considering this special case does not cause any loss of generality as the lower bounds obtained in the special case trivially extend to the general case. Let r be a fixed constant. We restrict ourselves to trees where every vertex

has degree at most r. The condition associated with each edge in the tree is restricted to be a disjunction of at most r binary comparisons. Thus $cond(v)$, the conjunction of the conditions on the path from the root to v, is now a conjunction of disjunctions of comparisons. For each vertex v, $data(v)$ is a set of keys. Vertices may share storage, so $data(v)$ is effectively the set of records accessed via vertex v.

Consider a fixed set K of keys, and a fixed tree for K. For a query box b, let $U(b)$ be the set of all those vertices v such that $Q(b)$ satisfies $cond(v)$. Given a query box b, the query answering algorithm visits all the vertices in $U(b)$, and extracts the set of keys $\bigcup_{v \in U(b)} data(v)$. The set of keys $b \cap K$ is then obtained by explicitly testing for each key in $\bigcup_{v \in U(b)} data(v)$ whether the key is located in b. Thus filtering search [2] is included in this model. Let $T(b)$ be the time required to answer the query corresponding to b. $T(b)$ is lower bounded by $|U(b)| + |\bigcup_{v \in U(b)} data(v)|$. Since the output size is query dependent, the time required to answer a query is not the correct measure of the overhead involved in producing the desired response to the query. With respect to a fixed set K of keys, and a fixed tree for K, we define a scaled query time T' as follows:

$$T' = \max_{1 \leqq |b \cap K| \leqq \log_2 n} \frac{T(b)}{|b \cap K|}.$$

For a fixed set of keys, the storage S is defined to be the minimum number of vertices a corresponding tree must have to ensure a scaled query time of T'.

At this point we remark that several common data structures, such as linked lists, range trees, etc. [1], [2], [4], [8], [9], [11], [12], fit into the tree model. Also note that the tree model restricts the manner in which data records are accessed; it does not place a restriction on how the data is stored. As long as there is a fixed tree that defines how the data in the data structure is accessed, and a node in this tree corresponds to a distinct unit of storage in the data structure, the data structure would still fit into the tree model; it would not matter that the data structure itself was not a tree.

THEOREM 2. *In the tree model, for the orthogonal reporting problem on a static data base with n records, there is a space-time trade-off* $(\log T' + \log\log n)^{d-1} T' S \geqq \Omega(n(\log n)^{d-\theta})$, *where* $\theta = 1$ *for* $d = 2$, *and* $\theta = 2$ *for* $d \geqq 3$.

The proof of Theorem 2 is based on Lemma 2 below. A proof of Lemma 2 is given in § 4.

LEMMA 2. *Let c_1 and c_2 be constants dependent on the dimension d. There exists a set of K of n keys that has a subset K' satisfying the following properties*:

(1) $|K'| \geqq c_1 n$.

(2) *With respect to a particular tree for K, let $V(k)$ be the set of all those vertices v that satisfy the conditions* (i) *key* $k \in data(v)$; *and* (ii) *there is a box b such that $Q(b)$ satisfies $cond(v)$ and* $1 \leqq |b \cap K| \leqq \kappa(d, n)$, *where* $\kappa(d, n) = 1$ *for* $d = 2$ *and* $\kappa(d, n) = \log_2 n$ *for* $d \geqq 3$. *Then for each tree for K, for each key* $k \in K'$, $|V(k)| \geqq (\log_2 n)^{d-1} / c_2 (\log_2 T' + \log_2 \log_2 n)^{d-1}$.

Let K be a set of n keys and K' be a subset of K such that K and K' satisfy the conditions in Lemma 2. Consider a fixed tree for K. For a key k, let $V(k)$ be as defined in Lemma 2. We must have that

$$\sum_{k \in K'} |V(k)| \leqq \sum_{v \in \bigcup_{k \in K'} V(k)} |data(v)|.$$

For each vertex v in $\bigcup_{k \in K'} v(k)$, $|data(v)| \leqq \kappa(d, n) T'$, since v is visited by a query corresponding to a box containing at most $\kappa(d, n)$ keys. Then from properties (1) and

(2) in Lemma 2 it follows that

$$\left| \bigcup_{k \in K'} V(k) \right| \geq \frac{c_1 n (\log_2 n)^{d-1}}{c_2 \kappa(d, n) T' (\log_2 T' + \log_2 \log_2 n)^{d-1}}.$$

Thus the storage S for a set K of n keys satisfying the conditions in Lemma 2 must obey the following constraint:

$$S \geq \frac{c_1 n (\log_2 n)^{d-1}}{c_2 \kappa(d, n) T' (\log_2 T' + \log_2 \log_2 n)^{d-1}}.$$

3. Canonical parallelepipeds and almost uniform distributions. We shall utilize a special class of parallelepipeds (boxes) in obtaining the desired space-time trade-offs. Let n be a power of 2 and let $I_l = \{[j2^l + 1, (j+1)2^l + 1): 0 \leq j < (n/2^l)\}$. I_l is the set of intervals obtained by breaking up $[1, n+1)$ into $n/2^l$ semiclosed intervals of equal size, each interval being closed on the left and open on the right. Let $I = I_0 \cup I_1 \cup \cdots \cup I_{\log_2 n}$. Then I is defined to be the set of canonical intervals, and I^d is defined to be the set of canonical parallelepipeds, or equivalently canonical boxes.

For a box b, we use $[\pi_{i1}(b), \pi_{i2}(b))$ to denote the interval that is the projection of box b onto to the ith coordinate axis. Equivalently, for $1 \leq i \leq d$, $\pi_{i1}(b)$ and $\pi_{i2}(b)$ denote the $(2i-1)$st and the $(2i)$th components of the $2d$-tuple $Q(b)$ corresponding to box b. For a box b, *dimensions*(b) is defined to be the d-tuple $((\pi_{12}(b) - \pi_{11}(b)), (\pi_{22}(b) - \pi_{21}(b)), \cdots, (\pi_{d2}(b) - \pi_{d1}(b)))$. We note that since I contains intervals of $\log_2 n + 1$ distinct lengths, the total number of choices possible for the dimensions of a canonical box is $(\log_2 n + 1)^d$. Let *vol*(b) denote the volume of a box b, and let $p(x) = 2^{\lceil \log_2 x \rceil}$. Let J be the canonical parallelepiped $J_0 \times J_1 \times \cdots \times J_{d-1}$ where $J_i = [2i(2p(d))^{-1} n + 1, (2i+1)(2p(d))^{-1} n + 1)$, for $0 \leq i \leq d-1$. The following lemmas list the properties of canonical boxes that we shall require.

LEMMA 3. *The number of canonical boxes of identical dimensions and of volume* 2^i *is* $n^d / 2^i$.

LEMMA 4. *Let* $0 \leq i \leq (\log_2 n / d^2)$. *Then the number of possibilities for the dimensions of a canonical box of volume* $2^i n^{d-1}$ *is* $\Omega((\log_2 n / d^2)^{d-1})$, *and at most* $(\log_2 n + d)^{d-1}$.

Proof. The number of nonnegative integer solutions to

$$j_1 + j_2 + \cdots + j_d = i + (d-1) \log_2 n,$$

$$\text{subject to} \quad 0 \leq j_l \leq \log_2 n, \quad 1 \leq l \leq d, \quad i \leq (\log_2 n / d^2)$$

is at least $(\log_2 n / d^2)^{d-1}$ for large enough n, and at most $(\log_2 n + d)^{d-1}$ for any i in the desired range. That gives the required bound on the number of possibilities for the dimensions of a canonical box of volume $2^i n^{d-1}$. □

LEMMA 5. *Let* $b_1, b_2, \cdots, b_\rho$ *be canonical boxes of volume* α *such that* $\bigcap_{i=1}^{\rho} b_i \neq \emptyset$. *Then* $vol(\bigcap_{i=1}^{\rho} b_i) \leq \alpha 2^{1 - \rho^{1/(d-1)}}$.

Proof. For $1 \leq m \leq d$, let

$$L_m = \{[\pi_{m1}(b_j), \pi_{m2}(b_j)): 1 \leq j \leq \rho\}.$$

The intervals in L_m can be ordered by containment, and the ratio of the lengths of the largest and the smallest intervals in L_m is $2^{|L_m|-1}$. Let $|L_m^*| = \max_m |L_m|$. By pigeonholing, $|L_m^*| \geq \rho^{1/(d-1)}$. Then

$$vol\left(\bigcap_{i=1}^{\rho} b_i \right) \leq \alpha 2^{1 - |L_m^*|} \leq \alpha 2^{1 - \rho^{1/(d-1)}}.$$

□

LEMMA 6. *Let $\hat{b}$ be a fixed canonical box of volume α. Then the number of canonical boxes b of volume α which satisfy the condition $vol(b \cap \hat{b}) \geqq 2^{-j} vol(\hat{b})$ is at most $(2j+d+1)^{d-1}$.*

Proof. Since $vol(b \cap \hat{b}) \geqq 2^{-j} vol(\hat{b})$ and $vol(b) = vol(\hat{b})$, for each m, $1 \leqq m \leqq d$, either $[\pi_{m1}(b), \pi_{m2}(b))$ contains $[\pi_{m1}(\hat{b}), \pi_{m2}(\hat{b}))$ or $[\pi_{m1}(\hat{b}), \pi_{m2}(\hat{b}))$ contains $[\pi_{m1}(b), \pi_{m2}(b))$, and

$$2^{-j} \leqq \frac{\pi_{m2}(b) - \pi_{m1}(b)}{\pi_{m2}(\hat{b}) - \pi_{m1}(\hat{b})} \leqq 2^{j}.$$

Thus for $1 \leqq m \leqq d$, there are at most $(2j+1)$ possibilities for the mth interval defining box b, and as the volume of the boxes b is fixed to be α the total number of possibilities for the boxes b is bounded by $(2j+d+1)^{d-1}$. □

Having described canonical boxes, we shall proceed to almost uniform distributions of n keys. The distributions are termed almost uniform because the number of keys in a canonical box does not deviate too far from the volume of the box divided by n^{d-1}. For $d=2$, we can explicitly construct such distributions, and thereby get Theorem 3. For $d \geqq 3$, we have to resort to counting arguments and show that the number of distributions of n keys, which do not satisfy the properties in Theorem 4, is less than the total of n^{dn} possible distributions.

THEOREM 3. *For $d=2$, there is a set K of n keys such that for each canonical box b,*

$$\left\lfloor \frac{vol(b)}{n^{d-1}} \right\rfloor \leqq |b \cap K| \leqq \left\lceil \frac{vol(b)}{n^{d-1}} \right\rceil.$$

Proof. It is adequate to obtain a set K of n keys such that each canonical box of volume n contains exactly one key. We use an inductive construction. Let x_1 and x_2 denote the two attributes of a key. Let K_m denote a set of m keys satisfying the conditions in Theorem 3. We shall obtain K_{2m} from K_m. Let

$$K'_m = \{(2x_1 - 1, x_2) : (x_1, x_2) \in K_m\}$$

and

$$K''_m = \{(2x_1, x_2 + m) : (x_1, x_2) \in K_m\}.$$

We let $K_{2m} = K'_m \cup K''_m$. A canonical box of volume $2m$ and x_1 dimension equal to 1 contains exactly one key, as each key has a distinct value for x_1. Let $b = [x_{11}, x_{12}) \times [x_{21}, x_{22})$ be a canonical box of volume m corresponding to $n = m$, and let $b_1 = [2x_{11} - 1, 2x_{12} - 1) \times [x_{21}, x_{22})$ and $b_2 = [2x_{11} - 1, 2x_{12} - 1) \times [x_{21} + m, x_{22} + m)$. Then b_1 and b_2 are canonical boxes of volume $2m$ corresponding to $n = 2m$. If $(x_1, x_2) \in b \cap K_m$ then $(2x_1 - 1, x_2) \in b_1 \cap K'_m$ and $(2x_1, x_2 + m) \in b_2 \cap K''_m$, and b_1, b_2 do not contain any other key in K_{2m}. Furthermore, all canonical boxes corresponding to $n = 2m$, of volume $2m$ and x_1 dimension at least 2, may be derived in this manner from canonical boxes of volume m corresponding to $n = m$. □

THEOREM 4. *Let σn^{dn} be the number of distinct sets K of n keys, each key in N^d, that satisfy the three properties given below. Then σ tends to 1 as n tends to ∞ and $\sigma = (1 - o(1/n))$.*

(1) *Let $a(n) = 2p(\log_2 n)$. For each canonical box b,*

$$\left\lfloor \frac{vol(b)}{a(n) n^{d-1}} \right\rfloor \leqq |b \cap K| < 6a(n) \left\lceil \frac{vol(b)}{a(n) n^{d-1}} \right\rceil.$$

(2) *Each canonical box of volume n^{d-1} contains at most $\log_2 n$ keys.*

(3) $|J \cap K| \geqq (n/(4(2p(d))^{2d})) = n/4(c_3)^2$.

Proof. The total number of possible key distributions (sets) K is n^{dn}. Let F_1, F_2, and F_3, be the fraction of distributions that do not satisfy properties (1), (2), and (3), in Theorem 4, respectively. We shall show that each of F_1, F_2, and F_3, is $o(1/n)$.

To bound F_1 note that, if there is a canonical box whose volume is not equal to $a(n)n^{d-1}$ and that violates property (1) above, then there is necessarily a canonical box of volume $a(n)n^{d-1}$ that violates property (1) in the same manner. Let F_{11} be the fraction of distributions K such that there is a canonical box of volume $a(n)n^{d-1}$ that does not contain a key in K, and let F_{12} be the fraction of distributions K such that some canonical box of volume $a(n)n^{d-1}$ contains at least $6a(n)$ keys in K. Then $F_1 \leqq F_{11} + F_{12}$. A bound on F_{11} may be obtained by noting that there are at most $n(\log_2 n + d)^{d-1}$ choices for a canonical box of volume $a(n)n^{d-1}$ and all the keys in K must lie outside the chosen canonical box. Thus

$$\begin{aligned} F_{11} &\leqq n(\log_2 n + d)^{d-1}\left(1 - \frac{a(n)}{n}\right)^n \\ &\leqq O(n(\log n)^{d-1} e^{-a(n)}) \\ &= o\left(\frac{1}{n}\right). \end{aligned}$$

An upper bound on F_{12} is obtained by observing that we may choose a canonical box of volume $a(n)n^{d-1}$ in at most $n(\log_2 n + d)^{d-1}$ ways, and we may choose $6a(n)$ keys to lie in the chosen box and then let the remaining keys be located anywhere in $[1, n+1)^d$. Then

$$\begin{aligned} F_{12} &\leqq n(\log_2 n + d)^{d-1}\binom{n}{6a(n)}\left(\frac{a(n)}{n}\right)^{6a(n)} \\ &\leqq n(\log_2 n + d)^{d-1}\frac{(a(n))^{6a(n)}}{6a(n)!} \\ &\leqq n(\log_2 n + d)^{d-1}\left(\frac{e}{6}\right)^{6a(n)} \cdots \quad \text{(using Stirling's approximation)} \\ &= o\left(\frac{1}{n}\right). \end{aligned}$$

Thus

$$F_1 \leqq F_{11} + F_{12} = o(1/n) + o(1/n) = o(1/n).$$

A bound F_2 is obtained as follows. A canonical box of volume n^{d-1} may be chosen in at most $n(\log_2 n + d)^{d-1}$ ways; then $\log_2 n$ keys may be selected to lie in the chosen box, and the remaining keys may lie anywhere in $[1, n+1)^d$. Thus,

$$\begin{aligned} F_2 &\leqq n(\log_2 n + d)^{d-1}\binom{n}{\log_2 n} n^{-\log_2 n} \\ &\leqq \frac{n(\log_2 n + d)^{d-1}}{(\log_2 n)!} \\ &= o\left(\frac{1}{n}\right). \end{aligned}$$

To bound F_3, we choose $(1-4^{-1}c_3^{-2})n$ points to lie outside the canonical box J, and let the remaining points be located anywhere in $[1, n+1)^d$. Furthermore, the volume of J is n^d/c_3. Thus

$$F_3 \leqq \left(\frac{n}{(1-4^{-1}c_3^{-2})n}\right)\left(1-\frac{1}{c_3}\right)^{n(1-4^{-1}c_3^{-2})}.$$

Using Stirling's approximation for factorials and taking logarithms we get

$$\log_e F_3 \leqq \frac{n}{4c_3^2}\left(\log_e (4c_3^2)+(4c_3^2-1)\log_e\left(1-\frac{4c_3-1}{4c_3^2-1}\right)\right)+O(\log n).$$

Then noting that $\log_e (1-x) \leqq -x$ for $0 < x \leqq 1$, we get

$$\log_e F_3 \leqq \frac{n}{4c_3^2}(\log_e (4c_3^2)+1-4c_3)+O(\log n)$$

$$\leqq -\frac{7n}{4c_3^2}+O(\log n) \quad \text{as } c_3 \geqq 16.$$

Thus $F_3 = o(1/n)$. □

4. Proofs of lemmas. In this section we give proofs of Lemmas 1 and 2 used to prove Theorems 1 and 2 in §§ 2.1 and 2.2, respectively. For the purposes of this section we shall let the set K of keys be fixed. For $d=2$ let K be a fixed set of n keys as specified by Theorem 3, and for $d \geqq 3$ let K be a fixed set of n keys as specified by Theorem 4. We assume that n is a power of 2. Let $\beta(\alpha)$ denote the set of canonical boxes of volume α, and let β_J denote the set of those canonical boxes that are also subboxes of J (for definition of J see § 3).

Proof of Lemma 1. We shall give a proof for $d \geqq 3$, the proof for $d=2$ is similar. Let $\chi(T) = (64T(2p(d))^{2d})$. Let $B(T, n)$ be the largest set of boxes satisfying the following conditions:

(1) For all $b \in B(T, n)$, $b \in \beta(\chi(T)a(n)n^{d-1})$, and $|b \cap K| \geqq 6Ta(n)$.

(2) For any two boxes b_1 and b_2 in $B(T, n)$, $vol(b_1 \cap b_2) \leqq a(n)n^{d-1}$.

By Theorem 4, the number of keys in K located in a canonical box of volume $\chi(T)a(n)n^{d-1}$ is at most $6\chi(T)a(n)$. It is then easily shown that the number of boxes in $\beta(\chi(T)a(n)n^{d-1})$ that have identical dimensions and that contain at least $6Ta(n)$ keys is $\Omega(n/(T\log_2 n))$. Then from Lemmas 4 and 6 in § 3, we can conclude that $|B(T, n)| = \Omega(n/T\log_2 n(\log_2 n/\log_2 T)^{d-1})$. The intersection of any two distinct boxes b_1, b_2 in $B(T, n)$ is a canonical box of volume at most $(a(n)n^{d-1})$ and so by Theorem 4, $|b_1 \cap b_2 \cap K| < 6a(n) < (1/T)\min\{|b_1 \cap K|, |b_2 \cap K|\}$. □

In Model B, as we restrict ourselves to binary comparisons, the only possible comparisons are those between two components of the input tuple, and those between a component of the input tuple and a constant. We shall focus on canonical boxes that are subboxes of the canonical box J. For each box $b \subseteq J$, the input tuple $Q(b) = (x_{11}, x_{12}, x_{21}, x_{22}, \cdots, x_{d1}, x_{d2})$ is such that $x_{11} < x_{12} < x_{21} < x_{22} < \cdots < x_{d1} < x_{d2}$, and so a comparison between two components of the input tuple has the same outcome for each subbox b of J. Then, in Model B we need to analyze only comparisons between a component of the input tuple and a constant. We note that, if the input tuple satisfies a comparison between x_j, the (j)th component of the tuple, and a constant, when x_j takes on the value z_1 as well as when x_j takes on the value z_2, then the input tuple satisfies the comparison whenever x_j takes on any value between z_1 and z_2. The following lemma follows directly from these observations.

LEMMA 7. *Let b_1 and b_2 be subboxes of J. Let C be a conjunction of binary comparisons, and let both $Q(b_1)$ and $Q(b_2)$ satisfy C. Let b be a subbox of J such that either $\pi_{ij}(b_1) \leqq \pi_{ij}(b) \leqq \pi_{ij}(b_2)$ or $\pi_{ij}(b_2) \leqq \pi_{ij}(b) \leqq \pi_{ij}(b_1)$, for $1 \leqq i \leqq d$, $1 \leqq j \leqq 2$. Then the tuple $Q(b)$ also satisfies C.* □

LEMMA 8. *Let t be an integer greater than 1, let γ be an integer greater than 0, and let m be an integer such that $1 \leqq m \leqq d$. Let $C = \neg(C_1 \vee C_2 \vee \cdots \vee C_{t-1})$, where each of C_i, $1 \leqq i \leqq t-1$, is a conjunction of comparisons. Let $b_1, b_2, \cdots, b_{t+1}$ be distinct canonical boxes satisfying the following conditions:*

(1) *For all i, $1 \leqq i \leqq t+1$, $b_i \in (\beta_J \cap \beta(\alpha))$.*

(2) *Each of the tuples $Q(b_1), Q(b_2), \cdots, Q(b_{t+1})$ satisfies condition C.*

(3) *For all i, $1 \leqq i \leqq t+1$, $\pi_{m2}(b_i) - \pi_{m1}(b_i) = \pi_{m2}(b_1) - \pi_{m1}(b_1)$.*

(4) *For all i, $1 \leqq i \leqq t$, $\pi_{m1}(b_{i+1}) - \pi_{m2}(b_i) \geqq \gamma(\pi_{m2}(b_1) - \pi_{m1}(b_1))$.*

Then there are at least γ boxes b in $\beta_J \cap \beta(\alpha)$ such that $Q(b)$ satisfies condition C.

Proof. Let $\lambda([y_1, y_2), m, b)$ be the box obtained by replacing the mth interval defining box b by the interval $[y_1, y_2)$. Note that I is the set of canonical intervals. For $1 \leqq i \leqq t$, let

$$A_i = \{\lambda([y_1, y_2), m, b_i) : [y_1, y_2) \in I, y_2 - y_1 = \pi_{m2}(b_1) - \pi_{m1}(b_1), \pi_{m2}(b_i) \leqq y_1 < y_2 \leqq \pi_{m1}(b_{i+1})\}.$$

Then $|A_i| \geqq \gamma$, and each box in A_i is a subbox of J and a canonical box of volume α. There exists an i such that for each box b in A_i, $Q(b)$ satisfies C. Suppose this is not true. Then there exist boxes $\hat{b}_{i_1} \in A_{i_1}$, $\hat{b}_{i_2} \in A_{i_2}$, $1 \leqq i_1 < i_2 \leqq t$, such that both $Q(\hat{b}_{i_1})$ and $Q(\hat{b}_{i_2})$ satisfy C_l, for some l, $1 \leqq l \leqq t-1$. Then by Lemma 7 it follows that $Q(b_{i_2})$ must satisfy C_l and thereby not satisfy C which is a contradiction. □

Proof of Lemma 2. We shall give a proof for $d \geqq 3$, the proof for $d = 2$ may be constructed along similar lines. Fix a tree for the set of keys K. For the purposes of the proof we shall restrict ourselves to canonical boxes that are subboxes of J. Note that as K satisfies the conditions in Theorem 4 in § 3, a canonical box in $\beta(n^{d-1})$ contains at most $\log_2 n$ keys in K, and so the query time T for such a box cannot exceed $T' \log_2 n$. We shall show that if the conditions in Lemma 2 do not hold then there must be a canonical box $b \in \beta(n^{d-1})$ such that $Q(b)$ satisfies $cond(v)$ for more than $T' \log_2 n$ vertices v. Then the query time for b would exceed $T' \log_2 n$, and that would be a contradiction.

Let $|J \cap K| = c_5 n$, by Theorem 4 we know that such a c_5 exists. For each key $k \in J \cap K$, there are at least $c_4(\log_2 n)^{d-1}$ boxes in $\beta_J \cap \beta(n^{d-1})$ that contain k, for some constant c_4 dependent on d. Let $\delta = 2^d (T' \log_2 n + 1)^d (\log_2 n + 1)^{d+2}$, and let $\psi(T', n) = 2rc_5^{-1}(T')^2 \log_2 n + d)^{d+3}\delta$. Let c_6 be a large enough constant such that $(T' \log_2 n)^{c_6^{1/(d-1)}} \geqq 2^d (4\psi(T', n) + 6)^d$. The constants c_1 and c_2 in Lemma 2 are given by $c_1 = c_5/2$ and $c_2 = c_6/c_4$.

For each key $k \in K$, let

$$\tau(k) = \{b : b \in (\beta_J \cap \beta(n^{d-1})), k \in (b \cap K)\},$$

and let

$$\hat{V}(k) = \{v : k \in data(v), \exists b \in \tau(k) \quad \text{s.t.} \quad Q(b) \text{ satisfies } cond(v)\}.$$

We note that for each box $b \in \tau(k)$ there exists a vertex $v \in \hat{V}(k)$ such that $Q(b)$ satisfies $cond(v)$. For each $k \in J \cap K$, $|\tau(k)| \geqq c_4(\log_2 n)^{d-1}$, and $\hat{V}(k) \subseteq V(k)$, where $V(k)$ is the set of boxes defined in Lemma 2 in § 2.2.

Let K' be the largest subset of $J \cap K$ such that

$$\forall k \in K' \quad |\hat{V}(k)| \geq \frac{c_4(\log_2 n)^{d-1}}{c_6(\log_2 T' + \log_2 \log_2 n)^{d-1}}.$$

If $|K'| \geq c_5 n/2$ then the conditions in Lemma 2 are satisfied. So assume that $|(J \cap K) - K'| \geq c_5 n/2$. With each key k in $(J \cap K) - K'$ we can associate a distinguished vertex $\mu(k)$ and a distinguished set of canonical boxes $A(k)$ satisfying the following conditions:

(1) $|A(k)| \geq c_6(\log_2 T' + \log_2 \log_2 n)^{d-1}$.
(2) $A(k) \subseteq \tau(k)$.
(3) $\mu(k) \in \hat{V}(k)$.
(4) For all $b \in A(k)$, $Q(b)$ satisfies $cond(\mu(k))$.

The query time for a box in $\beta(n^{d-1})$ cannot exceed $T' \log_2 n$, and hence for all $k \in ((J \cap K) - K')$, $|data(\mu(k))| \leq T' \log_2 n$. Then $|\{\mu(k): k \in ((J \cap K) - K')\}| \geq c_5 n/2T' \log_2 n$. Let η be the set of all vertices v such that there are at least δ vertices in the set $\{\mu(k): k \in ((J \cap K) - K')\}$ that are also present in the subtree rooted at v. As the degree of each vertex in the tree is at most r (r a fixed constant), $|\eta| \geq c_5 n/2r\delta T' \log_2 n$.

For a vertex u, let $num(u)$ be the number of canonical boxes b such that $b \in \beta(n^{d-1})$ and $Q(b)$ satisfies $cond(u)$. Suppose we can show that for each vertex u in η, $num(u) \geq \psi(T', n)$. From the lower bound on the number of vertices in η it would follow that

$$\sum_{u \in \eta} num(u) \geq |\eta| \psi(T', n) > n(\log_2 n + d)^{d+1} T'.$$

Since $|\beta(n^{d-1})| \leq n(\log_2 n + d)^{d-1}$, we could then conclude that there is a $b \in \beta(n^{d-1})$ such that $Q(b)$ satisfies $cond(u)$ for at least $T' \log_2 n + 1$ vertices u in η, and that would be a contradiction.

Let u be an arbitrary vertex in η other than the root. We have to show that $num(u) \geq \psi(T', n)$. With u one can associate distinct keys $k_1, k_2, \cdots, k_\delta$ in $(J \cap K) - K'$, such that for each box b in $\bigcup_{i=1}^{\delta} A(k_i)$, $Q(b)$ satisfies $cond(u)$. Let $b(k_i) = \bigcap_{b \in A(k_i)} b$. Then for $1 \leq i \leq \delta$, $b(k_i) \in \beta_J$, and $b(k_i)$ contains key k_i. By Lemma 5, each of $b(k_i)$ has volume at most $n^{d-1}/(4\psi(T', n) + 6)^d$.

Among $b(k_1), b(k_2), \cdots, b(k_\delta)$ we can find $2^d(T' \log_2 n + 1)^d(\log_2 n + 1)$ distinct boxes of identical dimensions, say $b_1, b_2, \cdots, b_l$. This is possible since the number of possibilities for the dimensions of a canonical box is at most $(\log_2 n + 1)^d$, and by Theorem 4 each of $b(k_i)$ can contain at most $\log_2 n$ keys in K. Corresponding to $b_1, b_2, \cdots, b_l$, for $1 \leq m \leq d$, let

$$L_m = \{[\pi_{m1}(b_j), \pi_{m2}(b_j)): 1 \leq j \leq l\}.$$

All the intervals in L_m are of the same length, and any two intervals in L_m do not overlap. Let $t = T' \log_2 n$. Suppose that for some m, $1 \leq m \leq d$, there exist $t+1$ intervals $[\pi_{m1}(b_{j_1}), \pi_{m2}(b_{j_1})), [\pi_{m1}(b_{j_2}), \pi_{m2}(b_{j_2})), \cdots, [\pi_{m1}(b_{j_{t+1}}), \pi_{m2}(b_{j_{t+1}}))$, in L_m such that for $1 \leq q \leq t$, $\pi_{m1}(b_{j_{q+1}}) - \pi_{m2}(b_{j_q}) \geq \psi(T', n)(\pi_{m2}(b_{j_1}) - \pi_{m1}(b_{j_1}))$. To each of these $t+1$ intervals $[\pi_{m1}(b_{j_q}), \pi_{m2}(b_{j_q}))$ there corresponds a distinct box $\hat{b}_q$ such that $[\pi_{m1}(\hat{b}_q), \pi_{m2}(\hat{b}_q)) = [\pi_{m1}(b_{j_q}), \pi_{m2}(b_{j_q}))$, $\hat{b}_q \in (\beta_J \cap \beta(n^{d-1}))$, and $Q(\hat{b}_q)$ satisfies $cond(u)$. The pathlength from the root to u does not exceed $t-1$, and so we may apply Lemma 8 to these $t+1$ boxes and conclude that there are at least $\psi(T', n)$ boxes b in $\beta(n^{d-1})$ such that $Q(b)$ satisfies $cond(u)$.

We shall now show that there exists a required collection of $T' \log_2 n + 1$ intervals in some set L_m. Assume that such a collection of intervals does not exist. Then for each m, $1 \leqq m \leqq d$, all the intervals in the set L_m can be covered by $T' \log_2 n$ intervals, each of length at most $(2\psi(T', n)+3)(\pi_{m2}(b_1) - \pi_{m1}(b_1))$. Each of the $dT' \log_2 n$ covering intervals is closed on the left and open on the right, and has integer end points in the range 1 to $n+1$. It then follows that the boxes $b_1, b_2, \cdots, b_l$, are themselves contained in the union of $(T' \log_2 n)^d$ boxes, each of volume at most $(2\psi(T', n)+3)^d vol(b_1)$. As each of $b_1, b_2, \cdots, b_l$ contains a distinct key, and $l \geqq 2^d (T' \log_2 n + 1)^d (\log_2 n + 1)$, and $vol(b_1) \leqq (4\psi(T', n)+6)^{-d} n^{d-1}$, it follows that there is a box of volume at most n^{d-1} that contains at least $2^d(\log_2 n + 1)$ keys in K. This box of volume at most n^{d-1} can be covered by 2^d canonical boxes of volume n^{d-1}. Thus there must be a canonical box of volume n^{d-1} that contains at least $\log_2 n + 1$ keys in K, and since K is a set of keys satisfying the conditions in Theorem 4 this is not possible. □

5. Conclusion. We have obtained space-time trade-offs for orthogonal range query in two models, the arithmetic model and the tree model. Most data structures used in practice are rooted trees and so it may be worth studying more problems in the context of Model B. We conclude by raising questions related to the tree model.

(1) Drawing an analogy with decision trees, what happens when the conditions associated with tree edges are allowed to be comparisons involving linear or higher order polynomial functions of the input? Do the bounds weaken in such a situation?

(2) What kind of bounds can one obtain for queries other than orthogonal range query, say circular range query or polyhedral query?

(3) Can the lower bounds in the tree model be extended to data structures that are directed acyclic graphs?

REFERENCES

[1] J. L. BENTLEY AND H. A. MAURER, *Efficient worst-case data structures for range searching*, Acta Inform., 13 (1980), pp. 155-168.

[2] B. CHAZELLE, *Filtering search: A new approach to query answering*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, 1983, pp. 122-132.

[3] ———, *Lower bounds on the complexity of multidimensional searching*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 87-96.

[4] R. COLE AND C. K. YAP, *Geometric retrieval problems*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, 1983, pp. 112-121.

[5] M. L. FREDMAN, *The inherent complexity of dynamic data structures which accommodate range queries*, in Proc. 21st Annual IEEE Symposium on Foundations of Computer Science, 1980, pp. 191-200.

[6] ———, *A lower bound on the complexity of orthogonal range queries*, J. Assoc. Comput. Mach., 28 (1981), pp. 696-705.

[7] ———, *Lower bounds on the complexity of some optimal data structures*, SIAM J. Comput., 10 (1981), pp. 1-10.

[8] H. GABOW, J. BENTLEY, AND R. TARJAN, *Scaling and related techniques for geometric problems*, in Proc. 16th Annual ACM Symposium on Theory of Computing, 1984, pp. 135-143.

[9] G. S. LUEKER, *A data structure for orthogonal range queries*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 28-34.

[10] P. M. VAIDYA, *Space-time tradeoffs for orthogonal range queries*, in Proc. 17th Annual ACM Symposium on Theory of Computing, 1985, pp. 169-174.

[11] D. E. WILLARD, *New data structures for orthogonal range queries*, Tech. Report TR-22-78, Aiken Computer Laboratory, Harvard University, Cambridge, MA, 1978.

[12] ———, *Polygon retrieval*, SIAM J. Comput., 11 (1982), pp. 149-165.

[13] A. C. YAO, *Space-time tradeoff for answering range queries*, in Proc. 14th Annual ACM Symposium on Theory of Computing, 1982, pp. 128-136.

[14] ———, *On the complexity of maintaining partial sums*, SIAM J. Comput., 14 (1985), pp. 277-288.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 759-765, August 1989

A LOWER BOUND FOR MATRIX MULTIPLICATION*

NADER H. BSHOUTY†

Abstract. This paper proves that computing the product of two $n \times n$ matrices over the binary field requires at least $2.5\,n^2 - o(n^2)$ multiplications.

Key words. matrix multiplication, arithmetic complexity, lower bounds, linear codes

AMS(MOS) subject classifications. 15A63, 15A03, 68Q40

1. Introduction. Let $\mathbf{x} = (x_1, \cdots, x_n)^T$ and $\mathbf{y} = (y_1, \cdots, y_m)^T$ be column vectors of indeterminates. A straight-line algorithm for computing a set of bilinear forms in $\mathbf{x}$ and $\mathbf{y}$ is called *quadratic* (respectively, *bilinear*), if all its nonscalar multiplication are of the shape $l(\mathbf{x}, \mathbf{y}) \cdot l'(\mathbf{x}, \mathbf{y})$, (respectively, $l(\mathbf{x}) \cdot l'(\mathbf{y})$), where l and l' are linear forms of the indeterminates.[1]

In this paper we establish the new $2.5n^2 - o(n^2)$ lower bound on the multiplicative complexity of quadratic algorithms for multiplying $n \times n$ matrices over the binary field $\mathbf{Z}_2$. Let $M_{\mathbf{F}}(n, m, k)$ and $\bar{M}_{\mathbf{F}}(n, m, k)$ denote the number of multiplications required to compute the product of $n \times m$ and $m \times k$ matrices by means of quadratic and bilinear algorithms, respectively, over the field $\mathbf{F}$. It is known from [18], [19], and [7] that the complexity of quadratic algorithms for matrix multiplication is lower than that of bilinear algorithms, namely, $M_{\mathbf{Z}_2}(n, 2, 2) \leqq 3n + 2$, whereas $\bar{M}_{\mathbf{Z}_2}(n, 2, 2) \geqq 3.5n$. It has been proved in [1] that for any field $\mathbf{F}$

$$M_{\mathbf{F}}(n, n, n) \geqq 2n^2 - 1,$$

and for the case of binary field, it has been shown in [11] that

$$\bar{M}_{\mathbf{Z}_2}(n, n, n) \geqq 2n^2 + \left\lceil \frac{3}{46} n \right\rceil - 2.$$

Our bound on $M_{\mathbf{Z}_2}(n, n, n)$ is given by Theorem 1 below.

THEOREM 1. *We have*

$$M_{\mathbf{Z}_2}(n, n, n) \geqq 2.5n^2 - 0.5n \log n - O(n).$$

Obviously, the above bound holds for the ring of integers as well.

The rest of the paper is organized as follows. In the next section we give some basic definitions and preliminary results. The proof of Theorem 1 is presented in § 3.

2. Basic definitions and preliminary results. Let $\mathbf{A} = \{A_1, \cdots, A_p\}$ be a set of matrices. A *characteristic matrix of* $\mathbf{A}$ is the matrix $\mathbf{A}(\mathbf{z}) = \sum_{i=1}^{p} A_i z_i$ where $\mathbf{z} = (z_1, \cdots, z_p)^T$ is a vector of indeterminates. Conversely, let $\mathbf{A}(\mathbf{z}) = \sum_{i=1}^{p} z_i A_i$ be a matrix whose entries are linear combinations of $\{z_1, \cdots, z_p\}$. The set of matrices $\{A_1, \cdots, A_p\}$ will be denoted by $\mathbf{A}$. We denote by $\mu_{\mathbf{F}}(\mathbf{A}(\mathbf{z}))$ or $\mu_{\mathbf{F}}(\mathbf{A})$ the minimal number of nonscalar multiplications required to compute $\mathbf{x}^T A_1 \mathbf{y}, \cdots, \mathbf{x}^T A_p \mathbf{y}$ by a quadratic algorithm.

---

* Received by the editors March 29, 1988; accepted for publication (in revised form) November 3, 1988. This research was supported by an M. Landau Research Prize.

† Department of Computer Science, Technion-Israel Institute of Technology, Haifa 32000, Israel.

[1] It is known from [17] that over infinite fields we can restrict ourselves to quadratic algorithms without increasing the multiplicative complexity of a set of quadratic forms. For the finite fields, quadratic algorithms are optimal in the family of algorithms without division, (see [21]).

The characteristic matrix of the bilinear forms defined by the product of the $n \times m$ and $m \times k$ matrices

$$X_{n\times m} \cdot Y_{m\times k} = \begin{bmatrix} x_1 & x_{n+1} & \cdots & x_{(m-1)n+1} \\ x_2 & x_{n+2} & \cdots & x_{(m-1)n+2} \\ \vdots & \vdots & \vdots & \vdots \\ x_n & x_{2n} & \cdots & x_{mn} \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_k \\ y_{k+1} & y_{k+2} & \cdots & y_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ y_{(m-1)k+1} & y_{(m-1)k+2} & \cdots & y_{mk} \end{bmatrix}$$

is

$$\mathbf{A}_{n,m,k}(\mathbf{z}) = \left.\begin{bmatrix} \Phi_{n,k}(\mathbf{z}) & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \Phi_{n,k}(\mathbf{z}) \end{bmatrix}\right\} m \text{ times},$$

where

$$\Phi_{n,k}(\mathbf{z}) = \begin{bmatrix} z_1 & z_2 & \cdots & z_k \\ z_{k+1} & z_{k+2} & \cdots & z_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ z_{(n-1)k+1} & z_{(n-1)k+2} & \cdots & z_{nk} \end{bmatrix}$$

(see [2]).

In [9] Hopcroft and Munsinski showed that

$$\bar{M}_{\mathbf{F}}(n, m, k) = \bar{M}_{\mathbf{F}}(n, k, m) = \bar{M}_{\mathbf{F}}(m, n, k).$$

Since $X_{n\times m} Y_{m\times k} = (Y^T_{m\times k} X^T_{n\times m})^T$ we have

$$M_{\mathbf{F}}(n, m, k) = M_{\mathbf{F}}(k, m, n).$$

Below we assume that $\mathbf{F} = \mathbf{Z}_2$ and omit the subscript $\mathbf{Z}_2$.

Let n be an integer. A *binary linear code* of length n is a linear subspace $\mathbf{C}$ of $\mathbf{Z}_2^n$. If $\dim \mathbf{C} = k$ then $\mathbf{C}$ is called $[n, k]$ code. For $c \in \mathbf{C}$ the *weight* of c, denoted by $\mathrm{wt}(c)$, is the number of the nonzero components of c. The *minimal weight* of $\mathbf{C}$ in $\min\{\mathrm{wt}(c) \mid c \in \mathbf{C} - \{\mathbf{0}\}\}$. We say that $\mathbf{C}$ is a *binary* $[n, k, d]$ *code* if $\mathbf{C} \subseteq \mathbf{Z}_2^n$ is a code of dimension k and minimal weight d. Let $N(k, d)$ be the smallest integer such that there exist a binary $[N(k, d), k, d]$ code. The connection between binary linear codes and the complexity of quadratic forms over $\mathbf{Z}_2$ is given in the following lemma.

LEMMA 1. *Let* $\mathbf{A} = \{A_1, \cdots, A_p\}$ *be a set of binary matrices and let* $L(\mathbf{A})$ *be the linear space spanned by* $\mathbf{A}$. *Let* $d = \min_{A \in L(\mathbf{A})} \operatorname{rank} A$ *and* $k = \dim L(\mathbf{A})$. *Then*

$$\mu_{\mathbf{F}}(\mathbf{A}) \geqq N(k, d).$$

In [2] Brockett and Dobkin proved Lemma 1 for bilinear algorithms. (See also [15].) The same proof is true for quadratic algorithms. This lemma was used in [3]–[12], and [14] to obtain lower bounds for some tensors rank.

Lemma 2 below gives a bound on $N(k, d)$.

LEMMA 2 (Griesmer Bound [13, p. 59]). *We have*

$$N(k, d) \geqq \sum_{i=0}^{k-1} \left\lceil \frac{d}{2^i} \right\rceil.$$

Another lower bound technique for quadratic algorithms is as follows.

LEMMA 3 ([21], [20], [10]). *Let* $\mathbf{A}(\mathbf{z}) = \mathbf{A}_1(\mathbf{z}_1) + \mathbf{A}_2(\mathbf{z}_2)$, *where* $\mathbf{z}_1$ *and* $\mathbf{z}_2$ *are distinct vectors of indeterminates of length* d_1 *and* d_2, *respectively. Then*

$$\mu_{\mathbf{F}}(\mathbf{A}(\mathbf{z})) \geqq \min_{N \in \mathcal{N}} \mu_{\mathbf{F}}[\mathbf{A}_1(\mathbf{z}_1) + \mathbf{A}_2(N\mathbf{z}_1)] + \dim L(\mathbf{A}_2),$$

where $\mathcal{N}$ *is the set of all* $d_2 \times d_1$ *matrices.*

For the polynomial $p(\lambda)=\lambda^n+a_{n-1}\lambda^{n-1}+\cdots+a_0\in \mathbf{Z}_2[\lambda]$ the *companion matrix* C_p of $p(\lambda)$ is defined by

$$C_p=\begin{bmatrix}0 & 0 & \cdots & 0 & -a_0\\ 1 & 0 & \cdots & 0 & -a_1\\ 0 & 1 & \cdots & 0 & -a_2\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ 0 & 0 & \cdots & 1 & -a_{n-1}\end{bmatrix}.$$

We define the set of $n\times n$ matrices $\mathbf{A}^p$ by

$$\mathbf{A}^p=\{C_p^0, C_p^1, \cdots, C_p^{n-1}\}.$$

If $p(\lambda)$ is an irreducible polynomial, then $L(\mathbf{A}^p)=\{f(C_p)\,|\,f\in \mathbf{Z}_2[\lambda],\ \deg f\leqq n-1\}$, every matrix in $L(\mathbf{A}^p)$ is of rank n, and dim $L(\mathbf{A}^p)=n$. Let $p(\lambda)$ and $q(\lambda)$ be irreducible polynomials over $\mathbf{Z}_2$ of degree n and m, respectively, where $m\geqq n$. Let $\mathbf{0}_{j\times k}$ denote the zero $j\times k$ matrix.

Consider the matrix

$$\mathbf{C}(\mathbf{z})=\begin{bmatrix}\mathbf{A}^p(\mathbf{z}_0) & \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m}\\ \mathbf{E}_{1,1}(\mathbf{z}_{1,1}) & \mathbf{A}^p(\mathbf{z}_0) & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m}\\ \mathbf{E}_{2,1}(\mathbf{z}_{2,1}) & \mathbf{E}_{2,2}(\mathbf{z}_{2,2}) & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m}\\ \vdots & \vdots & \vdots & \vdots & \vdots\\ \mathbf{E}_{k-1,1}(\mathbf{z}_{k-1,1}) & \mathbf{E}_{k-1,2}(\mathbf{z}_{k-1,2}) & & \mathbf{A}^p(\mathbf{z}_0) & \mathbf{0}_{n\times m}\\ \mathbf{G}_1(\mathbf{z}_{k,1}) & \mathbf{G}_2(\mathbf{z}_{k,2}) & \cdots & \mathbf{G}_k(\mathbf{z}_{k,k}) & \mathbf{A}^q(\mathbf{z}_0)\end{bmatrix},$$

where $\mathbf{z}_0, \mathbf{z}_{1,1}, \cdots, \cdots, \mathbf{z}_{k,k}$ are distinct vectors of indeterminates $\mathbf{z}_0=(z_{0,1}, \cdots, z_{0,m})$,

$$\mathbf{A}^p(\mathbf{z}_0)=\sum_{i=1}^{n} C_p^{i-1}z_{0,i}, \qquad \mathbf{A}^q(\mathbf{z}_0)=\sum_{i=1}^{m} C_q^{i-1}z_{0,i},$$

$$\mathbf{E}_{i,j}(\mathbf{z}_{i,j})=\begin{bmatrix}z_{i,j,1,1} & \cdots & z_{i,j,1,n}\\ \vdots & \vdots & \vdots\\ z_{i,j,n,1} & \cdots & z_{i,j,n,n}\end{bmatrix}, \qquad \mathbf{G}_i(\mathbf{z}_{k,i})=\begin{bmatrix}z_{k,i,1,1} & \cdots & z_{k,i,1,n}\\ \vdots & \vdots & \vdots\\ z_{k,i,m,1} & \cdots & z_{k,i,m,n}\end{bmatrix}.$$

Let

$$\mathbf{C}^*(\mathbf{z})=I_w\otimes\mathbf{C}(\mathbf{z})=\begin{bmatrix}C(\mathbf{z}) & & & \mathbf{0}\\ & \ddots & & \\ & & w \text{ times} & \\ & & & \ddots\\ \mathbf{0} & & & C(\mathbf{z})\end{bmatrix},$$

where I_w is the identity matrix of order w and $\otimes$ is the Kronecker product of matrices. Then $L(\mathbf{C}^*)\subseteq L(\mathbf{A}_{nk+m,w,nk+m})$ and therefore

$$\mu_{\mathbf{F}}(\mathbf{C}^*)\leqq M(nk+m, w, nk+m). \tag{1}$$

The characteristic matrix $\mathbf{C}^*(\mathbf{z})$ can be written as

$$\begin{aligned}\mathbf{C}^*(\mathbf{z})&=\mathbf{C}'(\mathbf{z}^{(1)})+\mathbf{E}'(\mathbf{z}^{(2)})+\mathbf{G}'(\mathbf{z}^{(3)})+\mathbf{H}'(\mathbf{z}^{(4)})\\ &=I_w\otimes(\tilde{\mathbf{C}}(\mathbf{z}^{(1)})+\tilde{\mathbf{E}}(\mathbf{z}^{(2)})+\tilde{\mathbf{G}}(\mathbf{z}^{(3)})+\tilde{\mathbf{H}}(\mathbf{z}^{(4)})),\end{aligned}$$

where

$$\tilde{\mathbf{C}}(\mathbf{z}^{(1)}) = \begin{bmatrix} I_k \otimes \mathbf{A}^p(\mathbf{z}_0) & \mathbf{0} \\ \mathbf{0} & \sum_{i=1}^{n} z_{0,i} C_q^{i-1} \end{bmatrix},$$

$$\tilde{\mathbf{E}}(\mathbf{z}^{(2)}) = \begin{bmatrix} \mathbf{0}_{n\times n} & \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ \mathbf{E}_{1,1}(\mathbf{z}_{1,1}) & \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ \mathbf{E}_{2,1}(\mathbf{z}_{2,1}) & \mathbf{E}_{2,2}(\mathbf{z}_{2,2}) & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{E}_{k-1,1}(\mathbf{z}_{k-1,1}) & \mathbf{E}_{k-1,1}(\mathbf{z}_{k-1,1}) & \cdots & \mathbf{E}_{k-1,k-1}(\mathbf{z}_{k-1,k-1}) & \mathbf{0}_{n\times m} \\ \mathbf{0}_{m\times n} & \mathbf{0}_{m\times n} & \cdots & \mathbf{0}_{m\times n} & \mathbf{0}_{m\times m} \end{bmatrix},$$

$$\tilde{\mathbf{G}}(\mathbf{z}^{(3)}) = \begin{bmatrix} \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ \mathbf{G}_{k,1}(\mathbf{z}_{k,1}) & \cdots & \mathbf{G}_{k,k}(\mathbf{z}_{k,k}) & \mathbf{0}_{m\times m} \end{bmatrix},$$

and

$$\tilde{\mathbf{H}}(\mathbf{z}^{(4)}) = \begin{bmatrix} I_k \otimes \mathbf{0}_{n\times n} & \mathbf{0} \\ \mathbf{0} & \sum_{i=n+1}^{m} z_{0,i} C_q^{i-1} \end{bmatrix}.$$

By Lemma 3, we have

$$\begin{aligned} \mu_{\mathbf{F}}(\mathbf{C}^*) \geqq \min \mu_{\mathbf{F}}[\mathbf{C}'(\mathbf{z}^{(1)}) + \mathbf{E}'(W_1\mathbf{z}^{(1)}) + \mathbf{G}'(W_2\mathbf{z}^{(1)}) + \mathbf{H}'(W_3\mathbf{z}^{(1)})] \\ + \dim L(\mathbf{E}') + \dim L(\mathbf{G}') + \dim L(\mathbf{H}'), \end{aligned} \tag{2}$$

where the minimum is over all the matrices W_1, W_2 and W_3. We shall therefore estimate all the terms appearing in the right-hand side of (2). We have

$$\dim L(\mathbf{E}') = \dim L(\tilde{\mathbf{E}}) = \sum_{s=1}^{k-1} \sum_{r=1}^{s} \dim L(\mathbf{E}_{s,r}) = \frac{k^2-k}{2} n^2, \tag{3}$$

$$\dim L(\mathbf{G}') = \dim L(\tilde{\mathbf{G}}) = \sum_{r=1}^{k} \dim L(\mathbf{G}_{k,r}) = kmn, \tag{4}$$

and

$$\dim L(\mathbf{H}') = \dim L(\tilde{\mathbf{H}}) = m - n. \tag{5}$$

Let

$$\mathbf{D}'(\mathbf{z}^{(1)}) = \mathbf{C}'(\mathbf{z}^{(1)}) + \mathbf{E}'(W_1\mathbf{z}^{(1)}) + \mathbf{G}'(W_2\mathbf{z}^{(1)}) + \mathbf{H}'(W_3\mathbf{z}^{(1)}).$$

Every nonzero matrix in $L(\mathbf{D}')$ is of the shape $D' = I_w \otimes D$, where

$$D = \begin{bmatrix} f(C_p) & \mathbf{0}_{n\times n} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ E_{1,1} & f(C_p) & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ E_{2,1} & E_{2,2} & \cdots & \mathbf{0}_{n\times n} & \mathbf{0}_{n\times m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ E_{k-1,1} & E_{k-1,2} & \cdots & f(C_p) & \mathbf{0}_{n\times m} \\ G_1 & G_2 & \cdots & G_k & f(C_q) + \sum_{r=n+1}^{m} \delta_r C_q^{r-1} \end{bmatrix}$$

for some polynomial $f(\lambda) \in \mathbf{Z}_2[\lambda]$ of degree less than n. Thus

$$\det D' = (\det D)^w = (\det f(C_p))^{kw}\left(\det\left(f(C_q)+\sum_{r=n+1}^{m}\delta_r C_q^{r-1}\right)\right)^w \neq 0.$$

Therefore every matrix in $L(\mathbf{D}')$ is of rank $(kn+m)w$. Since, obviously, dim $L(\mathbf{D}') = n$, by Lemma 1, for any W_1, W_2 and W_3 we have

$$\mu_{\mathbf{F}}[\mathbf{C}'(\mathbf{z}^{(1)})+\mathbf{E}'(W_1\mathbf{z}^{(1)})+\mathbf{G}'(W_2\mathbf{z}^{(1)})+\mathbf{H}'(W_3\mathbf{z}^{(1)})] \geqq N(n,(kn+m)w).$$

This in conjunction with (1)-(5) gives the following in equality.

LEMMA 4. *We have*

$$M(nk+m, w, nk+m) \geqq \frac{k^2-k}{2}n^2+kmn+m-n+N(n,(kn+m)w).$$

Note that the only property of the companion matrix of an irreducible polynomial we use is that every nonsingular space spanned by $\{C_p^0, \cdots, C_p^{n-1}\}$ is of rank n. Actually, every set $\mathbf{A}=\{A_0, \cdots, A_{n-1}\}$ of n independent matrices with the above property is available for our proof.

LEMMA 5 [2]. *We have*

$$\bar{M}(n, m, k) \geqq \bar{M}(n-j, m, k)+j\max(m, k), \quad M(n, m, k) \geqq M(n-j, m, k)+jk.$$

LEMMA 6. *If $m \geqq n \geqq k$, then for every $1 \leqq j \leqq k$*

$$\bar{M}(n, m, k) \geqq (n-k)m+(\lfloor k/j\rfloor -1)(k-0.5j\lfloor k/j\rfloor)j+(k-j\lfloor k/j\rfloor)+N(j, km).$$

If $n \geqq k$, then

$$M(n, m, k) \geqq (n-k)k+(\lfloor k/j\rfloor -1)(k-0.5j\lfloor k/j\rfloor)j+(k-j\lfloor k/j\rfloor)+N(j, km).$$

Proof. By Lemma 5 with $j = n-k$, we have

$$\bar{M}(n, m, k) \geqq (n-k)m+\bar{M}(k, m, k).$$

Then substituting m, $\lfloor k/j\rfloor -1$, j, and $k-(\lfloor k/j\rfloor -1)j$ for w, k, n and m, respectively, in Lemma 4, we obtain

$$\bar{M}(k, m, k) \geqq (\lfloor k/j\rfloor -1)(k-0.5j\lfloor k/j\rfloor)j+(k-j\lfloor k/j\rfloor)+N(j, km)$$

for any $1 \leqq j \leqq k$. The bound on $M(n, m, k)$ can be proved in the same manner. □

3. Proof of the Theorem 1.

By Lemma 5 we have

$$M(n, n, n) \geqq \left(\left\lfloor\frac{n}{j}\right\rfloor -1\right)\left(n-\frac{\lfloor n/j\rfloor}{2}j\right)j+\left(n-j\left\lfloor\frac{n}{j}\right\rfloor\right)+N(j, n^2).$$

Let t be the minimal positive integer such that j divide $n-t$, then $t<j$ and

$$\left\lfloor\frac{n}{j}\right\rfloor = \frac{n-t}{j}.$$

Therefore

$$\begin{aligned} M(n, n, n) &\geqq (n-t-j)\left(\frac{n+t}{2}\right)+t+N(j, n^2) \\ &= \frac{n^2}{2}+N(j, n^2)-j\frac{n}{2}-\frac{t^2}{2}-j\frac{t}{2}+t \\ &\geqq 2.5n^2-\left(\frac{n^2}{2^{j-1}}+\frac{jn}{2}\right)-j^2, \end{aligned}$$

where the last inequality follows from Lemma 2. Now for $j = \lceil \log n \rceil + 1$ we obtain

$$M(n, n, n) \geqq 2.5n^2 - 0.5n \log n - O(n). \qquad \square$$

Other results concerning $\bar{M}(n, m, k)$ and $M(n, m, k)$ are summarized in Theorem 2 below.

THEOREM 2. *If* $n \geqq k$, *then*

$$M(n, m, k) \geqq \begin{cases} nk + 2km - k^2 + k - \log km & \text{if } km \leqq 2^{k-1}, \\ nk + 2km - k^2 - (km/2^{k-1}) & \text{if } km \geqq 2^{k-1}, \\ nk + 2km - 0.5k^2 - 0.5k \log m - O(k) - O(\log^2 m) & \text{if } \log m = o(k). \end{cases}$$

If $m \geqq n \geqq k$, *then*

$$\bar{M}(n, m, k) \geqq \begin{cases} nm + km + k - \log km & \text{if } km \leqq 2^{k-1}, \\ nm + km - (km/2^{k-1}) & \text{if } km \geqq 2^{k-1}, \\ nm + km + 0.5k^2 - 0.5k \log m - O(k) - O(\log^2 m) & \text{if } \log m = o(k). \end{cases}$$

Proof. The first two bounds on $\bar{M}$ are obtained if we put $j = k$ in Lemma 6 and the last bound is obtained if we put $j = \lceil \log m \rceil$. The bound on $M(n, m, k)$ can be shown in a similar manner. $\square$

The bounds compare favorably with the bound $m(n+k) + (k-m) + 1$ established in [2] and [6]. We must note, however, that the bound in [2] and [6] holds over any field.

Acknowledgments. I am grateful to Michael Kaminski for introducing me to the subject and for many helpful discussions, to Ron M. Roth for checking the proofs, to my brother Daoud Bshouty and Michael Kaminski for correcting my very poor English, and to my wife Nadera Bshouty for encouraging me.

REFERENCES

[1] A. ALDER AND V. STRASSEN, *On the algorithmic complexity of associative algebras*, Theoret. Comput. Sci., 15 (1981), pp. 201-211.

[2] R. W. BROCKETT AND D. DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Linear Algebra Appl., 19 (1978), pp. 207-235.

[3] N. H. BSHOUTY, *On the algorithmic complexity of associative algebras over finite fields*, in preparation.

[4] M. R. BROWN AND D. P. DOBKIN, *An improved lower bound on polynomial multiplication*, IEEE Trans. Comput., 29 (1980), pp. 337-340.

[5] J. VON ZUR GATHEN, *Maximal bilinear complexity and codes*, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1988.

[6] W. HARTMANN, *On the multiplicative complexity of modules over associative algebras*, SIAM J. Comput., 14 (1985), pp. 383-395.

[7] J. E. HOPCROFT AND L. R. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30-36.

[8] ———, *Some techniques for proving certain simple programs optimal*, in Proc. 10th Annual Symposium Switching and Automata Theory, 1969, pp. 36-45.

[9] J. HOPCROFT AND J. MUNSINSKI, *Duality applied to the complexity of matrix multiplication*, SIAM J. Comput., 2 (1973), pp. 159-173.

[10] J. JA' JA', *On the complexity of bilinear forms with commutative*, SIAM J. Comput., 4 (1980), pp. 713-728.

[11] J. JA' JA' AND J. TAKCHE, *Improved lower bound for some matrix multiplication problems*, Inform. Process. Lett., 21 (1985), pp. 123-127.

[12] M. KAMINSKI, *A lower bound for polynomial multiplication*, Theoret. Comput. Sci., 40 (1985), pp. 319-322.

[13] J. H. VAN LINT, *Introduction to Coding Theory*, Springer-Verlag, Berlin, New York, 1980.

[14] A. LEMPEL, G. SEROUSSI AND S. WINOGRAD, *On the complexity of multiplication in finite fields*, Theoret. Comput. Sci., 22 (1983), pp. 285-296.

[15] A. LEMPEL AND S. WINOGRAD, *A new approach to error-correcting codes*, IEEE Trans. Inform. Theory, 23 (1977), pp. 503-508.

[16] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354-356.

[17] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.
[18] A. WAKSMAN, *On Winograd's algorithm for inner product*, IEEE Trans. Comput., 19 (1970), pp. 360–361.
[19] S. WINOGRAD, *A new algorithm for inner product*, IEEE Trans. Comput., 17 (1968), pp. 693–694.
[20] ———, *On the number of multiplications necessary to compute certain functions*, Comm. Pure Appl. Math., 23 (1970), pp. 165–179.
[21] ———, *On multiplication of* 2×2 *matrices*, Linear Algebra Appl., 4 (1971), pp. 381–388.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 766-776, August 1989

010

TIME/SPACE TRADE-OFFS FOR REVERSIBLE COMPUTATION*

CHARLES H. BENNETT†

Abstract. A reversible Turing machine is one whose transition function is 1:1, so that no instantaneous description (ID) has more than one predecessor. Using a pebbling argument, this paper shows that, for any $\varepsilon > 0$, ordinary multitape Turing machines using time T and space S can be simulated by reversible ones using time $O(T^{1+\varepsilon})$ and space $O(S \log T)$ or in linear time and space $O(ST^{\varepsilon})$. The former result implies in particular that reversible machines can simulate ordinary ones in quadratic space. These results refer to reversible machines that save their input, thereby insuring a global 1:1 relation between initial and final IDs, even when the function being computed is many-to-one. Reversible machines that instead erase their input can of course compute only 1:1 partial recursive functions and indeed provide a Gödel numbering of such functions. The time/space cost of computing a 1:1 function on such a machine is equal within a small polynomial to the cost of computing the function and its inverse on an ordinary Turing machine.

Key words. reversibility, invertibility, time, space, Turing machine, TISP

AMS(MOS) subject classifications. 68Q05, 68Q15

1. Introduction. A deterministic computer is called reversible if it is also backwards deterministic, having no more than one logical predecessor for each whole-machine state. Reversible computers must avoid such common irreversible operations as assignments and transfers of control that throw away information about the previous state of the data or program. Reversible computers of various kinds (Turing machines, cellular automata, combinational logic) have been considered [1], [11], [12], [13], [6], [2], [14] especially in connection with the physical question of the thermodynamic cost of computation; and it has been known for some time that they can simulate the corresponding species of irreversible computers in linear time [1] (or linear circuit complexity [13]), provided they are allowed to leave behind at the end of the computation a copy of the input (thereby rendering the mapping between initial and final states 1:1 even though the input-output mapping may be many-to-one).

The physical significance of reversible simulation stems from the fact [7], [1] that many-to-one data operations have an irreducible thermodynamic cost, while one-to-one operations do not. The ability to program any computation as a sequence of 1:1 operations therefore implies that, using appropriate hardware, an arbitrarily large amount of "computational work" (in conventional units of time/space complexity) can in principle be performed per unit of physical energy dissipated by the computer. This result contradicted an earlier widely held belief [15] that any computer operating at temperature T must dissipate at least $kT \ln 2$ (the heat equivalent to one bit of entropy, approximately equal to the kinetic energy of a gas molecule at temperature T) per elementary binary operation performed. Needless to say, most real computing equipment, both electronic and biological, is very inefficient thermodynamically, dissipating many orders of magnitude more than kT per step. However, a few thermodynamically efficient data processing systems do exist, notably genetic enzymes such as RNA polymerase, which, under appropriate reactant concentrations, can transcribe information from DNA to RNA at a thermodynamic cost considerably less than kT per step.

* Received by the editors January 27, 1988; accepted for publication October 18, 1988.
† IBM Thomas J. Watson Research Center, Room 21-125, Yorktown Heights, New York 10598.

The old linear-time simulation [1] is not at all economical of space: in the worst case, it uses exponentially more space than the irreversible computation being simulated.

Here we review this old result and present some new results on more space-efficient reversible computation using the formalism of multitape Turing machines. For simplicity we consider (deterministic) machines whose input tape and output tape, like the work tapes, are two-way, read-write, and included in the space accounting. As usual we restrict ourselves to "well-formed" machines (a recursive subset of machines) whose computations, if started in a standard initial ID will either fail to halt or will halt in a standard final ID. A standard initial ID is one with the Turing machine's control unit in the designated start state, with the input in standard format (input head scanning the blank square just left of a finite input string containing no embedded blanks) and with work and output tapes blank. A standard final ID is one in which the control unit is in a single designated terminal state, all work tapes are blank and all heads have been restored to their original locations, and the output tape contains a single string in standard format. Let $\{\varphi_i\}$ be a Gödel numbering of well-formed machines, and let $T_i(x)$ and $S_i(x)$ denote the time and space used by machine i on input x, the partial functions T_i and S_i having the same domain as φ_i.

A machine φ_i is *reversible* if its quintuples have disjoint ranges, thereby rendering the transition function 1 : 1. We define two special kinds of standard, reversible machine:

Input-saving machines whose input head is read-only, so that the input string remains unchanged throughout the computation.

Input-erasing machines in which the input tape, like the work tapes, is required to be blank at the end of any halting computation on standard input.

These requirements, like the well-formedness requirements for ordinary Turing machines, can be enforced through recursive constraints on the quintuples, so that the input-saving and input-erasing reversible machines can be Gödel-numbered as recursives subsets of $\{\varphi_i\}$.

2. Input-saving reversible computations. As a preliminary to the new results we restate the old, linear-time simulation of [1].

LEMMA 1. *Each conventional multitape Turing machine running in time T and space S can be simulated by a reversible input-saving machine running in time $O(T)$ and space $O(S+T)$. More precisely, there exists a constant c such that for each ordinary Turing machine φ_i, an index j can be found effectively from i, such that φ_j is reversible and input-saving, $\varphi_j = \varphi_i$, and $T_j < c + c \cdot T_i$, and $S_j < c + c \cdot (S_i + T_i)$.*

Proof. As illustrated by Table 1 below, the simulation proceeds by three stages. The first stage uses an extra tape, initially blank, to record all the information that would have been thrown away by the irreversible computation being simulated. This history-taking renders the computation reversible, but also leaves a large amount of unwanted data on the history tape (hence the epithet "untidy"). The second stage copies the output produced by the first stage onto a separate output tape, an action that is intrinsically reversible (without history-taking) if the output tape is initially blank. The third stage exactly reverses the work of the first stage, thereby restoring the whole machine, except for the now-written output tape, to its original condition. This cleanup is possible precisely because the first stage was reversible and deterministic. An apparent problem in constructing the cleanup stage, viz. that conventional read/write/shift quintuples have inverses of a different form (shift/read/write), can be avoided by expressing each read/write/shift of the untidy stage as a product of nonshifting read/writes and oblivious shifts, then inverting these. For more details, see the Appendix and [1].

TABLE 1

Basic linear-time reversible simulation using an extra tape to record, temporarily, a history of the irreversible computation being simulated (underbars denote head positions).

Stage	Input tape	Work tape(s)	History tape	Output tape
Untidy	_INPUT	–	–	–
	INPUT	WORK	HIST_	–
	_INPUT	_OUTPUT	HISTORY_	–
Copy output	_INPUT	_OUTPUT	HISTORY_	–
	_INPUT	OUTPUT	HISTORY_	OU_
	_INPUT	OUTPUT_	HISTORY_	OUTPUT_
	_INPUT	_OUTPUT	HISTORY_	_OUTPUT
Cleanup	_INPUT	_OUTPUT	HISTORY_	_OUTPUT
	INPUT	WORK	HIST_	_OUTPUT
	_INPUT	–	–	_OUTPUT

Note in passing that the operation of copying onto a blank tape serves in stage 2 as a substitute for ubiquitous assignment operation $a := b$, which is not permitted in reversible programming. Similarly, the inverse of copying onto blank tape, which reversibly erases one of two strings known to be identical, can take the place of the more general but irreversible operation of erasure. If the natural number 0 is identified with the empty string, reversible copying and erasure can be represented as incrementation and decrementation, with $a := a + b$ having the same effect as $a := b$, if $a = 0$ initially, and $a := a - b$ having the same effect as $a := 0$, if $a = b$ initially.

The above scheme for reversible simulation runs in linear time but sometimes uses as much as exponential space. The following theorem shows that by allowing slightly more than linear time a much more space-efficient simulation can be obtained.

THEOREM 1. *For any $\varepsilon > 0$, any multitape Turing machine running in time T and space S can be simulated by a reversible input-saving machine using time $O(T^{1+\varepsilon})$ and space $O(S \cdot \log T)$.*

Proof. This space-efficient simulation is obtained by breaking the original computation into segments of m steps, where $m \approx S$, then doing and undoing these segments in a hierarchical manner as shown in Table 2. This table shows that by hierarchically iterating the simulation of 2 segments of original computation by 3 stages of reversible computation, 2^n segments of the original computation can be simulated in 3^n stages of reversible computation. More generally, for any fixed $k > 1$, k^n segments of irreversible computation can be simulated by $(2k-1)^n$ stages of reversible computation. Each stage of reversible computation invokes the linear-time reversible simulation of Lemma 1 to create or destroy a checkpoint (a complete intermediate ID of the simulated computation) from the preceding, m-steps earlier, checkpoint, using time and space $O(S)$. Taking $n \approx \log(T/m)/\log(k)$, the number of intermediate checkpoints in storage at any one time is at most $n(k-1)$, which for fixed k is $O(\log T)$, and since each checkpoint uses storage $O(S)$, the space requirement is $O(S \cdot \log T)$, while the time requirement is $O(S \cdot (2k-1)^n) = O(T^{(1+1/\log k)})$. The simulation is applied to segments of length $m \approx S$ rather than to single steps ($m = 1$) to insure that the time $O(S)$ of copying or erasing a checkpoint does not dominate the time $O(m)$ of computing it from the previous checkpoint. The extra space $O(m)$ required for the history tape in

TABLE 2
Reversible simulation in time $O(T^{\log 3/\log 2})$ and space $O(S \cdot \log T)$.

Stage	Action	Checkpoints in storage (0 = initial ID, checkpoint $j = (jm)$th step ID)								
0	Start	0								
1	Do segment 1	0	1							
2	Do segment 2	0	1	2						
3	Undo segment 1	0		2						
4	Do segment 3	0		2	3					
5	Do segment 4	0		2	3	4				
6	Undo segment 3	0		2		4				
7	Do segment 1	0	1	2		4				
8	Undo segment 2	0	1			4				
9	Undo segment 1	0				4				
10	Do segment 5	0				4	5			
11	Do segment 6	0				4	5	6		
12	Undo segment 5	0				4		6		
13	Do segment 7	0				4		6	7	
14	Do segment 8	0				4		6	7	8
15	Undo segment 7	0				4		6		8
16	Do segment 5	0				4	5	6		8
17	Undo segment 6	0				4	5			8
18	Undo segment 5	0				4				8
19	Do segment 1	0	1			4				8
20	Do segment 2	0	1	2		4				8
21	Undo segment 1	0		2		4				8
22	Do segment 3	0		2	3	4				8
23	Undo segment 4	0		2	3					8
24	Undo segment 3	0		2						8
25	Do segment 1	0	1	2						8
26	Undo segment 2	0	1							8
27	Undo segment 1	0								8

the linear-time simulation between checkpoints does not increase the order of the overall space bound $O(S \cdot \log T)$.

To complete the proof, it remains to be shown how the reversible machine causes the above operations to be performed in correct sequence, how it organizes the storage of the checkpoints on tape, and finally how it arrives at satisfactory values for m and n without knowing T and S beforehand. The hierarchical sequencing of operations is conveniently performed by a recursive subroutine that, given m and n, reversibly calculates the $m \cdot k^n$th successor of an arbitrary ID, using $k-1$ local variables (each restricted to m squares of tape) at each lower level of recursion in n to store the necessary intermediate checkpoints. At the $n = 0$ level of recursion, the mth successor is calculated by the method of Lemma 1. The search for the correct m and n values is performed by a main procedure, which calls the recursive subroutine for successively larger m and n until the time $m \cdot k^n$ and space m allowances prove sufficient, meanwhile keeping a history of the sequence of m and n values to preserve reversibility. When the desired output has been obtained and copied onto the output tape, the search for m and n is undone to dispose of the history of this search. (For more details, see the Appendix.) □

Remark. The reversible creation and annihilation of checkpoints illustrated in Table 2 may be regarded as a case of *reversible pebbling* of a one-dimensional graph. In ordinary pebbling, addition of a pebble to a node requires that all predecessors of

the node already have pebbles, but removal of a pebble can be performed at any time. In reversible pebbling, both the addition and the removal of a pebble require that the predecessors have pebbles.

COROLLARY. *Each ordinary Turing machine running in space S can be simulated by a reversible, input-saving machine running in space $O(S^2)$.*

Related results. Using the same construction, but with k increasing as a function of T (and consequently having n increase less rapidly than $\log T$), one can obtain reversible simulations using less time and more space. For example, using fixed n, k increases linearly with T and one obtains a reversible simulation in time $O(T)$ and space $O(ST^{\varepsilon})$. Intermediate trade-offs can also be obtained, e.g., time $O(T \cdot 2^{\sqrt{\log T}})$ and space $O(S \cdot 2^{(\sqrt{\log T})+O(\log\log T)})$.

The usual linear tape-compression and time speedup theorems apply to reversible machines; therefore, if $T(x)$ and $S(x)$ are total functions bounded below by $3|x|+3$ and $|x|+2$, respectively, then

$$\mathbf{TISP}(T, S) \subseteq \mathbf{RTISP}(T^{1+\varepsilon}, S \log T),$$

where **TISP** (T, S) denotes the class of languages accepted by ordinary Turing machines in time T and space S, and **RTISP** (T, S) denotes the analogous class for input-saving reversible machines. Similarly, the class **TISP** (T, S) is contained in

$$\mathbf{RTISP}(T, ST^{\varepsilon}),$$

and in

$$\mathbf{RTISP}(T \cdot 2^{\sqrt{\log T}}, S \cdot 2^{(\sqrt{\log T})+O(\log\log T)}).$$

It is interesting to compare the time/space trade-off in reversible simulation of ordinary Turing machines with the analogous depth/width trade-off in reversible simulation of ordinary combinatorial logic circuits. A depth-efficient construction due to Fredkin and Toffoli [6], paralleling Lemma 1, simulates ordinary circuits of depth d and width w, in an input-saving manner, by reversible circuits of depth $O(d)$ and width $O(dw)$, composed of reversible gates with no more than three inputs and outputs. A construction parallel to Theorem 1 above would yield a more width-efficient simulation, by reversible circuits of depth $O(d^{1+\varepsilon})$ and width $O(w \cdot \log d)$.

However, a still more width-efficient simulation is possible, based on Coppersmith and Grossman's result [5] that any even permutation on the space of n-bit strings can be computed by a circuit of width n, and depth exponential in n, composed of the reversible gates of $\leqq 3$ inputs and outputs. Width n is insufficient to realize odd permutations, but these can be computed by circuits of width $n+1$, using an extra "garbage" bit whose initial value does not matter, and which is returned unmodified in the output.

Accessing such garbage during a reversible computation might at first appear useless, like renting space in a warehouse already full of someone else's belongings, but in fact it serves the purpose of allowing the odd permutation to be embedded in an even permutation of higher order (any permutation on strings that is oblivious to the values of one or more bits is perforce even) [3]. In particular, any m-bit function of an n-bit argument, $F_m(X_n)$, can be embedded in an even permutation of the form $(X_n, 0^m, g) \to (X_n, F_m(X_n), g)$, where g is a garbage bit, and computed reversibly in width $n+m+1$. Cleve [4] has explored other aspects of the depth/width trade-off for reversible circuits.

The extreme width-efficiency of the Coppersmith–Grossman circuits suggests the possibility of more space-efficient reversible Turing machine simulations using, say,

linear space rather than $S \cdot \log T$. However, this may not be possible, since a deep-but-narrow logic circuit can perform a sequence of *different* transformations, while a Turing machine is forced to apply the *same* transition function at each step.

3. Input-erasing reversible computations. The schemes described so far achieve global reversibility by saving the input, in effect embedding an arbitrary partial recursive function $\varphi(x)$ into the 1:1 partial recursive function $\langle x, \varphi(x)\rangle$. We now consider reversible machines that destroy their input, as well as clearing all working storage, while producing the desired output. Because of unique terminal state of the finite control, such machines cannot destroy or hide even a finite amount of the information present in the input, and so can compute only 1:1 functions. We show that such machines provide a Gödel numbering of the 1:1 partial recursive functions and that the time/space cost of computing a 1:1 function on such a machine is polynomially related to the cost of computing and inverting the function on ordinary Turing machines.

First we need a definition of the cost of inverting a function. Let T^* and S^* be arbitrary partial recursive functions. A 1:1 partial recursive function $f(x)$ will be said to be *invertible in time* $T^*(x)$ *and space* $S^*(x)$ if and only if there exists an (in general irreversible) multitape Turing machine for computing f^{-1} that, for all x in the domain of f, uses time $\leqq T^*(x)$ and space $\leqq S^*(x)$ to compute x from $f(x)$. The theorem below extends an analogous but space-inefficient result [1], [2] on reversible computation of 1:1 functions.

THEOREM 2 (on input-erasing reversible computations). (a) *A function is partial recursive and* 1:1 *if and only if it is computed by an input-erasing reversible multitape Turing machine. These machines thus provide a Gödel-numbering of* 1:1 *partial recursive functions.*

(b) *If f is a* 1:1 *partial recursive function computable in time T and space S, and invertible in time T^* and space S^*, then f can be computed in time $O(T + T^*)$ and space $O(\max\{S \cdot T^{\varepsilon}, S^* \cdot T^{*\varepsilon}\})$ on an input-erasing reversible machine.*

(c) *If f is a* 1:1 *partial recursive function computable in time T and space S, and invertible in time T^* and space S^*, then f can be computed in time $O(T^{1+\varepsilon} + T^{*1+\varepsilon})$ and space $O(\max\{S \log T, S^* \log T^*\})$ on an input-erasing reversible machine.*

(d) *If f is computable by an input-erasing reversible machine in time T and space S, then f is computable by an ordinary machine in time T and space S, and f is invertible by an ordinary machine in time $O(T)$ and space $S + O(1)$.*

Proof. The "if" of part (a) above is immediate from the nature of an input-erasing reversible machine. The "only if" of (a) and the time/space bounds (b) and (c) depend on the construction below (Table 3), in which irreversible algorithms for f and f^{-1} are combined to yield an input-erasing reversible algorithm for f. The "only if" of part (a) in addition depends on the fact noted by McCarthy [10] that, if f is 1:1, then an algorithm for f^{-1} can be obtained effectively from an algorithm for f. (Levin's optimal inverting algorithm [8], [9] is a refinement of this idea). Part (d) is immediate from the fact that an input-saving reversible Turing machine is already a Turing machine, and the fact that the inverse of a reversible machine (obtained by inverting its quintuples as explained in the proof of Lemma 1) runs in the same space and linear time as the original reversible machine.

Table 3 shows how two irreversible algorithms, for a 1:1 partial recursive function f and its inverse f^{-1}, can be combined to perform an input-erasing reversible computation of f. In the first half of the computation the irreversible algorithm for f is used in the manner of Table 1 or Table 2 to construct a temporary bridge (history or checkpoints) from x to $f(x)$, thereby allowing a copy of $f(x)$ to be left on the output

TABLE 3

Input-erasing reversible computation of a 1:1 *function f, given irreversible algorithms for f and* f^{-1}.

Action	Input tape	History tape	Work tapes	Output tape
Do then undo f, via	x			
history or checkpoints,	x	f-HISTORY	$f(x)$	
to create $f(x)$ reversibly	x	f-HISTORY	$f(x)$	$f(x)$
in presence of x	x			$f(x)$
Do then undo f^{-1}, via	x			$f(x)$
history or checkpoints,	x	f^{-1}-HISTORY	x	$f(x)$
to destroy x reversibly		f^{-1}-HISTORY	x	$f(x)$
in presence of $f(x)$				$f(x)$

tape after the bridge is removed. In the last half of the computation the algorithm for f^{-1} is used similarly to generate a bridge from $f(x)$ back to x, but in this case the bridge is used to delete the copy of x already present on the input tape before the bridge was constructed. The time and space required are evidently as stated in (b) and (c) above. □

Remark. Theorem 2 implies Theorem 1, since for any function φ computable in time T and space S, the function $f(x) = x * \varphi(x)$, where $*$ is a distinctive punctuation, is 1:1 and both computable and invertible in time $O(T)$ and space $O(S)$.

Appendix. Details of proofs. Here we give further details of the proofs of Lemma 1 and Theorem 1.

Lemma 1. Here we describe the construction of the reversible machine's Untidy Stage and Cleanup Stage quintuples corresponding to a typical quintuple

$$q_j, S_j \to S_k, \sigma, q_l$$

of the irreversible machine being simulated. The elements of the quintuple before the arrow represent, respectively, the initial control state and scanned tape symbol(s); those after the arrow represent the written tape symbol(s), shift, and final control state. In the Untidy Stage, each execution of such a quintuple should leave a record on the history tape sufficient to undo the operation in the subsequent Cleanup Stage. Since a deterministic Turing machine's quintuples have disjoint domains, it is sufficient to write $\langle i, j \rangle$ on the history tape. This can be achieved by replacing the above quintuple by the following quintuples:

(1) A nonshifting quintuple of the form

$$q_j, (S_j, b) \to (S_k, b), (0, 0), q'_{ij},$$

which executes the original quintuple's read/write operation while reading and writing blank on the history tape (second element in parentheses), and remembering i and j in a special control state;

(2) A quintuple of the form

$$q'_{ij}, X \to X, (\sigma, +), q''_{ij}$$

for each scanned-symbol combination X. Together these quintuples implement an oblivious shift, in which the history tape is right-shifted (+), while the other tapes perform the shift σ called for by the simulated computation. The memory of i and j is passed on via another special control state.

(3) A quintuple of the form

$$q''_{ij}, (Y, b) \rightarrow (Y, \langle i, j \rangle), (0, 0), q_l$$

for each scanned symbol combination Y on the nonhistory tapes. Together these quintuples implement a nonshifting read/write on the history tape, in which a blank is read and the information $\langle i, j \rangle$ representing the quintuple just executed is left in its place. Since each of these quintuples of types (1)-(3) has an inverse of the same form as itself, and since the history-taking and special starred control states guarantee disjointness of the quintuples' ranges, all operations of the Untidy Stage can be undone in the Cleanup Stage.

Theorem 1. We describe in more detail how the reversible machine causes operations to be performed in correct sequence, how it organizes the storage of the checkpoints on tape, and finally how it arrives at the satisfactory values for m and n without knowing T and S beforehand. We describe the case $k = 2$; generalization to larger k is straightforward. For the moment assuming m and n to be given, the hierarchical simulation is conveniently performed by a recursive routine such as RS(z,x,n,m,d) below, whose effect is to cause argument z, representing an ID of the simulated computation, to be incremented or decremented (according to whether argument d is +1 or −1) by the (m 2^n)th successor of argument x under the simulated computation's transition rule.

```
procedure RS(z,x,n,m,d)
 if (n=0)
  LINSIM(z,x,m,d)
 fi (n=0)
 if (n>0)
  RS(y,x,n-1,m,+1)
  RS(z,y,n-1,m,d)
  RS(y,x,n-1,m,-1)
 fi (n>0)
```

It should be noted that in reversible programs, the paths entering a merge point, like those leaving a branch point, must be labeled with mutually exclusive conditions. For example, in RS, the statement `fi (n=0)` denotes a merge point in which control may pass *from* the preceding block of code only if n=0. All assignments are done reversibly, presuming a knowledge of the previous value of the variable (initially zero).

All the actual simulation in RS above is done by the procedure LINSIM(z,x,m,d) below, which, according to the sign of d, increments or decrements argument z by the mth successor of argument x, computed by the method of Lemma 1, i.e., by creating, then clearing, an m-step history on a worktape reserved for that purpose.

```
procedure LINSIM(z,x,m,d)
 if (x is terminal or |x|>m-1)
  z:=z+d*x
 fi (x is terminal or |x|>m-1)
 if (x nonterminal and |x|<m)
  z:=z+d*[mth successor of x]
 fi (x nonterminal and |x|<m)
```

Note that LINSIM suspends forward computation, treating x as its own successor, when the computation being simulated terminates, or when the size of argument x exceeds the space bound m. This convention enables the main procedure, to be described later, to find the right segment size m $\approx S$ and recursion depth n $\approx \log(T/S)$.

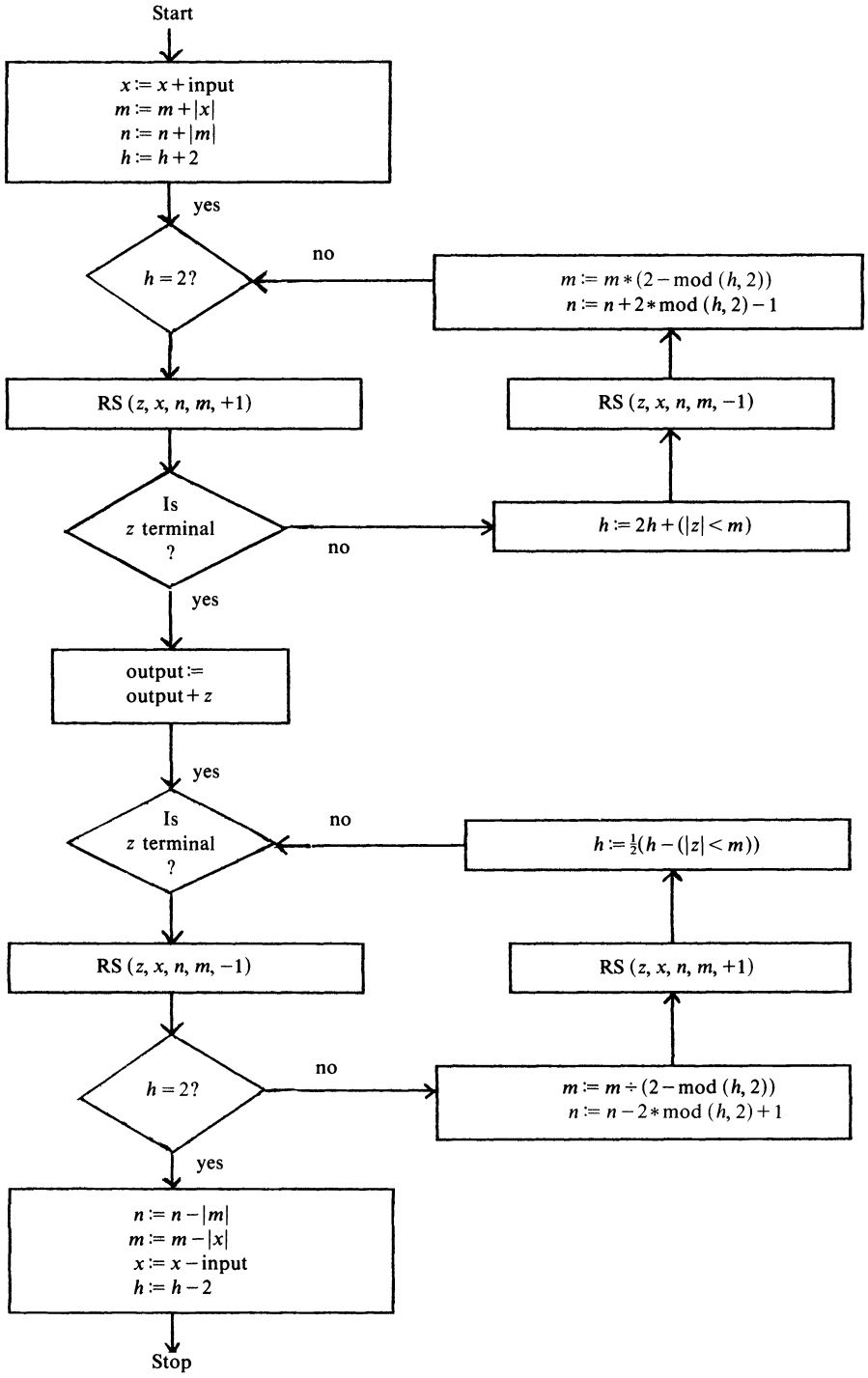


FIG. 1. MAIN *procedure for space-efficient reversible computation using recursive subroutine* "RS."

Proceeding now to the organization of data on tape, it can be seen that the procedure RS needs separate storage $O(\mathtt{m})$ at each level of recursion, for its arguments and for the local variable y. This storage can conveniently be organized as a sequence of n blocks of size $O(\mathtt{m})$ on a single "checkpoint" tape, demarcated by appropriate punctuation. Each call to RS would then involve reversibly passing arguments of size $O(\mathtt{m})$ back and forth between adjacent blocks, a job that can be done in time $O(\mathtt{m})$ if an extra blank tape is used as a buffer. The level of recursion would not have to be passed as an explicit argument, since that information would already be encoded by the head position on the checkpoint tape. At the 0 level of recursion, the linear time computation of each checkpoint from its predecessor would proceed using all the tapes of the original machine being simulated, plus an additional history tape.

We now go on to describe the main procedure, which tries successively larger m and n until the correct values are found, writes the final output (if the computation being simulated indeed terminates), and then reversibly disposes of all intermediate data generated in the course of the simulation. This procedure can be represented by a flow chart (Fig. 1). Here diamonds are used to represent both branch points and merge points since, as remarked before, it is necessary in reversible programming to label alternative paths into a merge point, like the paths out of a branch point, with mutually exclusive conditions. Note that the whole bottom half of the flow chart is devoted to undoing the work of the top half, so that all intermediate data are restored to their initial states after the desired output has been obtained.

In more detail, the main procedure begins by initializing m, n, and x, as well as a variable h, which will be used to encode the history of the search for correct m and n values. Each turn around the main loop in the upper half of the chart either increases n by one (thereby doubling the time allowance if it was exceeded before the space allowance), or doubles m and decreases n by one (thereby doubling the space allowance while keeping the time allowance fixed if the space allowance was exceeded first). In the former case a 0 is pushed into the least significant bit of h (regarded as a binary string); in the latter case a 1 is pushed. Since LINSIM simulates m steps of computation between space tests, the final m value may be less than S, but lies between $S/2$ and $2S$. The final n value is similarly equal to $\log(T/S)$ within a term of order unity. When the final m and n are found, the ID z generated by RS will be found to be terminal. It is then copied onto the output tape, and the entire computation up to that point is undone in the bottom half of the chart.

The space used by the algorithm is dominated by the $O(\mathtt{mn}) \leqq O(S \log(T))$ squares used on the checkpoint tape, all other terms being $O(S)$ or less. The time used in one turn around the main loop depends on m and n as $O(\mathtt{m} \cdot (2k-1)^{\mathtt{n}})$, from $O((2k-1)^{\mathtt{n}})$ recursive invocations of RS and $O((2k-1)^{\mathtt{n}})$ m-step linear-time simulations. Since n may be decreased at some turns around the loop (recall that when the space bound is exceeded, m is doubled and n is decreased by 1), the largest time cost is not necessarily associated with the last turn. Nevertheless, the total time cost of all the turns is bounded by a convergent geometric series

$$O\left((2k-1)^{\log T} \sum_i \left(\frac{k}{(2k-1)}\right)^i\right).$$

Executing the bottom half of the flow chart of course only introduces a factor of order unity, so the total time cost remains $O(T^{1+1/(\log k)})$ as claimed. □

Acknowledgments. I wish to thank Leonid Levin for urging and helping me to strengthen a previous, much weaker version of Theorem 1 and Martin Tompa for pointing out how to reduce the time exponent from 1.585 to $1+\varepsilon$.

Note added in proof. Robert Y. Levine and Alan T. Sherman, in a recent preprint, "A Note on Bennett's Time-Space Trade-off for Reversible Computation" (Department of Computer Science, Tufts University, Medford, Massachusetts 02155), give tighter time and space bounds for the reversible simulation as $\Theta(T^{1+\epsilon}/S^{\epsilon})$ and $\Theta(S \ln(T/S))$, respectively; and they point out that the space bound contains a constant factor diverging with small ϵ approximately as $\epsilon 2^{1/\epsilon}$.

REFERENCES

[1] C. H. BENNETT, *Logical reversibility of computation*, IBM J. Res. Develop., 17 (1973), pp. 525-532.
[2] ———, *The thermodynamics of computation—a review*, Internat. J. Theoret. Phys., 21 (1982), pp. 905-940.
[3] D. COPPERSMITH AND R. CLEVE, private communication.
[4] R. CLEVE, *Methodologies for designing block ciphers and cryptographic protocols*, Ph.D. thesis, Computer Science Department, University of Toronto, Toronto, Ontario, Canada, 1988.
[5] D. COPPERSMITH AND E. GROSSMAN, *Generators for certain alternating groups with applications to cryptography*, SIAM J. Appl. Math., 29 (1975), pp. 624-627.
[6] E. FREDKIN AND T. TOFFOLI, *Conservative Logic*, Internat. J. Theoret. Phys., 21 (1982), pp. 219-253.
[7] R. LANDAUER, *Irreversibility and heat generation in the computing process*, IBM J. Res. Develop., 5 (1961), pp. 183-191.
[8] L. LEVIN, *Universal sequential search problems*, Problems Inform. Transmission, 10 (1973), pp. 206-210.
[9] ———, *Randomness conservation inequalities; Information and independence in mathematical theories*, Inform. and Control, 61 (1984), pp. 15-37.
[10] J. MCCARTHY, *The inversion of functions defined by Turing machines*, in Automata Studies, C. E. Shannon and J. McCarthy, eds., Princeton University Press, Princeton, NJ, 1956.
[11] L. PRIESE, *On a simple combinatorial structure sufficient for sublying nontrivial self-reproduction*, J. Cybernetics, 6 (1976), pp. 101-137.
[12] T. TOFFOLI, "*Computation and construction universality of reversible cellular automata*, J. Comput. System Sci., 15 (1977), pp. 213-231.
[13] ———, *Reversible computing*, MIT Report MIT/LCS/TM-151, Mass. Inst. of Technology, Cambridge, MA, 1980.
[14] N. MARGOLUS, *Physics-like models of computation*, Phys. D., 10 (1984), pp. 81-95.
[15] J. VON NEUMANN, *Theory of Self-Reproducing Automata*, A. W. Burks, ed., University of Illinois Press, Urbana, IL, 1966.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 777-791, August 1989

011

ULTIMATE CHARACTERIZATIONS OF THE BURST RESPONSE OF AN INTERVAL SEARCHING ALGORITHM: A STUDY OF A FUNCTIONAL EQUATION*

PHILIPPE JACQUET† AND WOJCIECH SZPANKOWSKI‡

Abstract. The interval searching algorithm for broadcast communications of Gallager and Tsybakov and Mikhailov is analyzed. Ultimate characterizations of the burst response of the algorithm, that is, when the number of collided packets becomes large is presented. Three quantities are of interest: the *conflict resolution interval* (CRI); the fraction of the *resolved interval* (RI); and the number of *resolved packets* (RP). If n is the multiplicity of a conflict, then it is proved that the mth moments of CRI, RI, and RP are $O(\log^m n)$, $O(n^{-m})$, and $O(1)$, respectively. In addition, for the first two moments of these parameters precise asymptotic approximations are presented. The methodology proposed in this paper is applicable to asymptotic analysis of any problem that can be reduced to a solution of the *functional equation* $f(x)=2^s\cdot f(x/2)\cdot a(x)+b(x)$, where s is an integer and $a(x)$, $b(x)$ are given functions.

Key words. interval searching algorithms, analysis of algorithms, digital trees and other data structures, asymptotic analysis, functional equations, Mellin transform

AMS(MOS) subject classifications. primary 68Q25, 68M20; secondary 39B10

1. Introduction.

In a broadcast packet-switching network a number of distributed users share a common communication channel. Since the channel is the only means of communication among the users, packet collisions are inevitable if a central coordination is not provided. The problem is to find an efficient distributed algorithm for retransmitting conflicting packets. In recent years *conflict resolution algorithms* (CRA) [1], [2], [5]-[7], [9], [14]-[16] have become increasingly popular, mainly due to a nice stability property. The basic idea of CRA is to solve each conflict by splitting it into smaller conflicts (divide-and-conquer algorithm). This is possible if each user observes the channel and learns whether in the past it was idle, success, or collision transmissions. The partition of a conflict can be made on the basis of a random variable (flipping a coin) [2], [6], [7], [15] or on the basis of the time a user became active [2], [5], [9], [14], [16]. The former algorithm is known as the Capetanakis-Tsybakov-Mikhailov algorithm [2], [15] (the stack algorithm) while the latter as the Gallager-Tsybakov-Mikhailov algorithm [5], [16] known also as an interval-searching algorithm.

For interval searching algorithms, three parameters are of interest: the *conflict resolution interval* (CRI), the fraction of the *resolved interval* (RI), and the number of *resolved packets* (RP). For the initial multiplicity of a conflict n (i.e., n packets collide) the mth moments of the above parameters are denoted as T_n^m, W_n^m and C_n^m, respectively. We prove that $T_n^m=O(\log^m n)$, $W_n^m=O(n^{-m})$, and $C_n^m=O(1)$ for large n. However, for the first two moments, which are the most important from a practical viewpoint, we offer precise estimations. In particular, we show that asymptotically $W_n^1\sim a/(n+1)+P(\log n)$, $C_n^1\sim a+P(\log n)$, and $T_n^1\sim\log n+c+P(\log n)$, where a and c are constants that are determined in our Proposition, and $P(\log n)$ is a fluctuating

* Received by the editors September 28, 1987; accepted for publication (in revised form) November 29, 1988. This research was supported in part by National Science Foundation, grant NCR-8702115.

† INRIA, Rocquencourt, 78153 Le Chesnay, France. The research of this author was done primarily while he was visiting Purdue University.

‡ Department of Computer Science, Purdue University, West Lafayette, Indiana 47907.

function with a small amplitude. For the variance, we obtain $\operatorname{var}\mathbf{W}_n \sim a'/(n+1)^2$, $\operatorname{var}\mathbf{C}_n \sim a''$ and $\operatorname{var}\mathbf{T}_n \sim c'$, respectively. Our results are useful in characterizing the "burst response" of the algorithm. Indeed, it seems to be interesting to evaluate the ability of a communication protocol to resolve an initial collision of large multiplicity (e.g., a "bursty" access to the channel of several stations). It is known that the algorithm does not completely resolve the initial collision in one session, and it needs several sessions for successful transmissions of n initially collided packets. Our results show (for more details see also Remark (ii) at the end of § 2) that the algorithm needs on the average $n(\log_2 n - 1)/a + nc/a$ slots to resolve the initial collision of size n.

To the best of the authors' knowledge, previous research has been restricted only to the first moments of CRI, RI, and RP, however, the exact asymptotic expansions have not been achieved [14] (see also [9], [16]). The methodology presented in this paper can be easily extended to analyze other interval searching algorithms as well as some other data structures such as digital search trees [12]. Moreover, conflict resolution algorithms find applications in typical computer science problems such as semaphore conflicts and batch retrieval algorithms for databases [17], [18].

2. Problem statement and main results. Let us start with a short description of the Gallager-Tsybakov-Mikhailov algorithm with ternary feedback [5], [16]. Assume a channel is slotted and a slot duration is equal to a packet transmission time. The algorithm defined below allows the transmission of the packets on the basis of their generation times and we assume that packets are generated according to a Poisson point process with rate λ. Access to the channel is controlled by a window based on the current age of packets. This window will be referred to as the *enabled interval* (EI). Let s_i denote the starting point for the ith EI, and t_i is the corresponding starting point for the *conflict resolution interval* (CRI), where CRI represents the number of slots needed to resolve a collision. Initially, the enabled interval is set to be $[s_i, \min\{s_i+\tau, t_i\}]$, where τ is a constant that will be further optimized. At each step of the algorithm, we compute the endpoints of the EI based on the outcome of the channel. If at most one packet falls in the initial EI, then the conflict resolution interval ends immediately, and $s_{i+1} = s_i + \min\{\tau, t_i - s_i\}$. Otherwise, the EI is split into two halves, and three cases must be considered:

(i) All users whose current age of packets fall into the first (left) half are allowed to transmit packets. If it causes next *collision*, all knowledge about the second half is erased, and the first half is immediately split into two halves;

(ii) If enabling the first half causes an *idle* slot, the second half is immediately split into two halves;

(iii) If the first half gives a *success*, the entire second half is enabled.

A CRI that begins with a collision, continues until enabling the second half of some pairs gives a success.

Assume that an initial collision of a CRI is of multiplicity n, that is, n packets collide in the first slot of a CRI. Then all packets whose generation times fall into an interval (s_i, s_{i+1}) are successfully sent in the ith CRI. The interval (s_i, s_{i+1}) is called the ith *resolved interval* (RI) and its length is at most τ. The parameters of interest are: the length of a CRI, $\mathbf{T}$; the fraction of the resolved interval (i.e., the ratio of the resolved interval and τ), $\mathbf{W}$; and the number of resolved packets (successfully transmitted) in an enabled interval, $\mathbf{C}$. Let also $\mathbf{N}$ denote the multiplicity of a conflict. Then the mth conditional moments of the above quantities are defined as $T_n^m = E\{\mathbf{T}^m | \mathbf{N} = n\}$, $W_n^m = E\{\mathbf{W}^m | \mathbf{N} = n\}$, and $C^m = E\{\mathbf{C}^m | \mathbf{N} = n\}$, respectively. For $m = 0$, we define $T_n^0 = W_n^0 = C_n^0 = 1$. Then we have Theorem 1.

THEOREM 1. (i) *The mth conditional moment of* $\mathbf{T}$ *satisfies the following recurrence*:

$$T_0^m = T_1^m = 1,$$

$$(2.1)\quad (2^n-2)T_n^m = 2^n + n(2^m-2) + \sum_{i=1}^{m-1}\binom{m}{i}[T_n^i + n2^{m-i}T_{n-1}^i] + \sum_{j=2}^{n}\sum_{i=1}^{m-1}\binom{n}{j}\binom{m}{i}T_j^i + nT_{n-1}^m + \sum_{j=1}^{n-1}\binom{n}{j}T_j^m, \qquad n \geqq 2.$$

(ii) *The recurrence for* W_n^m *is*

$$W_0^m = W_1^m = 1,$$

$$(2.2)\quad (2^{n+m}-2)W_n^m = 1 + \sum_{j=1}^{m-1}\binom{n}{j}[W_n^i + nW_{n-1}^i] + nW_{n-1}^m + \sum_{j=1}^{n-1}\binom{n}{j}W_j^m, \qquad n \geqq 2.$$

(iii) *The recurrence for the mth conditional moment of* $\mathbf{C}$ *becomes*

$$C_0^m = 0, \qquad C_1^m = 1,$$

$$(2.3)\quad (2^n-2)C_n^m = n\sum_{j=1}^{m-1}\binom{m}{j}C_{n-1}^i + nC_{n-1}^m + \sum_{j=1}^{n-1}\binom{n}{j}C_j^m, \qquad n \geqq 2.$$

Proof. Note that an n-conflict is resolved by splitting n into two groups, and according to the algorithm description three basic cases must be considered: (1) $n \to (0, n)$; (2) $n \to (1, n-1)$; (3) $n \to (j, n-j)$, $j \geqq 2$. The probabilities of these cases are equal to 2^{-n}, $n2^{-n}$, and $\binom{n}{j}2^{-n}$, respectively. For example,

$$C_n^m = 2^{-n}E\{\mathbf{C}^m|\mathbf{N}=n\} + n2^{n-1}E\{(1+\mathbf{C})^m|\mathbf{N}=n-1\} + 2^{-n}\sum_{j=2}^{n}\binom{n}{j}E\{\mathbf{C}^m|\mathbf{N}=j\}.$$

After some algebraic manipulations, we prove the recurrences (2.1), (2.2), and (2.3). □

We now give a summary of the main results, delaying more complicated proofs to the next two sections. Let $W_m(x)$, $C_m(x)$, and $T_m(x)$ represent the exponential generating functions of W_n^m, C_n^m, and T_n^m, respectively, e.g., $T_m(x) = \sum_{n=0}^{\infty} T_n^m x^n/n!$. It turns out that to analyze recurrences (2.1)-(2.3), it is more convenient to deal with modified generating functions defined as $w_m(x) = W_m(x)\, e^{-x}$, $c_m(x) = C_m(x)\, e^{-x}$, and $t_m(x) = T_m(x)\, e^{-x} - 1$. Recurrences (2.1)-(2.3) can be transformed into functional equations for $w_m(x)$, $c_m(x)$, and $t_m(x)$. For example, from Theorem 1 we can easily prove that the generating functions $w_1(x)$, $c_1(x)$, and $t_1(x)$ satisfy the following functional equations:

$$w_1(x) = 2^{-1}[1 + (1+x/2)\, e^{-x/2}]w_1(x/2),$$

$$c_1(x) = [1 + (1+x/2)\, e^{-x/2}]c_1(x),$$

$$t_1(x) = [1 + (1+x/2)\, e^{-x/2}]t_1(x/2) + 1 - (1+3x/2)\, e^{-x} + 0.5x\, e^{-x/2}.$$

Note the generic form for these functional equations is

$$(2.4)\qquad f(x) = 2^s f(x/2)a(x) + b(x),$$

where s is an integer, $a(x) = \frac{1}{2}[1 + (1+x/2)\, e^{-x/2}]$, and the nonhomogeneous term $b(x)$ depends on the particular recurrence (see §§ 3 and 4). The general solution for (2.4) is given in the next lemma.

LEMMA 1. (i) *Functional equation* (2.4) *possesses the following solution*:

$$(2.5)\qquad f(x) = f^*(x) + \sum_{n=0}^{\infty} 2^{sn} b(x2^{-n}) \prod_{k=0}^{n-1} a(x2^{-k}),$$

where

$$f^*(x) = \lim_{n\to\infty} 2^{sn} f(x2^{-n}) \prod_{k=0}^{n-1} a(x2^{-k}) \tag{2.6}$$

assuming $f^*(x)$ exists and the series in (2.5) *is convergent.*

(ii) *Let $w(x) = \prod_{k=0}^{\infty} a(x2^{-k})$ exist and $f(0) = f'(0) = \cdots = f^{(s-1)} = 0$, $f^{(s)}(0) \neq 0$. Then*

$$f^*(x) = \begin{cases} 0 & s < 0, \\ x^s w(x) f^{(s)}(0)/s! & s \geqq 0, \end{cases} \tag{2.7}$$

where $f^{(s)}(0)$ denotes the sth derivative of $f(x)$ at $x = 0$.

(iii) *If $a(0) < 2$ and $b(x) = O(x^{s+1})$ for $x \to 0$, then the series in* (2.5) *is convergent.*

Proof. Part (i) follows immediately from formal iteration of (2.4). For (ii) assume first $s \geqq 0$. Let $u = x2^{-n}$; then

$$f^*(x) = x^s w(x) \lim_{u\to 0} \frac{f(u)}{u^s}. \tag{2.8}$$

Applying l'Hopital rule s-times, we prove (2.7). The case $s < 0$ is easy and left to the reader. For part (iii), we use D'Alembert's criterion, that is, to prove that $\sum_{n\geqq 0} \alpha_n$ is convergent, it is sufficient to show that $\lim_{n\to\infty} \alpha_{n+1}/\alpha_n < 1$. In our case, assuming $u = x2^{-n}$,

$$\lim \frac{\alpha_{n+1}}{\alpha_n} = a(0)2^s \lim_{u\to 0} \frac{b(u/2)}{b(u)} = \frac{a(0)}{2} < 1,$$

where the last equality follows from the assumption $b(x) = O(x^{s+1})$ for $x \to 0$. □

Direct use of Lemma 1 leads to very complicated formulas. Nevertheless, based on the lemma and using the Mellin transform, we can reduce these formulas to very simple asymptotics, as summarized in the following proposition.

PROPOSITION. *For $x \to \infty$ the following hold.*

(i) *The first two modified generating functions for the resolved interval are*

$$w_1(x) \sim \frac{a}{x} + \frac{a}{x} P(\log x), \tag{2.9}$$

$$w_2(x) \sim \frac{a(2+b)}{x^2} + P(\log x) \cdot \frac{a}{x^2}, \tag{2.10}$$

where $P(\log x)$ is a fluctuating function with a very small amplitude, and

$$a = \exp[\alpha/\log 2 + \log 2/2] \approx 2.505, \qquad b = \beta/\log 2 \approx 1.692, \tag{2.11}$$

where

$$\alpha = \int_0^\infty \frac{x e^{-x} \log x}{1+(1+x)e^{-x}}\, dx \approx 1.496, \qquad \beta = \int_0^\infty \frac{e^{-x/2}}{1+(1+x/2)e^{-x/2}}\, dx \approx 1.17. \tag{2.12}$$

(ii) *The modified generating functions for the number of resolved packets become*

$$c_1(x) = x w_1(x) \sim a + aP(\log x), \tag{2.13}$$

$$c_2(x) \sim a(1+b) + aP(\log x). \tag{2.14}$$

(iii) *The first two modified generating functions for the conflict resolution interval satisfy*

$$t_1(x) \sim \log_2 x + c + aP(\log x), \tag{2.15}$$

$$t_2(x) \sim \log_2^2 x + 2c \log_2 x + d + aP(\log x), \tag{2.16}$$

where

$$c = 2 - \frac{\mu a}{(\log 2)^2} \approx 4.149, \qquad d = \frac{5}{6} + 3(c-1) - c^2 - \frac{a\nu}{(\log 2)^2} \approx 17.169, \tag{2.17}$$

and

$$\mu = \int_0^\infty f_1(x) \frac{\log x}{x}\, dx \approx -0.41, \qquad \nu = -\frac{1}{2}\int_0^\infty f_2(x) \log^2 x\, dx \approx -1.484, \tag{2.18}$$

where the functions $f_1(x)$ and $f_2(x)$ depend on $a(x)$, $b(x)$, and $w_1(x)$; they are given in the next sections. □

The next stage consists of translating the expansions of $T_m(x)$, $W_m(x)$, and $C_m(x)$ for $x \to \infty$ into information about the asymptotics of their coefficients. That is, if $F(x) = f(x)\, e^x$ is the exponential generating function and $f(x) = \sum_{n=0}^\infty f_n x^n / n!\, e^{-x}$, then by the Cauchy formula [8] the coefficient f_n is given by

$$f_n = \frac{n!}{2\pi i} \oint f(x)\, e^x \frac{dx}{x^{n+1}}, \tag{2.19}$$

where the integration is done on the circle with center 0 and radius n. It should be stressed that $f(x) \sim g(x)$ does not necessarily imply $f_n \sim g_n$. Note, however, that the leading factor in the asymptotics for the generating functions of the quantities of interest is of the form $x^p (\log x)^q$ for some p and q. The next lemma establishes conditions under which this factor can be transformed into information on the coefficients (see also [11]).

LEMMA 2. *Let $f(x) = F(x)\, e^{-x}$ and S_θ be a cone $S_\theta = \{x: |\arg x| < \theta, 0 < \theta < \pi/2\}$. If for x in the cone, that is, for $x \in S_\theta$, and $x \to \infty$*

$$f(x) \sim x^p (\log x)^q \tag{2.20}$$

for some p and q, and for x outside the cone S_θ

$$|F(x)| < B\, e^{\alpha |z|} \tag{2.21}$$

for some B and $0 < \alpha < 1$, then for large n the coefficient f_n of $F(x)$ asymptotically satisfies

$$f_n = f(n) \cdot \left\{ 1 + O\left(\frac{1}{\sqrt{n} \log n} \right) \right\} = n^p (\log n)^q + O(n^{p-1/2} \log^{q-1} n). \tag{2.22}$$

Proof. The coefficient f_n is computed by Cauchy formula (2.19), where the integration is done along the circle $|x| = n$. Two cases are considered: $x \in S_\theta$ and x outside the cone S_θ. In the latter case, using (2.21) and Stirling's formula [7] we find that f_n defined in (2.19) can be upper bounded as $|f_n| < B\, e^{-(1-\alpha)n}$, that is, for large n the contribution of the integral outside the cone S_θ is negligible.

Now assume that $x \in S_\theta$. Using substitution $x = n\, e^{iv}$ in (2.19) and applying again Stirling's formula, one finds $f_n = I_n[1 + O(n^{-1})] + O(e^{-(1-\alpha)n})$, where

$$I_n = \sqrt{\frac{n}{2\pi}} \int_{-\theta}^{\theta} f(n\, e^{iv})\, e^{ne^{iv}}\, e^{-inv}\, dv. \tag{2.23}$$

The last integral can be further simplified by developing e^{iv} and the change of variable $y = \sqrt{n}\, v$. Then

$$I_n = \frac{1}{\sqrt{2\pi}} \int_{-\theta\sqrt{n}}^{\theta\sqrt{n}} f(n\, e^{iy\sqrt{n}})\, e^{n^{i\sqrt{n}}}\, e^{-i\sqrt{n}y}\, dy. \tag{2.24}$$

But the following is easy to establish: $f(n\,e^{iy\sqrt{n}})/f_n \to 1$, and $e^{ne^{iy\sqrt{n}}}\,e^{-i\sqrt{n}y} \to e^{-y^2/2}$ as $n \to \infty$. Then, by the *Lebesgue Dominated Convergence* theorem we show that

$$\frac{I_n}{f(n)} \to \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-y^2/2} = 1.$$

Finally, we note that a refinement in the evaluation of the order of convergence in the latter derivation leads to $f_n/f(n) = 1 + O(n^{-1/2}\log^{-1} n)$. This readily gives (2.22). □

Now we are prepared to state our second main result. Direct application of Lemma 2 to Proposition leads to Theorem 2.

THEOREM 2. *For large n the following hold*:

(i) *The fraction of the resolved interval becomes*

$$W_n^1 = \frac{a}{n+1} + \frac{1}{n+1} P(\log n) + O(n^{-3/2}) \approx \frac{2.505}{n+1}, \tag{2.25}$$

$$\begin{aligned} \operatorname{var} \mathbf{W}_n &= \frac{a(2+b)}{(n+1)(n+2)} - \frac{a^2}{(n+1)^2} + \frac{1}{(n+1)^2} P_1(\log n) + O(n^{-3/2}) \\ &\approx \frac{a(2+b)-a^2}{(n+1)^2} + \frac{1}{(n+1)^2} P_1(\log n) + O(n^{-3/2}) \approx \frac{2.97}{(n+1)^2}, \end{aligned} \tag{2.26}$$

$$W_n^m = O(n^{-m}). \tag{2.27}$$

(ii) *The number of resolved packets satisfies*

$$C_n^1 = a + P(\log n) + O(n^{-1/2}) \approx 2.505, \tag{2.28}$$

$$\operatorname{var} \mathbf{C}_n = a + ab - a^2 + P_2(\log n) + O(n^{-1/2}) \approx 0.47, \tag{2.29}$$

$$C_n^m = O(1). \tag{2.30}$$

(iii) *The conflict resolution interval can be represented as*

$$T_n^1 = \log_2 n + c + P(\log n) + O(n^{-1/2}) \approx \log_2 n + 4.144, \tag{2.31}$$

$$\operatorname{var} \mathbf{T}_n = \frac{5}{6} + 3(c-1) - c^2 - \frac{a\nu}{(\log 2)^2} + P_3(\log n) + O(n^{-1/2}) \approx 4.7, \tag{2.32}$$

$$T_n^m = O(\log^m m). \tag{2.33}$$

Remarks. (i) *Maximum throughput for the algorithm.* Let $x = \lambda\tau$, where λ is the input rate from the Poisson arrival process, and τ is the algorithm parameter defined in § 2. The *maximum throughput*, $\lambda_{\max}$, is defined as the maximum rate of successful transmissions that assures that the average packet delay is finite. In other words, for $\lambda < \lambda_{\max}$ the algorithm is stable. But, the throughput is also the ratio of the average number of successful transmissions to the average conflict resolution length, hence (see [1], [16])

$$\lambda_{\max} = \sup_x \frac{C_1(x)}{T_1(x)} = \sup_x \frac{x W_1(x)}{T_1(x)}. \tag{2.34}$$

It is easy to compute $\lambda_{\max}$ numerically for some values of x, and taking the first eight or nine values of T_n^1 and W_n^1 reveals $\lambda_{\max} = 0.48771$ for $x = 1.26$. This comes from the

fact that the maximum in (2.34) occurs for small values of x. Thus, from the numerical point of view, there is nothing to add. From the mathematical viewpoint, however, there is some interest in finding an analytical solution for $\lambda_{\max}$, e.g., to show whether some maximizing x is a global or a local maximum. Using the Proposition, we immediately see that for large x

$$\lambda_{\max} \sim \sup_x \frac{a}{\log_2 x + c}$$

and $\lambda_{\max}$ is a decreasing function of x. Thus, $x_{\text{op}} = 1.26$ is a global one.

(ii) *Average number of slots needed to resolve the initial collision.* Let us assume that one injects n packets into the system. How long does it take on the average to resolve completely this initial collision? It is well known that the Gallager-Tsybakov-Mikhailov algorithm does not solve the entire collision in one session. By Theorem 2(ii) we know that for large initial multiplicity n of a collision, only a packets are transmitted on the average in a conflict resolution session. Thus, n/a sessions are necessary to resolve the contention. According to Theorem 2(iii) each session with initial multiplicity n requires $\log_2 n + c$ slots. Therefore, on the average the algorithm with the initial multiplicity n needs $c + \log_2 n + c + \log_2 (n-a) + c + \log_2 (n-2a) + \cdots$ slots to resolve the conflict. This is roughly equal to $n/a \times (\log_2 n - 1) + nc/a$ slots. This can be compared with $\alpha \cdot n$ slots required to solve initial conflict for stack algorithms [2], [7], where α is a parameter of the system.

(iii) *Non-Poisson arrival processes and limiting distribution.* The reason why the asymptotic behavior of T_n^1 and W_n^1 is not essential for the algorithm follows from the fact that the probability of more than four packets collide is less than 0.01 if x is near the optimal value. This, however, holds only for the Poisson assumption model. If this assumption is dropped and the input process is more biased (e.g., multimodal or the peak of the distribution is for large values), then the asymptotic behavior, i.e., burst response, becomes more important. Although in general our recurrences (2.1)-(2.3) do not work for non-Poisson models, we can slightly change the algorithm (this change, anyhow, is required in practice) to assure that the recurrences are valid. Namely, in a non-Poisson model we spread randomly over a window all packets that fall into the window. Then, for such a randomized algorithm our recurrences hold since the binomial distribution of packets in each half of a window is preserved. Moreover, note that for such a randomized non-Poisson model limiting distributions for T_n^1 play a role. From our analysis (Theorem 2(iii)) we know that T_n^1 is *not* normally distributed, since $T_n^1 = O(\log n)$ and $\operatorname{var} T_n = O(1)$ (see [11]). At last, the burst response becomes increasingly important for extreme behaviors of the algorithm (even within the Poisson model) since in such situations the Poisson assumption is no longer true. In other words, the average behavior is well described by the throughput and the delay of the algorithm, but for dynamic behavior we need to study burst response.

(iv) *Other applications of our analysis.* Finally, we note that the interval searching algorithm motivated our study of a general functional equation of the form $f(x) = 2^s f(x/2) \cdot a(x) + b(x)$ (see Eq. (2.4)). In particular, we have investigated the asymptotic behavior of the coefficient f_n of the Taylor expansion of $f(x)$ (see (2.19)). Such a general approach has its own advantage in the fact that a class of problems can be solved in a unified manner. For example, noting that the conflict resolution session T_n can be represented as a path in an appropriate digital tree [16] we can easily apply our analysis to study some other properties of digital search trees. Note also that our methodology enables us to compute the asymptotics *without* exact solution of the recurrences (2.1)-(2.3). For example, the following recurrence is often met in the

analysis of digital search trees (radix trees, Patricia trees, etc. [12], [13]):

$$x_n = a_n + \sum_{k=0}^{n} \binom{n}{k} 2^{-n} (l_{n-k} x_k + r_k x_{n-k}), \tag{2.35}$$

where x_n is the numerical value of a given property of the tree, a_n is the amount of the property possessed by the root, and l_k, r_k are weights of the left and right subtrees. For instance, if $l_k = r_k = 1$, then (2.35) models (for appropriate a_n) the successful search time (all moments) of radix tries and Patricia tries [13]; if $l_k = r_k = 1 - \delta_{n,k}$, where $\delta_{n,k}$ is the Kronecker delta, then (2.35) represents the unsuccessful search in a Patricia tree. For the Gallager algorithm we have to assume $l_k = \frac{1}{2}$ and $r_k = \frac{1}{2}(1 + 2\delta_{n-1,k})$. Introduction of the general (additive) term a_n in (2.35) enables to study a class of properties instead of a particular one. For example, another version of Gallager's algorithm, namely Berger's protocol [2], is analyzed in exactly the same manner except that for T_n^1 the additive term is equal to $a_n = 2 + n2^{-n} - 2^{-n}$ instead of $a_n = 1 - 2^{-n}$.

3. Analysis of the first moments. In this section we concentrate on derivations of the asymptotics for W_n^1, C_n^1, and T_n^1. From the methodological viewpoint, we mention here the crucial role of the generating function $W_1(x)$ of W_n^1. This is a consequence of the fact that $w_1(x) = W_1(x)\, e^{-x}$ satisfies homogeneous functional equation of type (2.4), that is, $b(x) \equiv 0$ in (2.4). We shall see that all other generating functions, which satisfy nonhomogeneous functional equation (2.4), can be expressed in terms of $w_1(x)$. In the derivations we extensively use the Mellin transform. A good reference for properties of the Mellin transform is [4] and [3]. Some details of the derivations can also be found in [10].

3.1. The fraction of the resolved interval. The first moment of the fraction of resolved interval W_n^1 satisfies recurrence (2.2) with $m = 1$. Let $W_1(x)$ be the exponential generating function of W_n^1, and define $w_1(x) = W_1(x)\, e^{-x}$. Multiplying both sides of (2.2) by $x^n/n!$ and summing, we find

$$w_1(x) = \tfrac{1}{2}[1 + (1 + x/2)\, e^{-x/2}] w_1(x/2). \tag{3.1}$$

This functional equation falls into (2.4) with $b(x) \equiv 0$ (homogeneous equation) and $a(x) = \frac{1}{2}[1 + (1 + x/2)\, e^{-x/2}]$. Let us also define $a_1(x) = a(2x)$. Then, by Lemma 1 and $w_1(0) = 1$ the solution of (3.1) is

$$w_1(x) = \prod_{k=1}^{\infty} a_1(x2^{-k}) = \prod_{k=0}^{\infty} a(x2^{-k}). \tag{3.2}$$

Let $l(x) = \log w_1(x)$ and $g_1(x) = \log a_1(x)$. Then (3.2) becomes

$$l(x) = \sum_{k=1}^{\infty} g_1(x2^{-k}). \tag{3.3}$$

Equation (3.3) is used to derive asymptotics for $l(x)$ for $x \to \infty$. At this stage, we start plunging into complex analysis, that is, into the Mellin transform. First we give an overview of the methodology. We use the following facts about Mellin transform [4], [3].

PROPERTY 1. If $f(x)$ is piecewise continuous on $[0, \infty]$, and

$$f(x) = O(x^{\alpha}), \quad x \to 0, \qquad f(x) = O(x^{\beta}), \quad x \to \infty, \tag{3.4a}$$

then the Mellin transform of $f(x)$ defined as [4], [3]

$$f^*(s) = \int_0^{\infty} f(x) x^{s-1}\, dx \tag{3.4b}$$

exists in the *fundamental strip*

$$-\alpha < \operatorname{Re} s < -\beta.$$

PROPERTY 2. A *harmonic sum* of the form

$$F(x) = \sum_{k=0}^{\infty} f(\mu_k x) \tag{3.5a}$$

has a simple Mellin transform:

$$F^*(s) = f^*(s) \sum_{k=0}^{\infty} \mu_k^{-s}. \tag{3.5b}$$

PROPERTY 3. The following holds for $x \to \infty$.

$$f(x) \sim -\sum_{-\beta < \operatorname{Re}(s_k) < M} \operatorname{Res}\{f^*(s)x^{-s};\ s = s_k\} + O(x^{-M}) \tag{3.6a}$$

where s_k are the singularities of $f^*(s)$ lying in the strip $-\beta < \operatorname{Re}(s) < M$, and M is an arbitrary large number, and $\operatorname{Res}\{f(s);\ \delta = s_k\}$ is the residue of $f(s)$ at $s = s_k$.

PROPERTY 4. If for $x \to \infty$ the function $f(x)$ satisfies $f(x) \sim dx^{\beta}$, then

$$f^*(s) \sim -\frac{d}{s+\beta}, \qquad s \to -\beta. \tag{3.6b}$$

□

The plan for dealing with a sum such as (3.3) follows: Noting that (3.3) falls into the harmonic sum (3.5a) in Property 2, we first compute the Mellin $l^*(s)$ transform of $l(x)$ as suggested in (3.5b). The fundamental strip of $l^*(s)$ is defined in Property 1. Then, by Property 4 we find the asymptotic expansion for $l^*(s)$, and by the inversion formula (3.6a) in Property 3 we determine asymptotics for $l(x)$.

Note that (3.3) is of the form (3.5a), and hence for $\operatorname{Re} s < 0$ the Mellin transform of (3.3) is

$$l^*(s) = \frac{2^s}{1-2^s} g_1^*(s). \tag{3.7}$$

But $g_1(x) = O(x^2)$ for $x \to 0$ and $g_1(x) = -\log 2$ for $x \to \infty$. Hence, $g_1^*(s)$ exists in the strip $-2 < \operatorname{Re} s < 0$. Note also that by (3.6b) $g_1^*(s)$ has a pole at $s = 0$ and $g_1^*(s) \sim \log 2/s$. But, the first factor in (3.7), that is, $2^s/(1-2^s)$, has poles at $\chi_k = 2\pi ik/\log 2$, $k = 0, \pm 1, \cdots$. The singularity at $k = 0$, i.e., $\chi_0 = 0$ is the most difficult to treat, since it is a double pole, and it determines the leading component of the asymptotics [4], [12]. Let us first consider $k = 0$. Note that

$$\frac{2^s}{1-2^s} = -\frac{1}{\log 2} \cdot \frac{1}{s} - \frac{1}{2} + O(s), \tag{3.8a}$$

$$g_1^*(s) = \frac{\log 2}{s} + \alpha + O(s). \tag{3.8b}$$

To determine the constant α we integrate $g_1^*(s)$ by parts, and hence

$$g_1^*(s) = \int_0^{\infty} \log \frac{1}{2}[1 + (1+x)\, e^{-x}] x^{s-1}\, dx = \frac{1}{s} \int_0^{\infty} \frac{x\, e^{-x}}{1 + (1+x)\, e^{-x}} x^s\, dx. \tag{3.9}$$

Using $x^s = 1 + s \log x + O(s^2)$, we immediately find

$$\alpha = \int_0^{\infty} \frac{x\, e^{-x} \log x}{1 + (1+x)\, e^{-x}}\, dx \tag{3.10}$$

or in another form

$$\alpha = \sum_{n=0}^{\infty} (-1)^n \int_0^{\infty} x\, e^{-x}(1+x)^n\, e^{-nx} \log x\, dx$$

$$= \sum_{n=0}^{\infty} (-1)^n \sum_{k=0}^{n} \binom{n}{k} \frac{(k+1)!}{(n+1)^{k+2}} [H_{k+1} - \gamma - \log(n+1)]. \tag{3.11}$$

As a side effect, we also prove that

$$\int_0^{\infty} \frac{x\, e^{-x}}{1+(1+x)\, e^{-x}} = \log 2. \tag{3.12}$$

The rest is easy. Multiplying (3.8a) and (3.8b) and taking into account additional simple poles at $s = \chi_k = 2\pi ik/\log 2$, $k \neq 0$, we obtain

$$l^*(s) = -\frac{1}{s^2} - \left(\frac{\alpha}{\log 2} + \frac{\log 2}{2}\right) \cdot \frac{1}{s} + \sum_{\substack{k=-\infty \\ k\neq 0}}^{\infty} \frac{g_1^*(s)}{s-\chi_k} + O(1). \tag{3.13}$$

Property 3 applied to (3.13) implies [4], [7]

$$l(x) = -\log x + \left(\frac{\alpha}{\log 2} + \frac{\log 2}{2}\right) + P(\log x) + O(x^{-M}) \tag{3.14}$$

where M is a large positive constant, and

$$P(u) = \frac{1}{\log 2} \sum_{\substack{k=-\infty \\ k\neq 0}}^{\infty} g_1^*\left(\frac{2\pi ik}{\log 2}\right) \exp\left[-\frac{2\pi iku}{\log 2}\right]. \tag{3.15}$$

The series in (3.15) converges since $g_1^*(iy) = O(y^{-M})$ for $y \to \infty$. Note also that the fluctuating function $P(\log x)$ can be safely ignored in practice, since it has a very small amplitude. Indeed, this follows from the fact that $g_1^*(s)$ can be developed as a factorization of $\Gamma(s)$ and $\Gamma(2\pi i/\log 2) \approx 10^{-6}$. Finally, (3.14) implies

$$w_1(x) = \exp[l(x)] \sim \frac{a}{x} + \frac{a}{x} P(\log x), \tag{3.16}$$

where $a = \exp[\alpha/\log 2 + \log 2/2]$.

This proves (2.9) of Proposition (i). To estimate coefficient W_n^1 of $W_1(x)$, we apply Lemma 2, and for this we must prove that $|W_1(x)| < e^{\alpha|z|}$ holds for $\alpha \in (0,1)$ outside the cone S_θ. Since $W_1(x) = \frac{1}{2}(e^{x/2} + 1 + (x/2))\, W_1(x/2)$, hence $|W_1(x)| < e^{|x|/2} |W_1(x/2)|$. This implies $W_1(x) = (1 + O(x^2))\, e^x$, and for $x \in S_\theta$ $|W_1(x)| = (1 + O(x^2))\, e^{\alpha|x|}$, where $\alpha = \cos\theta$. Thus, $|W_1(x)| < e^{(\alpha+\varepsilon)|z|}$ and condition (2.21) in Lemma 2 holds. Thus, with (3.16) and (2.22) this proves (2.25) in Theorem 2(i).

3.2. The number of resolved packets. The first moment of the number of resolved packets, C_n^1, satisfies recurrence (2.3) with $m = 1$. Then, after simple algebra, we check that the modified generating function $c_1(x) = C_1(x)\, e^{-x}$ satisfies the functional equation

$$c_1(x) = [1 + (1 + x/2)\, e^{-x/2}] c_1(x/2) = 2a(x) c_1(x/2). \tag{3.17}$$

Let $c_1(x) = xF_1(x)$ for some function $F_1(x)$. Then (3.17) is transformed into $F_1(x) = a(x)F_1(x/2)$, which is exactly the same as the functional equation for $w_1(x)$ (see (3.17)). Thus, $c_1(x) = xW_1(x)$ and $C_n^1 = nW_{n-1}^1$. Proposition (ii), (2.13), and (2.28) of Theorem 2(ii) follow immediately.

3.3. The conflict resolution interval. The recurrence for T_n^1 is given in (2.1) with $m=1$. Let $T_1(x)$ be the exponential generating function for T_n^1, and define $t_1(x)=T_1(x)\,e^{-x}-1$. Then $t_1(x)$ satisfies recurrence

$$\begin{aligned} t_1(x) &= \left[1+\left(1+\frac{x}{2}\right)e^{-x/2}\right]t_1\frac{x}{2}+1-\left(1+\frac{3}{2}x\right)e^{-x}+\frac{x}{2}e^{-x/2} \\ &= 2a(x)t_1\frac{x}{2}+b(x) \end{aligned} \tag{3.18}$$

where $a(x)=\frac{1}{2}[1+(1+x/2)\,e^{-x/2}]$ and

$$b(x)=1-\left(1+\frac{3}{2}x\right)e^{-x}+\frac{x}{2}e^{-x/2}. \tag{3.19}$$

The recurrence (3.18) falls into (2.4) and by Lemma 1 it has solution (2.5). Note that our definition of $t_1(x)$ implies that $t_1(0)=0$; thus the solution of (3.18) is

$$t_1(x)=\sum_{n=0}^{\infty}2^n b(x2^{-n})\prod_{k=0}^{n-1}a(x2^{-k}). \tag{3.20}$$

The series is convergent since $b(x)=O(x^2)$ for $x\to 0$, as required in Lemma 1(iii). Define

$$w_1(x)=\prod_{k=0}^{\infty}a(x2^{-k}), \tag{3.21}$$

which falls into our solution of the homogeneous equation (3.2) on the fraction of the resolved interval. Noting that $\prod_{k=0}^{n-1}a(x2^{-k})=w_1(x)/w_1(x2^{-n})$ we transform (3.21) into

$$\frac{t_1(x)}{w_1(x)}=\sum_{n=0}^{\infty}2^n\frac{b(x2^{-n})}{w_1(x2^{-n})}. \tag{3.22}$$

Finally, defining

$$Q_1(x)=\frac{t_1(x)}{xw_1(x)},\qquad q_1(x)=\frac{b(x)}{xw_1(x)} \tag{3.23}$$

we obtain from (3.22)

$$Q_1(x)=\sum_{n=0}^{\infty}q_1(x2^{-n}). \tag{3.24}$$

This falls into our general harmonic sum discussed in Property 2, and by (3.5), the Mellin transform of (3.24) is

$$Q_1^*(s)=\frac{q_1^*(s)}{1-2^s} \tag{3.25}$$

for $-1<\operatorname{Re} s<0$. Unfortunately, straightforward application of the same trick as in § 3.1 does not work here, since the appropriate integrals do not exist. This follows from the fact that $w_1(x)$ involved in the definition of $q_1(x)$ has infinity many poles for $\operatorname{Re} s=0$. To avoid this problem, we define a new function:

$$f_1(x)=q_1(x)-q_1\left(\frac{x}{2}\right)=\frac{b(x)-2b(x/2)a(x)}{xw_1(x)}. \tag{3.26}$$

Note now that $f_1(x) = O(e^{-x})$ for $x \to \infty$ and the Mellin transform $f_1^*(s)$ exists in the strip $-1 < \operatorname{Re} s < \infty$. For, $f_1^*(s) = q_1(s)/(1-2^s)$, (3.25) becomes

$$Q_1^*(s) = \frac{f_1^*(s)}{(1-2^s)^2}. \tag{3.27}$$

But $f_1^*(s)$ is analytical at $s=0$, and hence $f_1^*(0) = f_1^*$ and $(d/ds)f^*(s)|_{s=0} = \mu$ exist. Trivial computations show

$$f_1^* = \int_0^\infty f_1(x)\frac{dx}{x}, \qquad \mu = \int_0^\infty f_1(x)\frac{\log x}{x}\,dx. \tag{3.28}$$

The second integral must be evaluated numerically, while the first reduces to

$$\begin{aligned} f_1^* &= \int_0^\infty \left[\frac{q_1(x)}{x} - \frac{q_1(x/2)}{x}\right] dx = \lim_{k\to\infty} \int_0^{2^{k+1}} \frac{q_1(x) - q_1(x/2)}{x}\,dx \\ &= \lim_{k\to\infty} \int_{2^k}^{2^{k+1}} \frac{q_1(x)}{x}\,dx = \frac{\log 2}{a} \end{aligned} \tag{3.29}$$

where the last equality follows from (3.14) and (3.23), noting that $\lim_{x\to\infty} q_1(x) = 1/a$.

The rest is really simple and standard. Using the following expansions:

$$\frac{1}{(1-2^s)^2} = \frac{1}{s^2(\log 2)^2} - \frac{1}{s\log 2} + \mathrm{O}(1),$$

$$f_1^*(s) = \frac{\log 2}{a} + s\mu + O(s^2),$$

we obtain:

$$Q_1^*(s) = \frac{1}{a\log 2}\cdot\frac{1}{s^2} - \frac{1}{s}\left[\frac{1}{a} - \frac{1}{(\log 2)^2}\right] + O(1). \tag{3.30}$$

Taking additional poles of $(1-2^s)^2$ at $\chi_k = 2\pi ik/\log 2$, $k \neq 0$, and translating (3.30) into $Q_1(x)$ through Property 3, we find for $x \to \infty$ that

$$Q_1(x) = \frac{\log x}{a\log 2} + \left(\frac{1}{a} - \frac{\mu}{\log^2 2}\right) + P(\log x).$$

Finally, definition (3.23) of $Q_1(x)$ and estimation (3.14) for $w_1(x)$ imply that for $x \to \infty$

$$T_1(x)\, e^{-x} \sim \log_2 x + 2 - \frac{\mu a}{(\log 2)^2} + aP(\log x) \tag{3.31}$$

which proves formula (2.15) of Proposition (iii). Application of Lemma 2 (in the same manner as for $w_1(x)$) to (3.31) establishes (2.31) in Theorem 2(iii).

4. Analysis of variances and higher moments. In the previous section, we have established methodology for studying nonhomogeneous functional equations of type (2.4). It consists of two parts: first the solution to a homogeneous equation is found, then using it and properties of the Mellin transform, we obtain asymptotics for the nonhomogeneous one as was done, for example, in § 3.3. Finally, we appeal to Lemma 2 to establish asymptotics for the coefficients of the generating function. The same plan is adopted in this section, however, this time we present only sketches of proofs. For more details see [10].

4.1. The fraction of the resolved interval. Let us compute first the second moment W_n^2 of the fraction of the resolved interval, that is, we deal now with recurrence (2.3) for $m=2$. It is easy to see that the modified generating function $w_2(x)$ satisfies

$$w_2(x)=2^{-1}a(x)w_2(x/2)+b(x), \tag{4.1}$$

where $a(x)$ is defined as before (see (3.1)), and

$$b(x)=w_1(x)-\tfrac{1}{2}w_1(x/2) \tag{4.2}$$

with $w_1(x)$ defined in (3.1). The functional equation (4.1) falls into (2.4) with $s=-1$ and $a(0)=\frac{1}{2}$. Since $b(x)=O(1)$ for $x\to 0$ the recurrence (4.1) has the solution of form (2.5). Define

$$Q_2(x)=\frac{xw_2(x)}{w_1(x)}, \qquad q_2(x)=\frac{xb(x)}{w_1(x)}; \tag{4.3}$$

then the Mellin transform of (4.3) is

$$Q_2^*(s)=\frac{q_2^*(s)}{1-2^s}$$

for $-1<\operatorname{Re} s<0$, and after simple algebra we show that

$$Q_2^*(s)=-\frac{2\log 2+\beta}{\log 2}\cdot\frac{1}{s}+\sum_{\substack{k=-\infty\\ k\neq 0}}^{\infty}\frac{q_2^*(s)}{s-\chi_k}+O(1), \tag{4.4}$$

where β is defined in (2.12). Alternatively, β can be computed as

$$\beta=2\sum_{n=0}^{\infty}(-1)^n\sum_{k=0}^{n}\binom{n}{k}\frac{k!}{(n+1)^{k+1}}.$$

Formula (4.4) can be translated into

$$w_2(x)\sim a(2+b)\frac{1}{x^2}+\frac{a}{x^2}P(\log x), \tag{4.5a}$$

where $b=\beta/\log 2$, and

$$P(u)=\frac{1}{\log 2}\sum_{\substack{k=-\infty\\ k\neq 0}}^{\infty}q_2^*\left(\frac{2\pi ik}{\log 2}\right)\exp\left[-\frac{2\pi iku}{\log 2}\right]. \tag{4.5b}$$

This proves (2.10) in Proposition (i), and by Lemma 2 it also establishes (2.26) of Theorem 2.

To obtain ultimate asymptotic analysis for $\mathbf{W}_n$, we need only to prove (2.27) for the mth moment of the fraction of the resolved interval. Using (2.2) for general m, we show that the modified generating function $w_m(x)$ satisfies

$$w_m(x)=2^{-m+1}a(x)w_m(x/2)+b(x),$$

where $b(x)=O(e^{-x})$ for $x\to\infty$. Then, generalizing (4.3), we define

$$Q(x)=\frac{x^{m-1}w_m(x)}{w_1(x)}, \qquad q(x)=\frac{x^{m-1}b(x)}{w_1(x)};$$

then $Q^*(s)=q^*(s)/(1-2^s)$. But $q^*(s)=O(1)$ for $s\to 0$, hence $Q^*(s)=O(s^{-1})$, and finally by (4.12), $W_m(x)=O(x^{-m})$. Using Lemma 2, we prove (2.27).

4.2. The number of resolved packets. The corresponding functional equation for the modified generating function $c_2(x)$ satisfies

$$c_2(x) = 2a(x)c_2\frac{x}{2} + b(x), \tag{4.6}$$

where

$$b(x) = x\,e^{-x/2}c_1\frac{x}{2} = \frac{x^2}{2}\,e^{-x/2}w_1\frac{x}{2}$$

and the last equality follows from $c_1(x) = xw_1(x)$ proved in § 3.2. Then by Lemma 1, the functional equation (4.6) possesses the solution $c_2(x) = c_2'(x) + c_2''(x)$, where by (2.7) of Lemma 1, $c_2'(x) = xw_1(x)$. The second term $c_2''(x)$ has the infinite series solution of form (2.5). Therefore, the same derivation as in § 4.1 leads finally to (2.14) of Proposition (ii). The proof of (2.30) of Theorem 2(ii) follows the same arguments as in the end of § 4.1 and is left to the reader.

4.3. The conflict resolution interval. The analysis of T_n^2 is much more intricate. Define $t_2(x) = T_2(x)\,e^{-x} - 1$; then the modified generating function $t_2(x)$ satisfies

$$t_2(x) = 2a(x)t_2\frac{x}{2} + b(x) \tag{4.7}$$

where

$$b(x) = 2t_1(x) + x\,e^{-x/2}t_1\frac{x}{2} + \frac{x}{2}\,e^{-x/2} - \frac{5}{2}\,x\,e^{-x/2} - 1.$$

We split $b(x)$ into two functions, $b(x) = b_1(x) + b_2(x)$, where $b_1(x)$ is given by (3.19) and

$$b_2(x) = 2t_1(x) - 2 + x\,e^{-x/2}t_1\frac{x}{2} - x\,e^{-x}. \tag{4.8}$$

Then, $t_2(x) = t_1(x) + t_2''(x)$, where $t_1(x)$ is evaluated in § 3.3, and $t_2''(x)$ satisfies our general functional equation (2.4) with the solution given by (2.5). Therefore, our general approach from the previous sections can be used. Define $Q_4(x) = t_2''(x)/xw_1(x)$ and $q_4(x) = b_2(x)/xw_1(x)$; then the Mellin transform is $Q_4^*(s) = q_4^*(s)/(1-2^s)$, and it exists only for $-1 < \operatorname{Re} s < -\varepsilon$. As in the case of $t_1(x)$, the transform $Q_4^*(s)$ needs a special treatment to find an asymptotic expansion. We follow the approach from § 3.3 and define a new function

$$f_2(x) = q_4(x) - q_4\frac{x}{2} = \frac{b_2(x) - 2b_2(x/2)a(x)}{xw_1(x)} \tag{4.9}$$

for which the Mellin transform $f_2^*(s)$ exists for $-1 < \operatorname{Re} s < 0$. The function $f_2^*(s)$ has pole at $s = 0$; hence $f_2^*(s) = -2/(s\alpha) + f_2^* + s\nu + O(s^2)$, where f_2^* and ν can be computed in the same manner as before and are given by

$$f_2^* = -\int_0^\infty f_2'(x)\log x\,dx, \qquad \nu = -\frac{1}{2}\int_0^\infty f_2'(x)\log^2 x\,dx. \tag{4.10}$$

Using the same arguments as in the derivation of (3.29), we can prove that

$$f_2^* = \frac{2}{\alpha}(c-1)\log 2 - \frac{\log 2}{\alpha}. \tag{4.11}$$

Finally, the Mellin transform of $Q_4(x)$ can be expressed as

$$Q_4^*(s) = \frac{-2}{a \log^2 2} \cdot \frac{1}{s^3} + \frac{1}{s^2}\left[\frac{f_2^*}{\log^2 2} + \frac{2}{a \log 2}\right] - \frac{1}{s}\left[\frac{5}{6a} + \frac{f_2^*}{\log 2} - \frac{\nu}{\log^2 2}\right] + O(1) \tag{4.12}$$

so (2.16) of Proposition (iii) is proved. By Lemma 2, the following also holds:

$$T_n^2 = \log_2^2 n + 2c \log_2 n + \frac{5}{6} + 3(c-1) - \frac{a\nu}{\log^2 2} + P(\log n) + O(n^{-1/2}) \tag{4.13}$$

and (2.32) of Theorem 2(iii) follows from $\operatorname{var} \mathbf{T}_n = T_n^2 - (T_n^1)^2$. The rest is simple and is left to the reader (see also [10]).

Acknowledgment. We thank an anonymous referee for very careful reading of our paper and several suggestions for improvement of the presentation.

REFERENCES

[1] T. BERGER, *Poisson multiple-access contention with binary feedback*, IEEE Trans. Inform. Theory, 30 (1984), pp. 745-751.

[2] J. CAPENTANAKIS, *Tree algorithms for packet broadcast channels*, IEEE Trans. Inform. Theory, 25 (1979), pp. 505-515.

[3] B. DAVIES, *Integral Transforms and Their Applications*, Springer-Verlag, Berlin, New York, 1985.

[4] P. FLAJOLET, M. REGNIER, AND R. SEDGEWICK, *Some uses of the Mellin integral transform in the analysis of algorithms*, NATO ASI Series, Vol. F12, Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, Berlin, New York, 1985.

[5] R. GALLAGER, *Conflict resolution in a random access broadcast network*, in Proc. AFOSR Workshop in Communication Theory and Applications, 1978, pp. 74-76.

[6] A. GREENBERG AND S. WINOGRAD, *A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels*, J. Assoc. Comput. Mach., 32 (1985), pp. 589-596.

[7] A. GREENBERG, P. FLAJOLET, AND R. LADNER, *Estimating the multiplicities of conflicts to speed their resolution in multiple access channels*, J. Assoc. Comput. Mach., 34 (1987), pp. 289-325.

[8] P. HENRICI, *Applied and Computational Complex Analysis*, Vol. 2, John Wiley, New York, 1977.

[9] J.-C. HUANG AND T. BERGER, *Delay analysis of* 0.487 *contention resolution algorithms*, IEEE Trans. Comm., 34 (1986), pp. 916-926.

[10] P. JACQUET AND W. SZPANKOWSKI, *Ultimate characterizations of the burst response of interval searching algorithm*, CSD TR-711, Purdue University, West Lafayette, IN, 1987.

[11] P. JACQUET AND M. REGNIER, *Limiting distributions for trie parameters*, Lecture Notes in Computer Science 214, Springer-Verlag, Berlin, New York, 1986, pp. 196-210.

[12] D. KNUTH, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, MA, 1973.

[13] W. SZPANKOWSKI, *Solution of a linear recurrence equation arising in the analysis of some algorithms*, SIAM J. Algebraic Discrete Methods, 8 (1987), pp. 233-250.

[14] ———, *An analysis of a contention resolution algorithm—Another approach*, Acta Inform. 24 (1987), pp. 173-190.

[15] B. TSYBAKOV AND V. MIKHAILOV, *Free synchronous packet access in a broadcast channel with feedback*, Problems Inform. Transmission, 14 (1978), pp. 259-280.

[16] ———, *Random multiple packet access: Part-and-try algorithm*, Problems Inform. Transmission, 16 (1980), pp. 305-317.

[17] D. WILLARD, *Sampling algorithms for differentiable batch retrieval problems*, in Proc. 11th International Conference on Automata, Languages and Programming (ICALP) 1984, pp. 514-526.

[18] ———, *Log-logarithmic selection resolution protocols in a multiple access channel*, SIAM J. Comput., 15 (1986), pp. 468-477.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 792-810, August 1989

012

AN OPTIMAL-TIME ALGORITHM FOR SLOPE SELECTION*

RICHARD COLE†, JEFFREY S. SALOWE‡, W. L. STEIGER§, AND ENDRE SZEMERÉDI¶§

Abstract. Given n points in the plane and an integer k, the problem of selecting that pair of points that determines the line with the kth smallest or largest slope is considered. In the restricted case, where k is $O(n)$, line sweeping gives an optimal, $O(n \log n)$-time algorithm. For general k the parametric search technique of Megiddo is used to describe an $O(n(\log n)^2)$-time algorithm. This is modified to produce a new, optimal $O(n \log n)$-time selection algorithm by incorporating an approximation idea.

Key words. computational geometry, selection, slopes

AMS(MOS) subject classifications. 68Q25, 68H05

1. Introduction. Given n distinct points in the plane, $(x_1, y_1), \cdots, (x_n, y_n)$, write $N = \binom{n}{2}$ and consider the N (not necessarily distinct) lines they determine, $y = a_{ij}x + b_{ij}$, $1 \leqq i < j \leqq n$, one for each pair of points. In [4], Chazelle mentions the problem of selecting one of these lines according to the rank of its slope: given $1 \leqq k \leqq N$ we seek the kth smallest element of $S = \{a_{ij}, 1 \leqq i < j \leqq n\}$.

When $k = N/2$, the median slope is the Theil estimator [13] of the slope of the regression line through the data. In [12], Shamos enquires about the complexity of computing the Theil estimator and wonders whether it might be $o(N)$. Chazelle [4] studies several interesting geometric selection problems but does not give any results for the selection of slopes. In this paper, we describe some facts relating to the problem that may not be widely known and give a new, optimal-time algorithm for the slope selection task. The algorithm is applicable to other geometric selection problems [10] and may also have wider interest.

If we were to just present S, the fast selection algorithm of Blum et al. [3] would use $O(N)$ comparison steps for any rank, k. However, if we exploit the geometric information embodied by the n points, we may be able to (i) avoid computing all N slopes and consequently (ii) find the rank k line in $o(N)$ steps. From now on we take a *step* to mean any arithmetic operation, comparison, pointer manipulation, or read to/write from memory.

The paper is organized as follows. In § 2, the shallow selection problem is studied. In shallow selection, k is nearly 1 or N. In § 3, an $O(n(\log n)^2)$-time algorithm is presented for the general selection problem. This algorithm is improved in § 4 to yield an optimal $O(n \log n)$-time algorithm. Section 5 contains general remarks and a summary of the results.

2. Shallow selection. In the slope selection problem, we are given n points, $(x_1, y_1), \cdots, (x_n, y_n)$. Throughout the paper, assume that the points are in general position. In this context, general position means that no three points are collinear and

* Received by the editors July 29, 1987; accepted for publication (in revised form) November 23, 1988. A preliminary version of this paper appeared as *Optimal slope selection*, in Proceedings of the 15th International Colloquium on Automata, Languages and Programming, pp. 133-146, 1988.

† Courant Institute of Mathematical Sciences, New York University, New York, New York 10002. The work of this author was supported by National Science Foundation grants DCR-84-01633 and CCR-8702271, and Office of Naval Research contract N00014-85-K-0046.

‡ Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903.

§ Department of Computer Science, Rutgers University, Busch Campus, New Brunswick, New Jersey 08903.

¶ Mathematical Institute, Hungarian Academy of Sciences, Budapest, Hungary.

that no two of the induced lines have the same slope; this assumption simplifies the arguments without affecting the asymptotic complexity of the algorithms.

Consider the selection problem when $k = 1$. It is easy to show that $\Omega(n \log n)$ time is a lower bound for this problem in the algebraic decision tree model of computation. It is known that $\Omega(n \log n)$ time is needed to determine if a set of n numbers are distinct in the algebraic decision tree model [8]. This problem is known as ELEMENT UNIQUENESS. In linear time, we can transform an instance of the ELEMENT UNIQUENESS problem to the problem of selecting the smallest slope. Given as input a set of n numbers H, map each number h_i in H to point (h_i, i) in the plane. If the smallest slope is finite, then input points are distinct; otherwise, some pair h_i and h_j have the same value.

In fact, $O(n \log n)$ steps are sufficient to find either the smallest or the largest slope (the extreme slopes). Sort the input points $(x_1, y_1), \cdots, (x_n, y_n)$ by x-coordinate. It is easy to see that the extreme slopes are determined by a line incident to points having adjacent x-coordinates. By computing the relevant $n-1$ slopes and finding the minimum, we see that extremal selection can be done in $O(n \log n)$ steps. The idea is easily adapted to cover optimal shallow selection.

LEMMA 1. *If $k = O(\log n)$, the rank k slope may be found in $O(n \log n)$ steps.*

Proof. Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, $x_1 \leqq x_2$, be the pair of points determining line l, whose slope has rank k. We claim that there are at most $k-1$ other points whose x-coordinates lie between x_1 and x_2. Each intervening point bears witness to the presence of an additional line with slope smaller than l. For any point p between x_1 and x_2 lying below l, the line incident to p_1 and p has slope less than that of l. For any point p between x_1 and x_2 lying above l, the line incident to p_2 and p has slope smaller than that of l. Since no three points are collinear, these are the only possibilities.

The algorithm to select slope k would sort the input points by x-coordinate, consider pairs of points that are separated by at most $k-1$ other points, and use the linear time selection algorithm to find the kth smallest slope. Since $k = O(\log n)$, this algorithm uses $O(n \log n)$ steps. □

It is easy to see that Lemma 1 can also be used to select slopes with rank $N - k + 1$, where $k = O(\log n)$.

To make more progress in designing selection algorithms, we transform the slope selection problem into a more convenient form using a point-line duality. This is defined by a mapping T that takes the point $p = (a, b)$ to the line Tp given by $y = ax + b$ and the line l given by $y = cx + d$ to the point $Tl = (-c, d)$. It is well known that T preserves incidence. That is, if points p_1 and p_2 are incident with line l, Tl is the point of intersection of the lines Tp_1 and Tp_2; similarly, if l_1 and l_2 intersect at point p, Tp is the line incident with the points Tl_1 and Tl_2.

Under T, the n given points map to lines $l_1, \cdots, l_n$ whose points of intersection, $l_i \cap l_j \equiv (u_{ij}, v_{ij})$, correspond to the lines incident with the ith and jth original points. Furthermore, $-u_{ij}$ is the slope of this line. Thus, the dual of the problem of selecting slope k is the following, equivalent selection problem: **Given distinct lines $l_1, \cdots, l_n$, find the intersection point whose x-coordinate is the $N - k + 1$th smallest element of the set $TS = \{u_{ij}, 1 \leqq i < j \leqq n\}$.** We will consider the general problem of selecting elements in TS in the remainder of the paper.

For ease of exposition, we will abuse notation by blurring the distinction between intersection points and their x-coordinates. This is justified because the y-coordinates provide no information for the selection problem. We write $t_1 < \cdots < t_N$ for the $\binom{n}{2}$ elements of TS is sorted order. Note that the t_i's are distinct because the input points are in general position. Given k, we seek t_k.

In this dual problem, optimal *shallow* selection ($k = O(n)$) can be effected by the line sweep technique of Bentley and Ottmann [2]. The sweep starts at $x = a$, $a < t_1$. Let π be the permutation that sorts the slopes in ascending order. Thus $m_{\pi_1} < \cdots < m_{\pi_n}$, where we write $y = m_i x + b_i$ as the equation of l_i. If $a < t_1$, the vertical line $x = a$ meets $l_1, \cdots, l_n$ at $y_1(a), \cdots, y_n(a)$ and $y_{\pi_1}(a) > \cdots > y_{\pi_n}(a)$. For each adjacent pair of lines, find the x-coordinate $z_i = (b_{\pi_{i+1}} - b_{\pi_i})/(m_{\pi_i} - m_{\pi_{i+1}})$ of their intersection point and place these $n-1$ numbers in a (min) heap. Clearly $t_1 = \min(z_i)$. It has been obtained in time $O(n \log n)$.

Suppose z_p is the smallest z_i. It is currently at the top of the heap. We sweep just beyond $t_1 = z_p$. When $a > t_1$, $y_{\pi_p}(a) < y_{\pi_{p+1}}(a)$ because lines l_{π_p} and $l_{\pi_{p+1}}$ have crossed at t_1. We compute the two new intersection points $z' = (b_{\pi_{p+1}} - b_{\pi_{p-1}})/(m_{\pi_{p-1}} - m_{\pi_{p+1}})$ and $z'' = (b_{\pi_{p+2}} - b_{\pi_p})/(m_{\pi_p} - m_{\pi_{p+2}})$ (if $p = n-1$, only compute z'; if $p = 1$, only compute z''). Delete z_p from the heap, insert z' and z'', and exchange π_p with π_{p+1}. We may now obtain $t_2 = \min(z_i)$ having expended $O(\log n)$ steps. The two new z's were computed in constant time, and the deletion and two insertions took $O(\log n)$ time. Continuing in the same fashion by sweeping through to t_k, we have Lemma 2.

LEMMA 2. *Given n lines $l_1, \cdots, l_n$ and an integer k, the cost of sweeping to the intersection point $l_i \cap l_j$ with the kth smallest x-coordinate is $O(n \log n + k \log n)$.*

When $k = O(n)$, the total effort would be $O(n \log n)$, and this is optimal. In the primal problem we achieve optimal selection for slopes of rank $k = N - O(n) + 1$. By symmetry, we can select slopes with $k = O(n)$ in optimal time as well.

When $k \neq O(n)$, a more powerful technique is required for efficient slope selection. In the next section we give an $O(n(\log n)^2)$ time algorithm for selecting elements in *TS*.

3. The general selection algorithm. Write $\pi(a)$ for the permutation that orders the intercepts of $l_1, \cdots, l_n$ in descending order at $x = a$. Thus, under obvious notation, $y_{\pi_1(a)}(a) > \cdots > y_{\pi_n(a)}(a)$. Renumber $l_1, \cdots, l_n$ so that for $a < t_1$, $\pi(a)$ is the identity. At the intersection points t_i, we disambiguate π by defining $\pi(t_i) = \pi(t_i + \varepsilon)$, where $0 < \varepsilon < \min(t_{j+1} - t_j)$. The permutation $\pi(t_1)$ has one inversion (exactly one pair of lines crossed at t_1), $\pi(t_2)$ has two, etc. In fact the function

$$I(\pi(x)) = \#\text{ inversions in } \pi(x)$$

is a monotone step function in x with unit jumps at the t_i's. $I(\pi(x)) = j$ if and only if $t_j = \max(t_i : t_i \leqq x)$. For a given k, the problem of finding t_k may be viewed as an unusual sorting problem to which we apply Megiddo's ingenious technique [7], improved by Cole [5], of building sequential algorithms from parallel ones. An implicit binary search over the t_i's is performed, each step taking $O(n \log n)$ time. This will give an $O(n(\log n)^2)$-time algorithm.

In seeking t_k, we will attempt to sort $y_1(a^*), \cdots, y_n(a^*)$ at $a^* = t_k + \varepsilon$. We know that this sort may be achieved in $O(n \log n)$ "comparisons," each answering a question Q_{ij} of the form "$y_i(a^*) \leqq y_j(a^*)$?". The $O(n \log n)$ answers yield the permutation π^* that sorts these intercepts: $y_{\pi_1^*}(a^*) > \cdots > y_{\pi_n^*}(a^*)$. Once π^* has been found,

$$t_k = \max[u_{\pi_i^*, \pi_{i+1}^*} : \pi_i^* > \pi_{i+1}^*];$$

the kth inversion must have just reversed a pair of adjacent intercepts in the permutation $\pi(t_{k-1})$.

The control structure for the sort will come from the $O(\log n)$-depth sorting network of Ajtai, Komlós, and Szemerédi [1]. At each level, $n/2$ of the questions

Q_{ij}: "$y_i(a^*) \leqq y_j(a^*)$?" are answered. The network is just a guide for blocking these comparisons into groups of size $n/2$; the comparisons are actually performed in series. The sort is complete once the $O(n \log n)$ answers are obtained and we have determined π^*.

Even though we do not know a^*, we can answer the question "$y_i(a^*) \leqq y_j(a^*)$?" in time $O(n \log n)$ by ranking. We find u_{ij}, the *x-coordinate of* $l_i \cap l_j$, $i<j$, in constant time and then obtain its rank among the t_i's as follows. Sort the n intercepts at u_{ij} in decreasing order to get $\pi(u_{ij})$. The rank of u_{ij} is the number of inversions in $\pi(u_{ij})$, $I(\pi(u_{ij}))$. If $I(\pi(u_{ij}))>k$ we know that $u_{ij}>t_k$ so the answer is "no;" lines i and j have not yet crossed at t_k. Similarly if $I(\pi(u_{ij}))<k$, $u_{ij}<t_k$ and the answer is "yes." If $I(\pi(u_{ij}))=k$, $t_k=u_{ij}$. I may be computed in time $O(n \log n)$ as explained in Knuth [6]: if $\pi(u_{ij})=(r_1, \cdots, r_n)$, we use merge-sort to sort the slopes $m_{r_1}, \cdots, m_{r_n}$ and count the inversions that were performed.

If we actually answered all $n/2$ questions $Q_{i_1 j_1}, \cdots, Q_{i_{n/2} j_{n/2}}$ on a level of the network by counting inversions, the complexity would be $O(n^2 \log n)$ for that level and $O((n \log n)^2)$ overall. The trick is to resolve the $n/2$ questions on a level by actually counting inversions only $O(\log n)$ times.

As mentioned, each question Q_{ij} determines an intersection point of a distinct pair of lines, and the answer to Q_{ij} is obtained by comparing the rank of that intersection point with k. On a given level of the sequentialized version of the parallel sorting network, denote the x-coordinates of these points by $z_{i_1}, \cdots, z_{i_{n/2}}$. We can compute the median of these intersection points, z_{med}, in time $O(n)$. It costs time $O(n \log n)$ to rank z_{med}, and this answer resolves half the questions. For example, if $z_{\text{med}}<t_k$, then $z_{i_j}<t_k$ for all the $z_{i_j} \leqq z_{\text{med}}$. Continuing with the $n/4$ unresolved questions on this level, we again find the median z and rank it in time $O(n \log n)$, etc. After $O(\log n)$ inversions counts, all $n/2$ questions on this level are resolved. Since each inversion count takes $O(n \log n)$ steps and there are $O(\log n)$ levels, the algorithm has time complexity $O(n(\log n)^3)$.

In [5], Cole shows how this result may be improved by a factor of $\log n$. Each question in the network has two inputs. On level 1, all inputs are prescribed. The inputs to each question on level $j>1$ depend on the answers to some pair of questions from level $j-1$. A question is "active" if both its inputs have been determined but it is not yet answered. The idea is to resolve questions as they become active. A weight of $1/4^{j-1}$ is assigned to active questions on level j. At each "stage" the weighted median, z_{wmed}, of intersection points of active questions is determined in linear time [9] and resolved in time $O(n \log n)$ by ranking z_{wmed}. As before, this resolves either all questions for which $z \leqq z_{\text{wmed}}$ or all questions for which $z \geqq z_{\text{wmed}}$. Once resolved, questions are no longer active. Cole shows that the weight of the active questions is reduced by a factor of at least $\frac{1}{4}$ after ranking z_{wmed}. In total, $O(\log n)$ inversion counts are sufficient to resolve all the questions asked by the network. This argument proves the following assertion:

THEOREM 3. *Given n lines* $l_1, \cdots, l_n$ *and an integer k, the intersection point* $l_i \cap l_j$ *with the kth smallest x-coordinate may be found in time* $O(n(\log n)^2)$.

4. Improvement with approximate ranking. The previous sections showed that for any given k, the worst-case time complexity to select the intersection point with the kth smallest x-coordinate is in the interval $(cn \log n, dn(\log n)^2)$. In this section we show that the lower bound is sharp or equivalently:

THEOREM 4. *Given n lines* $l_1, \cdots, l_n$ *and an integer k, the complexity of finding the intersection point* $l_i \cap l_j$ *with the kth smallest x-coordinate is* $O(n \log n)$.

The proof will describe an explicit algorithm, argue its correctness, and demonstrate the time complexity assertion. The algorithm is superimposed on the one used in Theorem 3, and it is presented in five sections. Section 4.1 presents the general strategy, §§ 4.2–4.4 describe the three essential procedures in the algorithm, and § 4.5 completes the analysis and proves Theorem 4.

4.1. The approximation algorithm. The new idea is to use an approximate rank for each point chosen by the sorting network. Let sign (x) be $+1$ if x is positive, 0 if x is zero, and -1 if x is negative. The algorithm discussed in § 3 expended time $O(n \log n)$ to find rank (z_{wmed}) for each of the $O(\log n)$ weighted median points given by the network. This rank determined sign $(z_{\text{wmed}} - t_k)$, and we could resolve all the z_i for which sign $(z_i - z_{\text{wmed}}) = \text{sign}\,(z_{\text{wmed}} - t_k)$. In the improved algorithm, we will use $O(n)$ (amortized) time to develop an approximation ρ_z to the rank of z. Let e_z denote the error of the approximation:

$$e_z = |\text{rank}\,(z) - \rho_z|.$$

If the error is small enough, ρ_z can tell us the relative ordering of z and t_k. By definition, rank (z) must lie inside the closed interval $[\rho_z - e_z, \rho_z + e_z]$. If k lies outside the interval $[\rho_z - e_z, \rho_z + e_z]$, then sign $(z - t_k) = \text{sign}\,(\text{rank}\,(z) - k) = \text{sign}\,(\rho_z - k)$; in this case, the relative ordering of z and t_k can be deduced from ρ_z and k, and we may resolve the relevant z_i's. On the other hand, if k lies inside the interval $[\rho_z - e_z, \rho_z + e_z]$, the approximation will not be accurate enough to distinguish the relative positions of z and t_k, so we cannot use z resolve any other questions. To account for this, we will do some extra comparisons to refine ρ_z and reduce e_z until k lies outside the interval $[\rho_z - e_z, \rho_z + e_z]$. It turns out that at most $O(n \log n)$ extra work will be done to refine rank approximations throughout the entire course of the algorithm.

The key structure used in the approximation is the following. Suppose we have a "reference" intersection point at $x = r$ and we know sign $(r - t_k)$. Also suppose that we have partitioned the n lines into $\tau = \lceil n/T \rceil$ groups $G_1, \cdots, G_\tau$ of size T (maybe G_τ is smaller) with the property that if $i_1 \in G_i$, $i_2 \in G_j$, and $i < j$, then $y_{i_1}(r) \geqq y_{i_2}(r)$, where T is an integer to be specified later. Thus, G_1 refers to the lines with the T largest intercepts at $x = r$, G_2 the lines with the next T largest intercepts, etc. The groups are therefore sorted, but within any particular group, we have no ordering information. Such a structure is called a *partition of size T at r*. We maintain partitions of size T at the current "reference" points as an invariant. The occasional refinements will divide T by 2 and restore the invariant in time $O(n)$ per refinement, $O(n \log n)$ overall.

A partition of size T at r is represented by a permutation $p(r, T) = (p_1(r, T), \cdots, p_n(r, T))$, where $p_1(r, T), \cdots, p_T(r, T)$ are the lines in G_1, $p_{T+1}(r, T), \cdots, p_{2T}(r, T)$ the lines in G_2, etc. Partition $p(r, T)$ is thus an approximation to $\pi(r)$, the permutation that correctly orders the intercepts $y_i(r)$. We define the approximate rank $\rho_{(r,T)}$ to be the number of inversions in $p(r, T)$:

$$\rho_{(r,T)} = I(p(r, T)). \tag{1}$$

Because $p(r, T)$ represents the partition, it can differ from $\pi(r)$ only with respect to inversions between pairs of lines within the same group. Since there are at most $\binom{T}{2}$ such inversions in a group, $\rho_{(r,T)}$ differs from rank (r) by less than $nT/2$. Therefore, for any given value of T,

$$e_r < nT/2. \tag{2}$$

If the value of T is clear from the context, we will remove the parameter T from $p(r, T)$ and $\rho_{(r,T)}$.

To prove the correctness and derive the time complexity of the approximation algorithm, we will choose T to be a power of 2 for which

$$|\rho_{(r,T)} - k| \leqq nT, \tag{3}$$

$$|\rho_{(r,T/2)} - k| > nT/2. \tag{4}$$

Because (2) implies that $\rho_{(r,T/2)} - nT/4 < \text{rank}\,(r) < \rho_{(r,T/2)} + nT/4$ and (4) implies that k lies outside the open interval $(\rho_{(r,T/2)} - nT/2,\ \rho_{(r,T/2)} + nT/2)$, $\text{sign}\,(\rho_{(r,T/2)} - k) = \text{sign}\,(\text{rank}\,(r) - k) = \text{sign}\,(r - t_k)$. From condition (4), we are able to determine the relative position of r and t_k, so our approximation correctly reveals whether the current point r lies on the left or right of t_k. If (3) and (4) should fail for the current value of T, we would halve T and create the partition for the new smaller groups. The impact of (3) and (4) is indicated in the following lemma.

LEMMA 5. *Let $p(r, T)$ be a partition of size T at r, and assume that* (3) *and* (4) *hold for $p(r, T)$. Then $nT/4 < |\text{rank}\,(r) - k| < 3nT/2$.*

Proof. By (2), $|\text{rank}\,(r) - \rho_{(r,T)}| < nT/2$, and by (3), $|\rho_{(r,T)} - k| \leqq nT$. Therefore, $|\text{rank}\,(r) - k| < 3nT/2$. By (4), $|\rho_{(r,T/2)} - k| > nT/2$, and by (2), $|\text{rank}\,(r) - \rho_{(r,T/2)}| < nT/4$. Therefore, $|\text{rank}\,(r) - k| > nT/4$. □

We now describe the flow of control in the approximation algorithm. The algorithm maintains *two reference points*: $r_L \leqq t_k$ and $r_R \geqq t_k$. We maintain orderings $p(r_L)$ and $p(r_R)$ and calibrated partition sizes T_L and T_R for both reference points. The reference point with the smaller value of T is "active." (From this point in the paper until § 4.5, we will abbreviate $p(r, T)$ by $p(r)$ because the value of T will be clear from the context.) When the network gives a new query point $x = q$, if $q \notin (r_L, r_R)$, the relative position of q with respect to t_k may be deduced by transitivity. We can immediately resolve this point and the relevant z_i's (of which q was the weighted median). Otherwise, we construct the partition at $x = q$. If $q < t_k$, the partition of size T at q is a modification of the partition of size T at r_L. If $q > t_k$, the partition of size T at q is a modification of the partition of size T at r_R. Since we do not know the relative ordering of q and t_k, an attempt is made to modify the partition at r, the active reference point. The attempt will be aborted if $|\text{rank}\,(r) - \text{rank}\,(q)|$ is "large;" in this case, the other reference point r' then becomes active and the partition at r' is modified to give the partition at q. Position q then replaces the appropriate reference point (on the same side of t_k as q) and the reference point with the smallest value of T becomes the new active reference point.

Figure 1 contains a description of the selection algorithm. To initialize the algorithm, we find any $r_L < t_1$ and $r_R > t_N$ in $O(n \log n)$ steps by the shallow selection technique and use $p(r_L) = (1, \cdots, n)$ and $p(r_R) = (n, \cdots, 1)$ regardless of the initial values of T_L and T_R. To calibrate T_L, we know $k - \rho_{(r_L,T)} = k - \rho_{(r_L,T/2)} = k$, and from (3) and (4), $k/n \leqq T_L < 2k/n$. Therefore, $T_L = 2^{\lceil \log(k/n) \rceil}$; similarly, $T_R = 2^{\lceil \log((N-k)/n) \rceil}$. In the algorithm, the variable SIDE indicates which reference point is active: SIDE $= -1$ means r_L is active and SIDE $= 1$, that r_R is.

We shall briefly explain the above steps and analyze them in detail in subsequent sections. The lines ⟨0⟩-⟨3⟩ initialize the active reference point. The algorithm will continue to make approximations until T is within a constant. When T is less than or equal to 10, the error bound and the properties of the partition imply that the maximum number of intervening vertices between q and t_k is less than $5n$, so the sweepline algorithm used in Lemma 2 can be slightly modified to find t_k in $O(n \log n)$ additional steps. At line ⟨5⟩, the next vertex to be compared with t_k is delivered; this vertex has x-coordinate q. If q is outside the interval from r_L to r_R, it may be resolved immediately

```
⟨0⟩  IF T_L ≦ T_R THEN
⟨1⟩    T ← T_L; SIDE ← −1; r ← r_L; ρ_r ← 0; p(r) = (1, 2, ···, n)
⟨2⟩  ELSE
⟨3⟩    T ← T_R; SIDE ← 1; r ← r_R; ρ_r ← n(n−1)/2; p(r) = (n, n−1, ···, 1)
⟨4⟩  WHILE T > 10
⟨5⟩    get next q from network
⟨6⟩    IF q ∉ (r_L, r_R) THEN
⟨7⟩      RESOLVE q
⟨8⟩    ELSE
⟨9⟩      PARTITION (0, T, p(r), p'(q), TOOFAR)
⟨10⟩     IF TOOFAR = 0 THEN
⟨11⟩       COUNT (ρ_q, ρ_r, p'(q), p(r), SIDE, TOOFAR)
⟨12⟩       WHILE (3) and (4) do not hold AND TOOFAR = 0 AND sign (ρ_r − k) = sign (ρ_q − k) DO
⟨13⟩         T ← T/2; HALVE (p'(q), ρ_q)
⟨14⟩       END WHILE
⟨15⟩       p(q) ← p'(q)
⟨16⟩     END IF
⟨17⟩     IF TOOFAR = 1 OR Sign (ρ_r − k) ≠ Sign (ρ_q − k) THEN
⟨18⟩       CHANGESIDES (SIDE, r, T, p(r)); PARTITION (0, T, p(r), p'(q), TOOFAR)
⟨19⟩       COUNT (ρ_q, ρ_r, p'(q), p(r), SIDE, TOOFAR)
⟨20⟩       WHILE (3) and (4) do not hold DO
⟨21⟩         T ← T/2; HALVE (p'(q), ρ_q)
⟨22⟩       END WHILE
⟨23⟩       p(q) ← p'(q)
⟨24⟩     END IF
⟨25⟩     RESOLVE q; FIXREFS
⟨26⟩   END IF
⟨27⟩ END WHILE
⟨28⟩ SWEEP TO t_k
```

FIG. 1. *Algorithm description.*

by transitivity. Therefore, RESOLVE q in line ⟨7⟩ should be interpreted to mean "answer q and all z_i (of which q is the weighted median) for which sign $(z_i - q) =$ sign $(q - t_k)$."

If q is inside the interval from r_L to r_R, more work must be done. As stated earlier, if $q < t_k$, the partition of size T at q must be obtained by modifying the partition of size T_L at r_L. If $q > t_k$, the partition of size T at q must be obtained by modifying the partition of size T_R at r_R. (The rationale for this restriction will be explained in § 4.2.) The important procedures are PARTITION, COUNT, and HALVE.

The procedure PARTITION is the heart of the algorithm and will be described in detail in § 4.2. For now, it is enough to take the following statements on faith: (i) whenever $|\text{rank}\,(r) - \text{rank}\,(q)| < 3nT/2$, it will use $p(r)$ to construct a partition of size T at q, $p'(q)$, in linear time. (ii) If $3nT/2 \leqq |\text{rank}\,(r) - \text{rank}\,(q)|$, PARTITION will halt in linear time either with TOOFAR = 1 or with a partition of size T at q, $p'(q)$.

If TOOFAR = 1 in line ⟨10⟩, then $|\text{rank}\,(q) - \text{rank}\,(r)| \geqq 3nT/2$. By Lemma 5, this implies that q and r are on opposite sides of t_k, so control proceeds to line ⟨17⟩.

The procedure COUNT computes ρ'_q, the number of inversions in $p'(q)$, given ρ_r, $p'(q)$, $p(r)$, and SIDE. (The reader should observe that ρ'_q never appears in Fig. 1 since ρ_q is always updated on the fly. We introduce ρ'_q so that the action of procedure COUNT is clear.) In linear time COUNT either returns the correct value of ρ'_q or the message that $|\rho_r - \rho'_q| \geqq 3nT/2$. In the latter case, it will set TOOFAR to 1 since it is known that q and r cannot be on the same side of t_k. (This claim will be justified in § 4.4.) If TOOFAR is set to 1 in line ⟨11⟩, then control proceeds to line ⟨17⟩, as above.

If TOOFAR = 0 at the time line ⟨12⟩ is executed, $p'(q)$ defines a partition of size T at q. If the condition in line ⟨12⟩ is true, then (3) and (4) do not hold. The procedure HALVE converts the partition of size T into one of size $T/2$ and corrects ρ_q. The WHILE loop can end in one of two ways. Either T has been correctly calibrated and $p(q)$ has been formed, or sign $(\rho_q - k) \neq$ sign $(\rho_r - k)$. In the latter case, this will indicate that r and q are on opposite sides of t_k. (This claim will be justified in § 4.5.)

At line ⟨17⟩, the algorithm has either succeeded in computing a new partition at q, the flag TOOFAR has been set to 1, or sign $(\rho_q - k) \neq$ sign $(\rho_r - k)$. The test in line ⟨17⟩ decides whether q is on the same side of t_k as r. If the test is false, q is on the same side of t_k as r and the partition has already been computed. If the test in line ⟨17⟩ is true, q is on the opposite side of t_k as r, so q is on the same side of t_k as the other reference point r'. In this case, the partition of size T at q is computed with respect to r' in lines ⟨18⟩-⟨23⟩. Procedure CHANGESIDES in line ⟨18⟩ changes the value of variable SIGN, switches the reference point, and resets T to the value corresponding to the new reference point.

Since the relative position of q with respect to t_k is known when line ⟨25⟩ is executed, RESOLVE q can be performed. Actually, the partition of size T at q may not be accurate enough to determine the relative positions of q and t_k; as is stated in (4), it is the next smaller size of T that permits the resolution of q. To set up the variables for the next iteration, a procedure called FIXREFS sets $r_L \leftarrow q$, $p(r_L) \leftarrow p(q)$, and $T_L \leftarrow T$ if SIDE $= -1$ or $r_R \leftarrow q$, $p(r_R) \leftarrow p(q)$, and $T_R \leftarrow T$ otherwise. It then sets $T \leftarrow \min(T_L, T_R)$ and makes the reference point with the smaller value of T the active reference point.

4.2. The procedure PARTITION. At reference point $x = r$ (r is either r_L or r_R), the permutation $p(r)$ defines a partition of the n lines into $\tau = \lceil n/T \rceil$ groups $G_1 \geqq \cdots \geqq G_\tau$ of size T ($|G_\tau|$ may be $< T$). Recall that the inequalities mean that if $c \in G_i$, $d \in G_j$, and $i < j$, then $y_c(r) \geqq y_d(r)$. We wish to construct a new partition of size T' at q, $p(q)$, where $T' = T/2^s$ for some $s \geqq 0$. Procedure PARTITION performs the first step in this process. With respect to the active reference point, procedure PARTITION will, in linear time, either construct a partition of size T at q, $p'(q)$, or deduce that $|\text{rank}(r) - \text{rank}(q)| \geqq 3nT/2$. In the former case, procedure HALVE will, if necessary, form $p(q)$ by refining $p'(q)$ until (3) and (4) are reestablished. In the latter case, Lemma 5 implies that r and q are on opposite sides of t_k.

To make the selection algorithm correct, constraints must be placed on the partitions $p'(q)$ and $p(q)$. First, we require that the partition $p(q)$ is derived from the partition at the reference point r that satisfies sign $(q - t_k) =$ sign $(r - t_k)$. Second, we insist that when $r < q$:

(5) Lines appearing in any particular group in $p(q)$ or $p'(q)$ are in the same order as they were in $p(r)$.

There are two reasons to constrain the partition in the manner described above. First, the constraints simplify the maintenance of invariants (3) and (4). Specifically, a partition of size T at r may be in error by at most $nT/2$. At reference line r, $|\rho_r - k| \leqq nT$. To ensure that T cannot increase during the course of the algorithm, we insist that $|\rho'_q - k| \leqq |\rho_r - k| \leqq nT$ if q and r are on the same side of t_k. Here, ρ'_q is the number of inversions present in $p'(q)$. We will prove this result in Lemma 12.

The second motivation for (5) is to simplify the design of the counting procedures COUNT and HALVE. Suppose that a new partition $p'(q)$ is constructed by procedure PARTITION. Procedure COUNT will compute the number of inversions ρ'_q in $p'(q)$ by comparing the partition $p(r)$ with $p'(q)$. If the approximation ρ'_q is not sufficiently

accurate, procedure HALVE will be run. If no constraints are imposed within groups of the partition $p'(q)$, it is possible that PARTITION might invert a pair of lines between $x = r$ and $x = q$ and then HALVE reverses this inversion. The problem is that our COUNT and HALVE procedures will account twice for an inversion that does not appear in $p(q)$.

In the description of the algorithm, the groups in partition $p'(q)$ are denoted by $G'_1 \geqq \cdots \geqq G'_\tau$. When convenient, $p'(q)$ will be referred to as as $\{G'_i\}$. The groups in $p(r)$ are denoted by $G_1 \geqq \cdots \geqq G_\tau$ and will sometimes be referred to as $\{G_i\}$. Various intermediate groups are created during the construction of $p'(q)$; these groups are designated by $\{G_i^j\}$ and are distinguished by the superscript j.

PARTITION begins by *blocking intercepts at* $x = q$. That is, PARTITION puts the first 20 groups $G_1, \cdots, G_{20}$ into block 1, the next 20 into block 2, etc. The lines in each block are *regrouped* according to their intercepts $y_s(q)$ at q. Thus G^1_{20i-19} contains the T largest intercepts at q among the block i lines, G^1_{20i-18} the next T largest intercepts at q among the block i lines.

LEMMA 6. *Let* $\{G_i\}$ *be a partition of size* T *at* r. *The cost of forming* $\{G_i^1\}$ *is* $O(|\{G_i\}|)$.

Proof. In each block we find the Tth largest $y_i(q)$, the $(2T)$th largest, and so on up to the $(19T)$th largest, each in time $O(T)$. Once these 19 group dividers are determined, each element in the block may be placed into the correct group in constant time, and since there are $O(n/T)$ blocks, the linear time bound holds. □

Furthermore, since lines are blocked in the order in which they appear in $p(r)$, lines in the same group at $x = q$ satisfy (5). With each line j, we will maintain $\gamma(j)$, its index at $x = r$ (i.e., line j is the $\gamma(j)$th line in descending order in $p(r)$).

Here is a quick overview of how PARTITION will continue after blocking. Within each block, the groups $\{G_i^1\}$ are correctly ordered at q. This is a good start in constructing $p'(q)$. Call a pair of groups G_i^1 and G_j^1 *out of order* if G_i^1 is above G_j^1 at r (i.e., every line in G_i^1 is above every line in G_j^1), but G_j^1 is above G_i^1 at q. The idea is that not too many pairs of groups from different blocks can be out of order, unless q is far from r. If groups G_i^1 and G_j^1 were in the same block at r, their images at q "should" differ by less than one block; that is, few groups should be out of order according to the definition above. If the group images differ by more than one block, then many groups are out of order and $|\text{rank}\,(r) - \text{rank}\,(q)|$ would have to be large. Thus we may (A) *extract* a large subset $\{G_i^2\}$ of the groups $\{G_i^1\}$ which are already properly ordered at q. The remaining elements, comprising at most $3n/5$ lines, may be (B) *reblocked* and treated in the same manner, recursively, until they are all in groups $\{G_i^3\}$ which are properly ordered at q. Then we (C) *merge* these two collections of ordered groups of size T into one collection $\{G_i^4\}$ of ordered groups of size at most $2T$, all in linear time. Finally, using another $O(n)$ steps, we can (D) *reorganize* the groups in $\{G_i^4\}$ to form $p'(q)$. We now give the details.

(A) *The procedure* EXTRACT $(\{G_i^1\}, T)$ takes ordered groups of size T which are blocked at $x = q$ and, in linear time, extracts a collection $\{G_i^2\}$ of groups of size T which are ordered at q. If less than two-fifths of the groups $\{G_i^1\}$ are extracted, $|\text{rank}\,(r) - \text{rank}\,(q)|$ must be at least $3nT/2$ and TOOFAR is set to 1.

The procedure EXTRACT will first construct two lists. One list contains the lines in the odd-numbered blocks, and the other list contains the lines in the even-numbered blocks. The lists are formed by stack operations. Each partially formed list has the property that the groups appearing in it are correctly ordered at $x = q$. Consider the list corresponding to the odd-numbered blocks; the even-numbered blocks are handled similarly. The groups in the odd-numbered blocks are examined from the lowest-numbered group to the highest-numbered group. The group on the top of the stack

has the highest index and corresponds to the lines currently in the stack with the T smallest intercepts at q. The group on the bottom has the lowest index and corresponds to the T lines with the largest intercepts. For each group G_i^1, compute

$$\phi_i = \min(y_s(q), s \in G_i^1), \qquad \Phi_i = \max(y_s(q), s \in G_i^1).$$

This takes linear time because in each of $\tau = \lceil n/T \rceil$ groups, we are selecting from T elements.

Starting with blocks 1 and 3, push block 1 onto the stack. We compare the current top of the stack, group G_{20}^1, to G_{41}^1. If $\phi_{20} \geqq \Phi_{41}$, not only are these groups correctly ordered at q, but all pairs of groups in these two blocks are correctly ordered as well. We push block 3 onto the stack and move on to block 5. Otherwise, these groups may not be correctly ordered. We pop the group on the top of the stack, discard group G_{41}^1, and next compare group G_{19}^1 with group G_{42}^1. This process continues until we either find a pair of groups in correct order or the stack is emptied. In the former case, we push the remainder of the block onto the stack, so we now have a stack of (at most 40) undeleted groups which are correctly ordered at $x = q$ and can process block 5. In the latter case, block 5 is pushed on the stack and block 7 is considered.

We process the next odd block, say $2i+1$, comparing group G_{40i+1}^1 (the first group of block $2i+1$) with the group G_j' on the top of the stack. If $\Phi_{40i+1} \leqq \phi_j'$, block $2i+1$ is pushed on the top of the stack and we move on to the next odd block, $2i+3$. Otherwise, these groups may not be correctly ordered at q. We pop G_j' from the stack, discard G_{40i+1}^1 from the block, and compare the group on the current top of the stack with the next group in block $2i+1$. Whenever the line in the group on the top of the stack with minimum intercept crosses the line in the current group in block $2i+1$ with maximum intercept, the stack is popped and the current group is block $2i+1$ is deleted. If a pair is found to be correctly ordered, that group and the rest of its block are pushed on the stack and we move to the lowest-indexed group in the next odd block. If the last group in block $2i+1$ is deleted, we move on to block $2i+3$. If the stack is ever emptied, either the remainder of the current odd block or the entire next odd block is pushed on the stack, whichever is appropriate.

The "odd" stack forms a collection of groups, $\{G_i^o\}$, which were not deleted from odd blocks and which are properly ordered with respect to each other at q, and the "even" stack forms a collection, $\{G_j^e\}$, of groups which were not deleted from even blocks and which are properly ordered with respect to each other at q. The time taken to form these two lists is clearly $O(n)$. If we started with 20 groups or fewer, no deletion would have been necessary. Block 1 is $\{G_i^o\}$ and the groups in block 2 comprise $\{G_j^e\}$. In the second phase of EXTRACT, these two lists are merged into one list whose groups are properly ordered at q.

LEMMA 7. *$\{G_i^0\}$ and $\{G_j^e\}$ can be merged into a collection $\{G_i^m\}$ of groups in time $O(|G_i^o\}|+|\{G_j^e\}|)$, where each group in $\{G_i^m\}$ has size at most $2T$, and if $c \in G_i^m$, $d \in G_j^m$ and $i<j$, then $y_c(q) \geqq y_d(q)$. Furthermore, $\{G_i^m\}$ satisfies invariant* (5).

Proof. A procedure MERGE $(\{G_i^o\}, \{G_j^e\})$ combines $\{G_i^o\}$ and $\{G_j^e\}$ into one collection $\{G_i^m\}$ of ordered groups of size at most $2T$. Comparing G_i^0 with G_j^e, there are three cases, (i) the groups do not overlap, (ii) one group is "inside" the other, and (iii) they partly overlap. In case (i) suppose $\max(y_s(q), s \in G_i^o) \leqq \min(y_s(q), s \in G_j^e)$. Then G_j^e is taken next into the merge and $j \leftarrow j+1$. If G_i^o is above G_j^e, G_i^o is taken next into the merge and $i \leftarrow i+1$.

In case (ii), G_j^e is "inside" G_i^o if $\max(y_s(q), s \in G_i^o) \geqq \max(y_s(q), s \in G_j^e)$ and $\min(y_s(q), s \in G_i^o) \leqq \min(y_s(q), s \in G_j^e)$. We take next G_j^e and all those $s \in G_i^0$ for which $y_s(q) \geqq \min(y_s(q), s \in G_j^e)$ as the group G which will be added to the merge.

To satisfy (5), lines must appear in G in the order in which they appear in $p(r)$. Therefore we form G by merging the lines in G_j^e with those that were selected from G_i^o according to the index γ at $x=r$. Line $u \in G_j^e$ is taken into G before $v \in G_i^o$ only if $\gamma(u) < \gamma(v)$. It takes $O(T)$ steps to form G and add it to the merge. We now remove the elements that were selected from G_i^o, set $j \leftarrow j+1$ and continue merging. The case where G_i^o is inside G_j^e is similar. In both instances, the group G added to the merge is of size at most $2T$. Also, at least a full-sized group or the remainder of a full-sized group must be added to the merge at each step, so the number of merge steps is bounded by the number of groups.

Finally, for case (iii) suppose that G_i^o has both a larger max $(y_s(q))$ and a larger min $(y_s(q))$ than G_j^e. Then we take next into the merge G_i^o and all those $s \in G_j^e$ for which $y_s(q) \geqq \min(y_s(q), s \in G_i^o)$, modify G_j^e accordingly, and set $i \leftarrow i+1$. Once again the group G taken next into the merge has lines ordered by the group indices at $x=r$. If G_j^e is partly above G_i^o, the processing is similar. As in case (ii), the number of merge steps is bounded by the number of groups involved in case (iii) merges. In both cases the set taken into the merge has size at most $2T$. The merge compared $O(n/T)$ groups, each in time $O(T)$, so it took linear time in all. □

At this point we have a collection $\{G_i^m\}$ of groups of size at most $2T$ which are ordered at $x=q$ and which maintain the invariant in (5). Within each group, i precedes j only if $\gamma(i) < \gamma(j)$. The final phase of EXTRACT is to form $\{G_i^2\}$ from $\{G_i^m\}$, where the groups in $\{G_i^2\}$ all have size T. Some care is required to adjust these groups in linear time to form the ordered groups $\{G_i^2\}$, principally because of the invariant in (5). The procedure SIZET$(T, \{G_i^m\})$ performs this task. Its action is described by the following lemma.

LEMMA 8. *$\{G_i^m\}$ may be reorganized into $\{G_i^2\}$ in time $O(|\{G_i^m\}|)$, where each group in $\{G_i^2\}$ has size exactly T, $\{G_i^2\}$ satisfies invariant (5), and if $c \in G_i^2$, $d \in G_j^2$ and $i<j$, then $y_c(q) \geqq y_d(q)$.*

Proof. Procedure SIZET forms groups of size T in the following way. In general, the ith group in $\{G_i^2\}$, G_i^2, is formed from the s lines in G_{j-1}^m with the smallest $y(q)$, in order, a whole group G_j^m of size s', and the $T-s-s'$ lines from G_{j+1}^m with largest $y(q)$, in order. To preserve (5) for G_i^2, we use order in $p(r)$ to decide how to organize the lines in G_{j-1}^m, G_j^m, and G_{j+1}^m. Since only three adjacent groups in $(G_i^m\}$ need to be examined to form each new group in $\{G_i^2\}$ and each group in $\{G_i^m\}$ has size at most $2T$, SIZET runs in time $O(|\{G_i^m\}|)$.

To see that only three groups are sufficient, we show that every pair of adjacent groups G_j^m and G_{j+1}^m in $\{G_i^m\}$ satisfies $|G_j^m|+|G_{j+1}^m| \geqq T$. Recall that the algorithm MERGE compares group fragments G_{au}^o with G_{bu}^e. (The u in the subscript indicates that these fragments are involved in the uth step of MERGE.) We first claim that one of G_{au}^o and G_{bu}^e must have been a full-sized group.

When the merge begins, both G_{a1}^o and G_{b1}^e are full-sized groups. Suppose that at the time of the uth merging step, at least one of G_{au}^o and G_{bu}^e is a full-sized group. In the $(u+1)$th step of the merge, either all of G_{au}^o will be taken or all of G_{bu}^e will be taken, and the other group may be fragmented. Without loss of generality, suppose G_{au}^o is taken; then $G_{a(u+1)}^o$ is a full-sized group, justifying the claim.

If a full-sized group is taken next into the merge, $|G_j^m| \geqq T$, so $|G_j^m|+|G_{j+1}^m| \geqq T$. If a full-sized group G' is not taken into the merge, then G' may be fragmented. On the next step, either the remains of G' are taken or a full-sized group is taken. In either case, $|G_j^m|+|G_{j+1}^m| \geqq T$. □

An important property of EXTRACT is that *at least $2n/5$ lines are represented in the collection $\{G_i^2\}$ of groups, correctly orderd at q*, unless the ranks of r and q differ

by at least $3nT/2$. Let n_j be the number of groups that were deleted from block j, $0 \leqq n_j \leqq 20$. A group G_i^1 is deleted only if a line in G_i^1 crosses a line in a group G_j^1 at least 2 blocks away (i.e., separated by at least one block). Suppose that both G_i^1 and G_j^1 are not the top or bottom groups of their blocks, where we assume that the top group has the largest intercepts at q and the bottom group has the smallest intercepts. For example, let G_i^1 be separated from the bottom of its block by sub-block A and let G_j^1 be separated from the top of its block by sub-block B. Then all the $20T$ lines in the intervening block between G_i^1 and G_j^1 must either cross all of the lines in A, all of the lines in B, or both. This implies that if n_j is 3, we may account for $10T^2$ inversions; we have divided 20 by 2 because it pertains to the two groups, G_i^1 and G_j^1. Therefore the total number of inversions between $p(r)$ and $p'(q)$ must be at least

$$I = \sum_{j=1}^{n/(20T)} (n_j - 2)5T^2;$$

here we have divided 10 by 2 to adjust for double counting of inversions with lines in the intervening block. If at least $3n/(5T)$ groups $\{G_i^1\}$ were deleted, $I \geqq 5nT/2$. Since $p(r)$ and $p'(q)$ are in error by less than $nT/2$ each, this implies that $|\text{rank}\,(r) - \text{rank}\,(q)| \geqq 3nT/2$. It is useful to make a slightly more general statement.

LEMMA 9. *Let i be a positive integer between 0 and $\lfloor \log n/T \rfloor$. Suppose h_i lines, ordered according to $p(r)$, are reblocked into a partition $\{H_i\}$ of size 2^iT with respect to ordering at q by the procedure described in Lemma 6. In time $O(h_i)$,* EXTRACT $(\{H_i\}, 2^iT)$ *will determine groups $\{H_i^*\}$ of size 2^iT that are ordered at $x = q$ and eliminate h_{i+1} lines. Furthermore, if $|\text{rank}\,(r) - \text{rank}\,(q)| < 3nT/2$, then*

$$h_{i+1} < \frac{n}{2^{i+1}} + \frac{h_i}{10}.$$

Proof. The time bound to construct $\{H_i^*\}$ was justified in the description of the "odd" and "even" stacks along with Lemmas 7 and 8. If $|\text{rank}\,(r) - \text{rank}\,(q)| < 3nT/2$, then $|\rho_r - \rho_q'| < 5nT/2$. As before, n_j denotes the number of groups deleted from block j. The number of inversions I between $p(r)$ and $p'(q)$ is bounded so that

$$I = \sum_{j=1}^{h_i/(20(2^iT))} (n_j - 2)5(2^iT)^2 \leqq |\rho_r - \rho_q'| < 5nT/2.$$

However,

$$I = 5(2^iT)\left[(2^iT)\sum_{j=1}^{h_i/(20(2^iT))} n_j\right] - 10h_i(2^iT)^2/(20(2^iT))$$

$$= 5(2^iT)h_{i+1} - h_i(2^iT)/2 < 5nT/2.$$

This inequality can be simplified to give

$$h_{i+1} < \frac{n}{2^{i+1}} + \frac{h_i}{10}.$$

□

If procedure PARTITION is called on a list of size h_i but h_{i+1} does not satisfy the inequality, TOOFAR is set to 1 because $|\text{rank}\,(r) - \text{rank}\,(q)| \geqq 3nT/2$.

(B) So far EXTRACT has organized at least $2n/5$ of the lines into groups $\{G_i^2\}$ of size T, ordered at $x = q$, or set TOOFAR = 1. In the latter case PARTITION halts (see Fig. 2), but in the former, we use PARTITION recursively.

For $i = 1, \cdots, \tau$ let G_i^d denote those lines in G_i in the original order in $p(r)$ that have been deleted; clearly $0 \leqq |G_i^d| \leqq T$. This collection is easily obtained by marking

```
PARTITION (s, T, {G_i}, {G'_i}, TOOFAR)
⟨1⟩  h_s ← |{G_i}|
⟨2⟩  {G_i^1} ← BLOCK-REGROUP at q
⟨3⟩  {G_i^2} ← EXTRACT ({G_i^1}, T)
⟨4⟩  IF h_s − |{G_i^2}| ≧ n/2^{s+1} + h_s/10 THEN
⟨5⟩    TOOFAR ← 1
⟨6⟩  ELSE
⟨7⟩    {G_i^D} ← lines deleted from {G_i^1}, partitioned at r
⟨8⟩    T' ← 2T
⟨9⟩    PARTITION (s+1, T', {G_i^D}, {Ḡ_i^3}, TOOFAR)
⟨10⟩   IF TOOFAR ≠ 1 THEN
⟨11⟩     {G_i^3} ← HALVE ({Ḡ_i^3})
⟨12⟩     {G_i^4} ← MERGE ({G_i^2}, {G_i^3})
⟨13⟩     {G'_i} ← SIZET(T, {G_i^4})
⟨14⟩   END IF
⟨15⟩ END IF
```

FIG. 2. *The Procedure* PARTITION.

deleted items in the EXTRACT process and removing unmarked lines from permutation $p(r)$. If $i=0$ and $h_0=n$ in Lemma 9, then

$$h_1 \equiv \sum_{i=1}^{\tau} |G_i^d| < \frac{3n}{5}.$$

We can reorganize these lines into $h_1/(2T)$ groups of size $2T$ that are properly ordered according to $p(r)$. These reorganized groups, denoted by $\{G_i^D\}$, form a partition of deleted lines of size $2T$ at r. We invoke PARTITION $(s+1, 2T, \{G_i^D\}, \{\bar{G}_i^3\}, \text{TOOFAR})$. Parameter $s+1$ indicates that the recursion is one level deeper. The groups in $\{\bar{G}_i^3\}$ form a partition of size $2T$ at q of the lines not included in $\{G_i^2\}$, or else TOOFAR $=1$. In the latter case $|\text{rank}(q)-\text{rank}(r)| \geqq 3nT/2$ and PARTITION is aborted. In the former case, procedure HALVE is applied to $\{\bar{G}_i^3\}$ to form $\{G_i^3\}$, a partition of the lines that were deleted from $\{G_i^1\}$ of the original size T, at $x=q$. (In this case, the number of inversions counted by HALVE is ignored. The number of inversions induced by PARTITION will be counted entirely by procedure COUNT.)

(C) Merging $\{G_i^2\}$ and $\{G_i^3\}$ gives a collection $\{G_i^4\}$ consisting of groups of size at most $2T$, correctly ordered at q. As with the merge of $\{G_i^o\}$ and $\{G_j^e\}$, the current one is simplified by the structure of the sets involved. The MERGE procedure described earlier may be used here as well.

(D) The final step of the procedure PARTITION is to reorganize $\{G_i^4\}$ into $\{G'_i\}=p'(q)$. This task can be done by procedure SIZET$(T, \{G_i^4\})$ in $O(n)$ time.

A simplified description of this algorithm is presented in Fig. 2. The only portion of procedure PARTITION that still needs to be described is procedure HALVE.

4.3. The procedure HALVE. The procedure HALVE converts the partition $p'(q)$ of size T at q into a partition $p''(q)$ of size $T/2$ at q and computes the number of additional inversions in $p''(q)$. For each group G'_i in $p'(q)$, we find $\mu=$ median $(y_s(q), s \in G'_i)$ and then assign each line to the correct subgroup depending on whether $y_s(q)$ is less or greater than μ, using the following strategy. Let $i_1, \cdots, i_T$ be the T lines in G'_i in the order in which they appear in $p'(q)$. The elements in group G'_i will be redistributed into two groups in $p''(q)$. These groups will be denoted by A and B; group A will have index $2i-1$ in $p''(q)$, and group B will have index $2i$. If $y_{i_1}(q)$ is greater than μ, it is placed in position j_1 (initially 1), the first available position in the upper half, A, and j_1 is increased by 1. Otherwise $y_{i_1}(q)$ is placed in position j_2

(initially 1), the first available position in the lower half, B, and j_2 is increased by 1, etc., until all T lines have been placed into the correct half. The group G_i' has now been replaced by groups A and B of size $T/2$ each, and they are correctly ordered at q.

While modifying $p'(q)$ in this way, the count of the number of inversions at q is updated to reflect these changes. Specifically, if line i_s is assigned to position j_2 in the lower half, it can only be involved in inversions with lines in G_i' that were below it but which will now be assigned to the upper half-group A. There are exactly $T/2-j_1+1$ such lines, so we change ρ_q by

$$(T/2-j_1+1)/2.$$

Similarly, line i_s that is assigned to position j_1 in the upper half can only be involved in inversions with lines in G_i' that were orginally above it but which have just been assigned to the lower half-group, B. As there are exactly j_2-1 such lines, we change ρ_q by

$$(j_2-1)/2.$$

The change is an increase when SIDE $=-1$ and a decrease otherwise. Division by 2 reflects the double counting of each inversion. For each group, the cost of finding the median, partitioning, and adjusting ρ_q is $O(T)$, or $O(n)$ overall.

This proves the following lemmas.

LEMMA 10. *A partition $p'(q)$ of size T at q can be converted into a partition $p''(q)$ of size $T/2$ at q in linear time. Furthermore, the number of additional inversions between $p'(q)$ and $p''(q)$ can be computed in linear time.*

LEMMA 11. *Let r be an active reference point, let $p(r)$ be a partition of size T at r, and let q be the query point. If* $|\text{rank}(r)-\text{rank}(q)|<3nT/2$, *procedure* PARTITION *will construct $p'(q)$, a partition of size T at q, in linear time. If* $|\text{rank}(r)-\text{rank}(q)|\geqq 3nT/2$, *procedure* PARTITION *will halt in linear time with either $p'(q)$ or* TOOFAR *set to* 1.

Proof. The correctness of the algorithm follows from Lemmas 6-10. To apply Lemma 9, note that whenever PARTITION $(s, T', \{G_i\}, \{G_i'\}, \text{TOOFAR})$ is invoked, $T'=2^sT$, T being the size of the partition in the original call. This is clear when $s=0$, and inductively it is true because the partition size is doubled before each recursive call.

For the time complexity, suppose that $|\text{rank}(r)-\text{rank}(q)|<3nT/2$. Each statement in PARTITION has been shown to have cost $O(h)$ (Lemmas 6-10), where h is the current size of the list. Initially, PARTITION is called for a list of size $h_0=n$. By Lemma 9, the size of the list in the $(i+1)$th recursive call statisfies

$$h_{i+1}<\frac{n}{2^{i+1}}+\frac{h_i}{10}.$$

As a result, the total cost of the algorithm is bounded by $H=c\sum_{i=0}^{\infty}h_i<20cn/9$.

If $|\text{rank}(r)-\text{rank}(q)|\geqq 3nT/2$, procedure PARTITION will either construct a partition of size T at q, or it will set TOOFAR to be 1. TOOFAR is set to 1 as soon as the recurrence relation in Lemma 9 is not satisfied for the first time. Consequently, PARTITION will have made at most a linear number of steps when TOOFAR is set to 1, and it will make no more recursive calls. If TOOFAR is not set to 1, the algorithm successfully constructs a partition of size T at q in linear time. □

4.4. The procedure COUNT. The purpose of procedure COUNT is to count p_q' the number of inversions in $p'(q)$. It will do this by computing $\Delta=|\rho_q'-\rho_r|$ if $|\rho_q'-\rho_r|<3nT/2$. In this section, we assume that $r=r_L$; analogous lemmas and algorithms can be devised when $r=r_R$.

LEMMA 12. (i) *Every inversion in* $p(r_L)$ *also occurs in both* $p(q)$ *and* $p'(q)$; *and* (ii) *inversions present in either* $p(q)$ *or* $p'(q)$ *actually occur between the corresponding* $y_s(q)$. *Furthermore,* (iii) *if* $\text{rank}(q) < k$, $|\rho'_q - k| \leqq nT$.

Proof. When $p(r_L) = (1, 2, \cdots, n)$, property (5) implies that (i) and (ii) are true for the first $p(q)$ and $p'(q)$. Assume that (i) and (ii) are true at $p(r)$, and let $a < b$ be two lines in $p(r)$. By (5), if a and b appear in the same group of $p(q)$, they are in the same order as in $p(r)$. Therefore, (i) and (ii) must still hold for a and b at $p(q)$. If a and b appear in different groups of $p(q)$, there are two possibilities. First, a and b may be in the same relative order in $p(q)$ as in $p(r)$. In this case, an inversion was neither created nor destroyed, so (i) and (ii) are maintained. In the second case, a and b have inverted between $p(r)$ and $p(q)$. Properties (i) and (ii) at $p(r)$ imply that a and b invert in $p(q)$ for the first time. Claim (ii) holds for $p(q)$ because the ordering information given by the partition implies that a and b must have crossed. In fact, since properties (i) and (ii) are true at $p(r)$, a similar argument shows that (i) and (ii) also hold for $p'(q)$ as well.

Since $p'(q)$ contains all of the inversions in $p(r)$, $\rho'_q \geqq \rho_r$. If both ρ'_q and ρ_r are less than k, then $|\rho'_q - k| \leqq |\rho_r - k| \leqq nT$. If ρ'_q exceeds k, then (2) implies that ρ'_q cannot exceed k by more than $nT/2$ if $\text{rank}(q) < k$. Therefore, $|\rho'_q - k| \leqq nT$. □

By Lemma 12, if SIDE $= -1$, then $\rho'_q = \rho_r + \Delta$ since every inversion in $p(r)$ also occurs in $p'(q)$.

LEMMA 13. *If* $|\rho'_q - \rho_r| \geqq 3nT/2$, *then* q *and* r *must be on opposite sides of* t_k.

Proof. Lemma 12 implies that $\rho'_q \geqq \rho_r$, so $\rho'_q - \rho_r \geqq 3nT/2$. By (3), $|\rho_r - k| \leqq nT$, so $\rho'_q - k \geqq 3nT/2 + \rho_r - k \geqq nT/2$. By (2), $|\text{rank}(q) - \rho'_q| < nT/2$, so $\text{rank}(q) - k > 0$ and $\text{sign}(r - t_k) \neq \text{sign}(q - t_k)$. □

Lemma 12 shows that the invariant (5) simplifies the counting procedure by allowing us to calculate the number of inversions in $p'(q)$ by computing Δ. Let $\xi(i)$ be the index of the group in $p'(q)$ containing line i. The value of $\xi(i)$ was found by procedure PARTITION. By the invariant (5), an inversion involving i and j between $p(r)$ and $p'(q)$ can occur only if $\xi(i) \neq \xi(j)$.

Procedure COUNT will examine the lines in $p(r)$ in descending order, and it can be separated into four stages. First, COUNT uses a data structure that must be (A) *initialized.* This data structure contains a collection of "bins" of size of at least T but less than $3T$; each individual bin is ordered according to $p(r)$, but different bins are ordered according to q. For each line $p(r)$, we (B) *find* its correct location in the data structure and place it into the appropriate bin. When a bin reaches capacity $3T$, we (C) *split* the bin into two new bins, each with size at least T. After all lines are processed, we (D) *finish* by redistributing the elements in the bins to form $p'(q)$. Inversions are counted in steps (B), (C), and (D).

(A) As stated in the previous paragraph, COUNT will use a data structure. This data structure $u(q)$ is called the binned partition at q, and it consists of a series of bins U_1, U_2, etc. A bin can either be empty or contain at least T but less than $3T$ lines ordered according to $p(r)$. As with partitions, if $c \in U_i$ and $d \in U_j$, $i < j$, then $y_c(q) \geqq y_d(q)$. If there are k nonempty bins, they must have index $1, 2, \cdots, k$.

Each bin in $u(q)$ has three parameters associated with it. The *size* of a bin is the number of elements it contains. Variable *min* contains the smallest $\xi(i)$ in the bin (the only possible exception is $\min(U_1)$, which is always defined to be 1). For each U_j, the closed interval from $\min(U_j)$ to $\min(U_{j+1}) - 1$ is called the *range* (if U_k is the highest numbered bin, its range extends from $\min(U_j)$ to τ). Finally, each bin U_j contains a pointer *nextbin* to U_{j-1}. The binned partition $u(q)$ is simply a linked list which is entered at the highest-numbered nonempty bin.

In addition to the three parameters associated with each bin, a variable Δ_u is associated with $u(q)$. Intuitively, Δ_u contains the current number of inversions induced by forming $u(q)$ from $p(r)$. Throughout the procedure COUNT, two invariants will hold: (a) the ranges in the nonempty bins are disjoint and the union of the ranges is 1 to τ; and (b) lines in a particular bin are positioned according to order in $p(r)$.

To initialize $u(q)$, the elements in G_1 in $p(r)$ are placed into U_1, the size of U_1 is set to T, min is set to 1 and Δ_u is set to 0. The initialization clearly takes $O(T)$ time.

(B) Let i be the next line examined by the procedure COUNT. At the time line i is examined, all the lines preceding i in $p(r)$ have been placed into $u(q)$, and no bin has size $3T$. Let $u(q) = U_1 U_2 \cdots U_k$. If $\xi(i)$ is in the range of U_k, then line i is placed at the bottom of U_k. Otherwise, $\xi(k)$ is less than min (U_k), and U_{k-1} is examined. If U_{k-1} is examined, line i must have crossed every line in U_k, so Δ_u is incremented by size (U_k). If $\xi(i)$ lies in the range of U_{k-1}, then line i is placed at the bottom of U_{k-1}. Otherwise, Δ_u is incremented by size (U_{k-1}) and U_{k-2} is examined. In general, $\xi(i)$ is compared with the range of U_j. If $\xi(i)$ is inside the range of U_j, then it is placed at the bottom of U_j and the find routine halts. If $\xi(i)$ is less than min (U_j), then Δ_u is incremented by size (U_j) and U_{j-1} is examined. It is important to note that at least T inversions are counted each time $\xi(i)$ is outside the range of U_j.

Once line i is placed into bin U_j, the size condition may be violated because size $(U_j) = 3T$. In this case, the split routine is performed.

(C) The split routine is intended to separate a bin of size $3T$ into two bins of size at least T. It is very similar to the procedure HALVE. An element l can be used to split bin U_j into two bins, U_{j1} and U_{j2}. U_{j1} contains the lines v in U_j for which $\xi(v) < \xi(l)$, and they appear in their order at $p(r)$. U_{j2} contains the remaining lines in their order at $p(r)$. Since there are $3T$ lines in U_j, U_j must contain at least 3 distinct values of ξ.

The partitioning element is chosen in the following manner. Once size $(U_j) = 3T$, the median line μ in U_j with respect to the ordering at q is computed. U_j will be partitioned by one of two lines, l_1 or l_2. Line l_1 is the line in U_j with smallest index in $p(r)$ satisfying $\xi(l_1) = \xi(\mu)$, and line l_2 is the line in U_j with largest index in $p(r)$ satisfying $\xi(l_2) = \xi(\mu)$. Since at most T lines in U_j can be in group $\xi(\mu)$, at least one of l_1 and l_2 splits U_j into two bins of size at least T. The number of inversions induced by split is counted in the same way as the inversions detected in HALVE. Note that the split routine did not separate any lines in U_j with the same value of ξ, so the invariants are maintained.

(D) After all lines have been placed in $u(q)$, the binned partition has the following form. Each bin U_j contains at least T but less than $3T$ lines. By invariant (a), lines with the same value of ξ must appear in the same bin. Consequently, size (U_j) must be a multiple of T, so U_j contains either T or $2T$ lines. In the former case, the number of inversions with respect to lines in U_j has already been computed in steps (B) and (C). In the latter case, U_j must be split and inversions counted in the same manner as part (C). This routine takes at most $O(T)$ time per bin, and $O(n)$ time overall.

LEMMA 14. *If* $|\rho_r - \rho'_q| < 3nT/2$, *the* COUNT *procedure correctly computes the number of inversions between* $p(r)$ *and* $p'(q)$ *in linear time.*

Proof. By the invariants (a) and (b), correctness follows from a straightforward induction on the number of applications of find and split. The time complexity requires a bit more care.

Parts (A) and (D) have already been shown to take $O(n)$ time. For part (B), the cost of each examination of U_j is $O(1)$ because $u(q)$ is implemented as a linked list. Call such an examination a *find operation.* Suppose that the find operation is performed

n_i times for line i. All but the first find operation guarantees T inversions, so

$$\sum_{i=1}^{n} (n_i - 1)T < \frac{3nT}{2}.$$

Therefore,

$$\sum_{i=1}^{n} n_i < \frac{5n}{2}.$$

The total time spent on the find operations is $O(n)$.

For part (C), each split operation takes $O(T)$ time and increases the number of nonempty bins by one. Since each bin must have size T, there are at most n/T nonempty bins. At most n/T split operations can be done, so the total time for the splits is $O(n)$. □

4.5. The proof of Theorem 4. We first show that the algorithm presented in Fig. 1 correctly finds t_k. If the test at ⟨6⟩ fails, procedure PARTITION is applied. At the active reference point r, say r_L (similar arguments can be made if $r = r_R$),

$$|\rho_{(r,T)} - k| \leqq nT, \qquad |\rho_{(r,T/2)} - k| > nT/2.$$

If the new query point q is on the same side of t_k as r, then Lemma 11 implies that PARTITION will succeed in finding $p'(q)$. By Lemma 14, COUNT will determine the number of inversions in $p'(q) = p(q, T)$, $\rho_{(q,T)}$. Lemma 12(iii) implies that $|\rho_{(q,T)} - k| \leqq nT$, so (3) is satisfied. Procedure HALVE will be run to see if (4) is satisfied as well. If (4) is satisfied, then $p(q) = p(q, T)$ and q can be resolved. If (4) is not satisfied, then $|\rho_{(r,T/2)} - k| \leqq nT/2$. HALVE will be executed until some T satisfies both (3) and (4). Note that unless $|\text{rank}(q) - k| \leqq n/2$, some T will satisfy (4) since there is some separation between q and t_k. Therefore, if r and q are on the same side of t_k, $p(q)$ will be determined in line ⟨15⟩.

The new query point q can be on the opposite side of t_k from r. There are several ways that this can be detected. First, Lemma 11 implies that PARTITION may detect that $|\text{rank}(r) - \text{rank}(q)|$ is large and set TOOFAR = 1. Lemma 14 shows that COUNT will either succeed in determining the number of inversions in $p(q, T)$, or it will halt if there are at least $3nT/2$ inversions between $p(r, T)$ and $p(q, T)$. By Lemma 13, $3nT/2$ inversions implies that r and q are on opposite sides of t_k. If both PARTITION and COUNT succeed in constructing and counting $p(q, T)$, then we will determine the relative order of q and t_k by the approximation.

Since we are assuming that the partition $p'(q)$ was computed by modifying the partition at r_L, invariant (5) implies that the number of inversions present in each refinement cannot decrease as T decreases. That is, $\rho_{(q,T/2)} \geqq \rho_{(q,T)}$. This implies that once there are more than k inversions present in an approximation, q and r_L must be on opposite sides of t_k. The number of executions of HALVE to detect that $\rho_{(q,T/2)} > k$ cannot be greater than the number of executions of HALVE to satisfy (3) and (4). This explains the third condition in line ⟨12⟩ as a stopping condition for the calibration loop.

If r and q are on the same side of t_k, $p(q)$ is determined in line ⟨15⟩. If r and q are not on the same side of t_k, then the test in ⟨17⟩ is true, and $p(q)$ is determined in line ⟨23⟩. In conclusion, the algorithm correctly computes the partition $p(q)$ and uses (4) to answer the question delivered by the network.

The procedures PARTITION, HALVE, and COUNT all took linear time. By the construction of the algorithm, PARTITION and COUNT are invoked at most twice

for any given q. Therefore, the total amount of work done by PARTITION and COUNT over the entire algorithm is $O(n \log n)$ steps. HALVE is invoked immediately after PARTITION to evaluate invariant (4). However, the procedure HALVE may be run many more times for any particular q. Suppose that r_L is the active reference point and that HALVE is called $s > 0$ times in line ⟨13⟩ of the algorithm in Fig. 1. If q and r_L are on the same side of t_k, then the new value of T_L is $T_L/2^s$. In the remainder of the algorithm, Lemma 12(iii) implies that T_L can never be increased because the upper bound in (3) is satisfied. If q and r_L are on opposite sides of t_k, then the partition $p(q)$ is computed with respect to r_R. Since the number of executions of HALVE to establish (3) and (4) is greater than or equal to the number of executions of HALVE actually used to determine the relative positions of q and t_k, consider the value of T_L when $|\rho_{(q,T_L/2)} - k| > nT_L/2$ and $|\rho_{(q,T_L)} - k| \leqq nT_L$. From the latter equation, we can deduce that $|\text{rank}\,(q) - k| < 3nT_L/2$.

The new value of T_R must satisfy both (3) and (4). Lemma 5 implies that properly calibrated values of T_R must satisfy $|\text{rank}\,(q) - k| > nT_R/4$. Therefore, $nT_R/4 < 3nT_L/2$, so $T_R < 6T_L$. Recall that $T_R \geqq T_L$ since r_L is the active reference point. For invariants (3) and (4) to be established, HALVE must be run at least $s-2$ times until $T_R < 6T_L$.

This shows that if line ⟨13⟩ is executed s times and r and q are on opposite sides of t_k, then line ⟨21⟩ will be executed at least $s-2$ times. Either T_L or T_R is reduced by a factor of at least 2^{s-2}, implying that the number of calls to procedure HALVE is at most $O(\log n)$. This proves the asserted $O(n \log n)$ time bound and Theorem 4.

5. General remarks. This paper presented an optimal-time algorithm to compute the kth largest or smallest slope determined by a set of points in the plane. The control structure of the algorithm is based on the parametric search technique devised by Megiddo. To obtain the optimal-time algorithm, however, an approximation idea was incorporated. This general paradigm may be useful for other problems to which Megiddo's technique is amenable.

The ideas behind the algorithm described have also been applied successfully to other selection problems [10]. In particular, the kth smallest interdistance between n points in d-space, using the L_∞ metric, may be found in time $O(n(\log n)^d)$ [11]. Unfortunately nontrivial lower bounds are not known for this problem.

REFERENCES

[1] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *Sorting in c* Log *n parallel steps*, Combinatorica, 3 (1983), pp. 1-19.

[2] J. L. BENTLEY AND T. OTTMANN, *Algorithms for reporting and counting geometric intersections*, IEEE Trans. Comput., 28 (1979), pp. 643-647.

[3] M. BLUM, R. W. FLOYD, V. PRATT, R. L. RIVEST, AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System Sci., 7 (1973), pp. 448-461.

[4] B. CHAZELLE, *New techniques for computing order statistics in Euclidean space*, in Proc. ACM Syposium on Computational Geometry, 1985, pp. 125-134.

[5] R. COLE, *Slowing down sorting networks to obtain faster sorting algorithms*, J. Assoc. Comput. Mach., 34 (1987), pp. 200-208.

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

[7] N. MEGIDDO, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852-865.

[8] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, New York, Berlin, 1985.

[9] A. REISER, *A linear selection algorithm for sets of elements with weights*, Inform. Process. Lett., 7 (1978), pp. 159-162.

[10] J. S. SALOWE, *Selection problems in computational geometry*, Ph.D. thesis, Department of Computer Science, Rutgers University, New Brunswick, NJ, 1987.

[11] ———, *L-infinity interdistance selection by parametric search*, Inform Process. Lett., 30 (1989), pp. 9–14.

[12] M. I. SHAMOS, *Geometry and statistics: Problems at the interface*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 251–280.

[13] H. THEIL, *A rank-invariant method of linear and polynomial regression analysis* I, Proc. Kon. Ned. Aka. v. Wetensch. A., 53 (1950), pp. 386–392.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 811-830, August 1989

013

FULLY DYNAMIC POINT LOCATION IN A MONOTONE SUBDIVISION*

FRANCO P. PREPARATA† AND ROBERTO TAMASSIA‡

Abstract. In this paper a dynamic technique for locating a point in a monotone planar subdivision, whose current number of vertices is n, is presented. The (complete set of) update operations are insertion of a point on an edge and of a chain of edges between two vertices, and their reverse operations. The data structure uses space $O(n)$. The query time is $O(\log^2 n)$, the time for insertion/deletion of a point is $O(\log n)$, and the time for insertion/deletion of a chain with k edges is $O(\log^2 n + k)$, all worst-case. The technique is conceptually a special case of the chain method of Lee and Preparata and uses the same query algorithm. The emergence of full dynamic capabilities is afforded by a subtle choice of the chain set (separators), which induces a total order on the set of regions of the planar subdivision.

Key words. point location, planar subdivision, monotone polygon, dynamic data structures, on-line algorithm, computational geometry, analysis of algorithms

AMS(MOS) subject classifications. 68U05, 68Q25

1. Introduction.

A fundamental and classical problem in computational geometry, *planar point location*, is an instrument in a wide variety of applications. In a geometric context, it is naturally viewed as the extension to two dimensions of its one-dimensional analogue: search in a total ordering. It is formulated as follows. Given a planar subdivision **P** with n vertices (a planar graph embedded in the plane with straight-line edges), determine to which region of **P** a query point q belongs. The repetitive use of **P** and the on-line requirement on the answers call for a preprocessing of **P** that may ease the query operation, just as sorting and binary search intervene in one-dimensional search. The history of planar point location research spans more than a decade and is dense in results; the reader is referred to the extensive literature on this subject: [DL], [EKA], [EGS], [Ki], [LP], [LT], [P1], [P2], [PS], and [ST].

Most of the past research on the topic has focused on the *static* case of planar point location, where the planar subdivision **P** is fixed. For this instance of the problem, several practical techniques are available today (e.g., [EKA], [EGS], [LP], [P1], [ST]), some of which are provably optimal in the asymptotic sense [EGS], [ST]. The analogy with one-dimensional search, for which both static and dynamic optimal techniques have long been known, naturally motivates the desire to develop techniques for *dynamic* planar point location, where the planar subdivision can be modified by insertions and deletions of points and segments. In this setting, the three traditional measures of complexity for the static problem—preprocessing time, space for the data structure, and query time—are supplemented by measures of pertinent update times, and preprocessing time is no longer relevant if the data structure is dynamically built.

Work on dynamic point location is a rather recent undertaking. Overmars [O] proposed a technique for the case where the n vertices of **P** are given, the boundary of each region has a bounded number of edges, and only edges can be easily inserted or deleted. The basic entities used in Overmars's method are the edges themselves;

* Received by the editors January 20, 1988; accepted for publication (in revised form) November 23, 1988. This work was supported in part by National Science Foundation grant ECS-84-10902. A preliminary version of this paper was published in the Proceedings of the 29th IEEE Symposium on Foundations of Computer Science, pp. 558-567, 1988.

† Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801.

‡ Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. Present address, Department of Computer Science, Brown University, Providence, Rhode Island 02912.

each edge currently in **P** is stored in a segment tree defined on the fixed set of vertex abscissae, and the edge fragments assigned to a given node of the segment tree form a totally ordered set and are therefore efficiently searchable. This approach yields $O(n \log n)$ space, and $O(\log^2 n)$ query and update times (worst-case). Note that these measures are unrelated to the current number of edges in **P**. This technique can be extended to support insertions and deletions of vertices in $O(\log^2 n)$ amortized time, at the expense of deploying a rather complicated and not very practical data structure.

Another interesting dynamic point location technique, allowing both edge and vertex updates, is presented by Fries, Mehlhorn, and Naeher [F], [FMN], [M]. Their approach achieves $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log^4 n)$ amortized update time. If only insertions are considered, the update time is reduced to $O(\log^2 n)$ (amortized) [M, pp. 135–143]. This technique is suited for general subdivisions, and is based on the static point location algorithm of Lee and Preparata [LP].

In a recent paper [ST], Sarnak and Tarjan indicated as one of the most challenging problems in computational geometry the development of a fully dynamic point location data structure whose space and query time performance are of the same order as that of the best known static techniques for this problem. In this paper we make significant progress toward this goal, as expressed by the following theorem, which represents the central result of this paper.

THEOREM A. *Let* **P** *be a monotone planar subdivision with n vertices. There exists a dynamic point location data structure with* $O(n)$ *space requirement and* $O(\log^2 n)$ *query time that allows for insertion/deletion of a vertex in time* $O(\log n)$ *and insertion/deletion of a chain of k edges in time* $O(\log^2 n + k)$, *all time bounds being worst-case.*

It must be underscored that our method allows for arbitrary insertions and deletions of vertices and edges, the only condition being that monotonicity of the subdivision be preserved. The restriction to monotone subdivisions prevents the achievement of a more general goal. However, the class of monotone subdivisions is very important in applications, since it includes convex subdivisions, triangulations, and rectangular dissections. Moreover, from a methodological standpoint, the presented dynamic technique does not use bizarre data structures and is based on the same geometric objects, the separating chains, that yielded the first practical, albeit suboptimal, point location technique of Lee and Preparata [LP], and later the practical and optimal algorithm of Edelsbrunner, Guibas, and Stolfi [EGS].

The paper is organized as follows. In § 2 we review the technical background and precisely formulate the problem. We recall the static point location technique of Lee and Preparata that consists essentially of performing binary search on a set of monotone chains, called separators, sorted from left to right. Also, we introduce a complete repertory of update operations.

In § 3 we define a total ordering $<$ on the set of regions of **P** and define a subclass of monotone subdivisions, called regular subdivisions, for which the ordering $<$ is induced by the left-to-right adjacency of the regions. We show that the separators of a regular subdivision have a simple structure that allows for an efficient dynamic maintenance. We then transform an arbitrary subdivision **P** into a regular subdivision **P*** by (virtually) duplicating the edges along some monotone chains, called "channels." The regions of **P***, called "clusters," are sets of regions of **P** (consecutive in the order $<$) connected by channels. The formal underpinning of the order $<$ can be found in the theory of planar *st*-graphs [LEC], [TP] and planar lattices [KR].

In § 4, we present the dynamic technique. The data structure is based on organizing the separators of **P*** into a balanced tree, called "separator tree." We show that the insertion/deletion of a chain causes the ordering $<$ of the regions of **P** to be modified

in a simple way. Namely, the transformation consists of partitioning the sorted sequence of regions into four subsequences and swapping the two middle ones. Regarding $\mathbf{P}^*$, we show that only $O(1)$ channels are affected by the update. These properties are the basis of the algorithms for the insertion/deletion of chains. At the end of the section we describe the simpler algorithms for the insertion/deletion of vertices.

Finally, § 5 concludes the paper with some open problems.

2. Preliminaries. The geometric objects to be defined below are readily motivated if we view a point of the plane as the central projection of a point of a hemisphere to whose pole this plane is tangent.

A *vertex* v in the plane is either a finite point or a point at infinity (the latter is the projection of a point on the hemisphere equator). An *edge* $e=(u, v)$ is the portion of the straight line between u and v, with the only restriction that u and v be not points at infinity associated with the same direction. Thus e is either a segment or a straight-line ray, but not a whole straight-line. When both u and v are at infinity, then e is an edge at infinity, i.e., a portion of the line at infinity (the projection of an arc of the equator). A (*polygonal*) *chain* γ is a sequence $(e_i: e_i=(v_i, v_{i+1}), i=1, \cdots, p-1)$ of edges; it is *simple* if nonself-intersecting; it is *monotone* if any line parallel to the x-axis intersects γ in at most a point or a segment. The notions of "left" and "right" are referred to a bottom-up orientation of the involved entity (a chain, or, later on, a separator, an edge, etc.). A *simple polygon* r is a region of the plane delimited by a simple chain with $v_p=v_1$, called the *boundary* of r. Note that r could be unbounded; in this case the boundary of r contains one or more edges belonging to the line at infinity. A polygon r is monotone if its boundary is partitionable into two monotone chains γ_1 and γ_2, called the *left chain* and *right chain* of r, respectively (see Fig. 1). Chains γ_1 and γ_2 share two vertices referred to as $HIGH(r)$ and $LOW(r)$, respectively, with $y(HIGH(r))>y(LOW(r))$. In other words, $HIGH(r)$ and $LOW(r)$ are, respectively, points of maximum and minimum ordinates in polygon r; each is unique unless it is the extreme of either a horizontal edge or an edge at infinity, in which case the selection between the two edge extremes is arbitrary.

A *monotone subdivision* $\mathbf{P}$ is a partition of the plane into monotone polygons called the *regions* of $\mathbf{P}$ (see Fig. 2(a)). It is easily realized (see also [EGS]) that a monotone subdivision of $\mathbf{P}$ is determined by a planar graph G embedded in the plane whose edges are either segments or rays of straight lines (referred to as a "planar

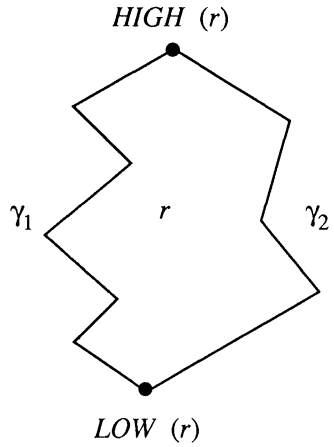


FIG. 1. *Nomenclature for a monotone polygon. Left chain*: γ_1; *right chain*: γ_2.

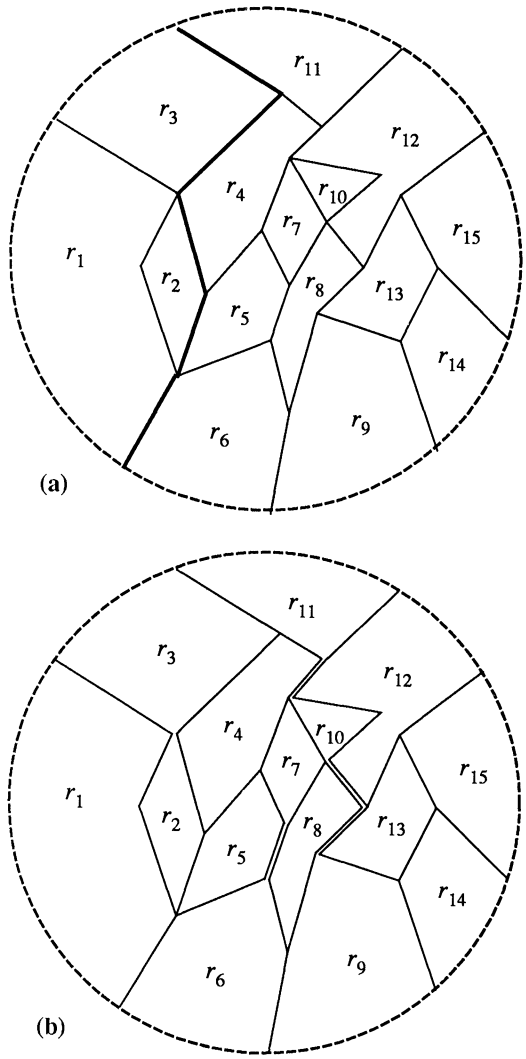


FIG. 2. (a) *A monotone subdivision* **P**. - - - *line at infinity*; ——— *separator.* (b) *The regular subdivision* **P*** *obtained from* **P** *by forming all maximal clusters.*

straight-line graph" in [LP]): edges, vertices, and chains of G are referred to as edges, vertices, and chains of **P**. The vertices of **P** are both the finite ones and those at infinity. This ensures the validity of the well-known Euler's formula and its corollaries:

$$|E| = O(|V|), \quad |R| = O(|V|),$$

where V, E, and R, respectively, are the set of vertices, edges, and regions of **P**.

Given a monotone subdivision **P**, a *separator* σ of **P** is a monotone chain $(v_1, \cdots, v_p)$ of **P** with the property that v_1 and v_p are points at infinity (hence, each horizontal line intersects a separator either in a point or in a segment). A separator of **P** is illustrated with bold line segments in Fig. 2(a). Given separators σ_1 and σ_2 of **P**, we say that σ_1 *is to the left of* σ_2, denoted $\sigma_1 < \sigma_2$, if, for any horizontal line l intersecting σ_1 and σ_2 in distinct points, l intersects σ_1 to the left of σ_2. A *partial subdivision* is the portion of a monotone subdivision contained between two distinct separators σ_1

and σ_2, with $\sigma_1 < \sigma_2$. A *complete family of separators* Σ for **P** is a sequence $(\sigma_1, \sigma_2, \cdots, \sigma_t)$ with $\sigma_1 < \sigma_2 < \cdots < \sigma_t$, such that every edge of **P** is contained in at least one separator of Σ. Notice that from Euler's formula $t = O(n)$. As shown in [LP], every monotone subdivision admits a complete family of separators.

Given a complete family of separators Σ for **P**, it is well known [LP] how to use Σ to perform planar point location in **P**. If n is the number of vertices of **P**, then in time $O(\log n)$ we can decide on which side of a separator the query point q lies; applying this operation as a primitive, a bisection search on Σ determines, in time $O(\log t \cdot \log n)$, two consecutive separators between which lies q. This process can be adapted or supplemented to determine the actual region r to which q belongs.

Since Σ is used in a binary search fashion, each separator is assigned to a node of a binary search tree, called the *separator tree* **T**. With a minor abuse of language, we call "node σ" the node of **T** to which σ has been assigned. An edge e of **P** belongs, in general, to a nonempty interval $(\sigma_i, \sigma_{i+1}, \cdots, \sigma_j)$ of separators. Let node σ_k, $i \leqq k \leqq j$ be the least common ancestor of nodes $\sigma_i, \sigma_{i+1}, \cdots, \sigma_j$; then e is called a *proper edge* of σ_k and is stored only once at node σ_k. We denote by $proper(\sigma_k)$ the set of proper edges of σ_k, i.e., the edges of σ_k stored at node σ_k. This yields $O(n)$ storage space while guaranteeing the correctness of the technique (see [EGS], [LP]). Note that edges whose extremes are both at infinity need not be stored.

We now illustrate that a planar subdivision **P** can be constructed by an appropriate sequence of the following operations.

INSERTPOINT $(v, e; e_1, e_2)$:
Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ by inserting vertex v.

REMOVEPOINT $(v; e)$:
Let v be a vertex of degree 2 whose incident edges, $e_1 = (u, v)$ and $e_2 = (v, w)$, are on the same straight line. Remove v and replace e_1 and e_2 with edge $e = (u, w)$.

INSERTCHAIN $(\gamma, v_1, v_2, r; r_1, r_2)$:
Add the monotone chain $\gamma = (v_1, w_1, \cdots, w_{k-1}, v_2)$, with $y(v_1) \leqq y(v_2)$, inside region r, which is decomposed into regions r_1 and r_2, with r_1 and r_2, respectively, to the left and to the right of γ, directed from v_1 to v_2.

REMOVECHAIN $(\gamma; r)$:
Let γ be a monotone chain whose nonextreme vertices have degree 2. Remove γ and merge the regions r_1 and r_2 formerly on the two sides of γ into region r. (The operation is allowed only if the subdivision **P**′ so obtained is monotone.)

With the above repertory of operations, we claim that a monotone subdivision **P** can always be transformed into a monotone subdivision **P**′ having either fewer vertices or fewer edges. Then by $O(n)$ such transformations, we obtain the trivial subdivision whose only region is the entire plane (bounded by the line at infinity).

Indeed, let σ be a separator of **P**, and imagine traversing σ from $-\infty$ to $+\infty$. Let $\deg^-(v)$ and $\deg^+(v)$ denote the numbers of edges incident on v and lying in the halfplanes $y < y(v)$ and $y > y(v)$, respectively. (For simplicity, we assume here that no edge is horizontal. In the general case, a horizontal edge is conventionally directed from left to right.) Let $(v_1, \cdots, v_p)$ be the sequence of finite vertices of σ with $\deg^+(v_i) + \deg^-(v_i) \geqq 3$, as encountered in the traversal of σ. If this sequence is empty, the entire chain can be trivially removed. Therefore, assume $p \geqq 1$. If there are two consecutive vertices v_i and v_{i+1} such that $\deg^+(v_i) \geqq 2$ and $\deg^-(v_{i+1}) \geqq 2$, then the chain γ (of degree-2 vertices) between v_i and v_{i+1} can be deleted by *REMOVECHAIN* while preserving monotonicity. Suppose that there are no two such vertices. If

$\deg^-(v_1) \geqq 2$, then the portion of σ from $-\infty$ to v_1 can be deleted. Otherwise, $\deg^-(v_1) = 1$, and the preceding conditions give rise to the following chain of implications:

$$(\deg^-(v_1) = 1) \Rightarrow (\deg^+(v_1) \geqq 2) \Rightarrow (\deg^+(v_2) \geqq 2) \Rightarrow \cdots \Rightarrow (\deg^+(v_p) \geqq 2);$$

the latter shows that the portion of σ from v_p to $+\infty$ can be deleted. This establishes our claim.

When all finite vertices have disappeared, the resulting subdivision consists of a closed chain of edges whose union is the line at infinity. Removal of the vertices at infinity completes the transformation. Since all of the above operations are reversible, this shows that any monotone subdivision **P** with n vertices can be constructed by $O(n)$ operations of the above repertory.

THEOREM 1. *An arbitrary planar subdivision* **P** *with n vertices can be assembled starting from the empty subdivision by a sequence of* $O(n)$ *INSERTPOINT and INSERTCHAIN operations, and can be disassembled to obtain the empty subdivision by a sequence of* $O(n)$ *REMOVEPOINT and REMOVECHAIN operations.*

Although the above operations are sufficient to assemble and disassemble any monotone subdivision, the following operation is also profitably included in the repertory.

MOVEPOINT $(v; x, y)$:

Translate a degree-2 vertex v from its present location to point (x, y). (The operation is allowed only if the subdivision $\mathbf{P}'$ so obtained is monotone and topologically equivalent to **P**.)

3. Ordering the regions of a monotone subdivision. Let **P** be a monotone subdivision, and assume for simplicity that none of its finite edges is horizontal. Given two regions r_1 and r_2 of **P**, we say that r_1 is *left-adjacent to* r_2, denoted $r_1 \ll r_2$, if r_1 and r_2 share an edge e, and any separator of **P** containing e leaves r_1 to its left and r_2 to its right. Note that relation $\ll$ is trivially antisymmetric. But we can also show Lemma 1.

LEMMA 1. *Relation* $\ll$ *on the regions of* **P** *is acyclic.*

Proof. Assume $r_1 \ll r_2 \ll \cdots \ll r_k \ll r_1$. Let Σ be a complete family of separators and let $\{\sigma_1, \cdots, \sigma_{k-1}\} \subseteq \Sigma$ be such that σ_i separates r_i and r_{i+1}, so that $\sigma_1 < \sigma_2 < \cdots < \sigma_{k-1}$. If $\sigma \in \Sigma$ leaves r_k to its left and r_1 to its right, we have $\sigma_{k-1} < \sigma$ and $\sigma < \sigma_1$, a contradiction since the separators are ordered. □

Thus, the transitive closure of $\ll$ is a partial order, referred to as "*to the left of*," and denoted by $\rightarrow$. Specifically, $r_1 \rightarrow r_2$ if there is a path from r_1 to r_2 in the directed graph of the relation $\ll$ on the set of regions. Correspondingly, given two regions r_1 and r_2 of **P**, we say that r_1 *is below* r_2, denoted $r_1 \uparrow r_2$, if there is a monotone chain from $HIGH(r_1)$ to $LOW(r_2)$. Obviously, $\uparrow$ is a partial order on the set of regions. The following lemma shows that these two partial orders are complementary, and it is the geometric counterpart of a topological property of planar lattices [B, ex. 7(c), p. 32], [Ka], [KR].

LEMMA 2. *Let* r_1 *and* r_2 *be two regions of a monotone subdivision* **P**. *Then one and only one of the following holds*:

$$r_1 \rightarrow r_2, \quad r_2 \rightarrow r_1, \quad r_1 \uparrow r_2, \quad r_2 \uparrow r_1.$$

Proof. Let σ_L be the leftmost separator that contains the left chain of the boundary of r_1 and, analogously, let σ_R be the rightmost separator containing the right chain of the boundary of r_1. These separators partition **P** into five portions, each a partial subdivision: one of them is r_1 itself, and the others are denoted L, R, B, and T (see Fig. 3). Now, we consider four mutually exclusive cases for r_2, one of which must occur:

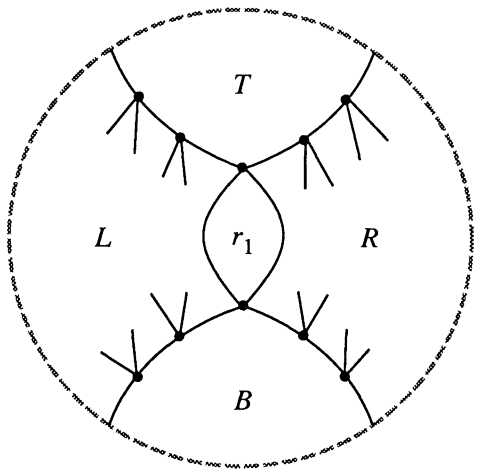


FIG. 3. **P** *partitioned into five portions, each a partial subdivision (for the proof of Lemma* 2).

(1) $r_2 \in L$. Consider any sequence of regions $(r'_1, \cdots, r'_s)$ such that $r_2 \to r'_1$, $r'_i \to r'_{i+1}$ for $i = 1, \cdots, s-1$, and the right chain of r'_s has a nonempty intersection with σ_L. If the right chain of r'_s has an edge that is also on the left boundary of r_1, then $r_2 \to r_1$. Otherwise, by the definition of σ_L, there is a sequence $r'_{s+1}, \cdots, r'_p$ of regions such that $r'_j \to r'_{j+1}$ for $j = s, \cdots, p-1$, and $r'_p \to r_1$. Thus, in all cases, $r_2 \to r_1$.

(2) $r_2 \in R$. Arguing as in (1), we establish $r_1 \to r_2$.

(3) $r_2 \in B$. Since $LOW(r_1)$ is the highest ordinate vertex in B, there is a monotone chain from any vertex in B to $LOW(r_1)$, and, in particular, from $HIGH(r_2)$ to $LOW(r_1)$. Thus $r_2 \uparrow r_1$.

(4) $r_2 \in T$. Arguing as in (3), we establish $r_1 \uparrow r_2$. □

We say that r_1 *precedes* r_2, denoted $r_1 < r_2$, if either $r_1 \to r_2$ or $r_1 \uparrow r_2$.

THEOREM 2. *The relation* $<$ *on the regions of* **P** *is a total order.*

As an example, the region subscripts in Fig. 2(a) reflect the order $<$.

DEFINITION 1. A *regular subdivision* is a monotone subdivision having no pair (r_1, r_2) of regions such that $r_1 \uparrow r_2$.

For example, in Fig. 2(a) we have $r_9 \uparrow r_{10}$, which shows that the illustrated monotone subdivision is not regular. An example of regular subdivision is given in Fig. 4(a).

The significance of regular subdivisions is expressed by Theorem 3 below. It is easily realized that there is a unique complete family $\Sigma = (\sigma_1, \cdots, \sigma_t)$ of separators for a regular subdivision **P**. By the definition of separator, all regions to the left of σ_j precede all those to its right in the order $<$. Let **T** be a separator tree for the above family Σ. Recalling the rule for storing the edges of separator σ in **T**, as reviewed in § 2, we have Theorem 3.

THEOREM 3. *In a regular subdivision* **P**, *the edges of proper*(σ) *in* **T** *form a single chain* (*see Fig.* 4(b)).

Proof. Assume for a contradiction that σ contains a chain γ that is the bottom-to-top concatenation of three nonempty chains γ_1, γ_2, and γ_3, where γ_1 and γ_3 consist of proper edges of σ, and γ_2 contains no such edge. Let v_1 and v_2 be the bottom and top vertices, respectively, of γ_2, and let $e' \in \gamma_1$ and $e'' \in \gamma_2$ be the edges of σ incident on v_1. Since $e'' \notin proper(\sigma)$, we must have $e'' \in proper(\sigma')$, where node σ' is an ancestor of node σ in **T**. We claim there is a region r_1 such that $v_1 = HIGH(r_1)$. Otherwise, each separator containing e''—and so σ'—also contains e', contrary to the hypothesis

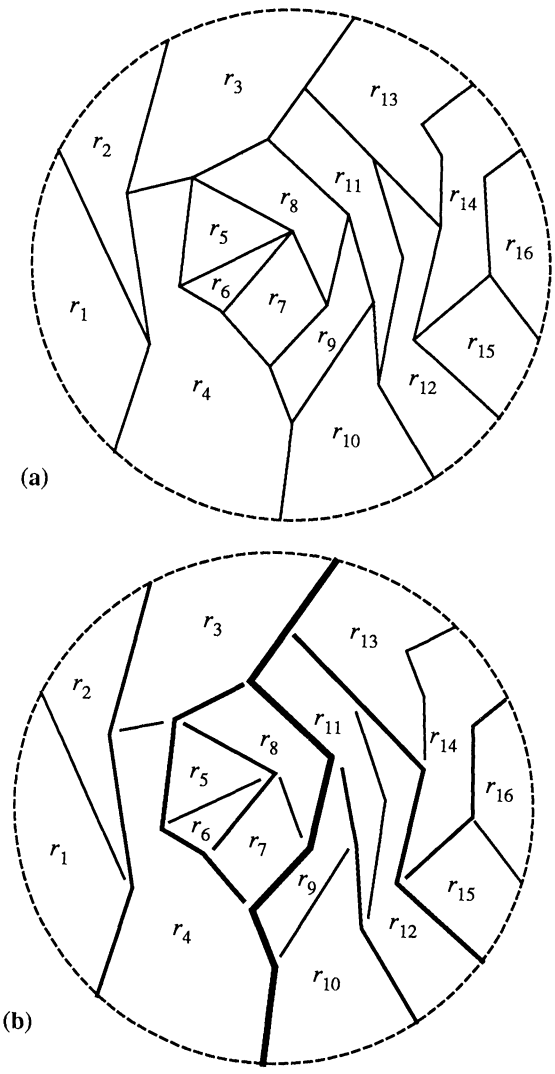


FIG. 4. (a) *A regular subdivision.* (b) *Its chains* $\{proper(\sigma)\colon \sigma \in \Sigma\}$.

that node σ is closest to the root among the nodes whose separator contains e'. Analogously, we show that there is a region r_2 for which $v_2 = LOW(r_2)$. Since γ_2 is a monotone chain from $HIGH(r_1)$ to $LOW(r_2)$, then $r_1 \uparrow r_2$, whence a contradiction. □

This theorem shows that a regular subdivision has a particularly simple separator tree. In the next section we shall show that the property expressed by Theorem 3 is crucial for the efficient dynamization of the chain method for point location. We now slightly generalize the notion of region in a way that will enable us to show that any monotone subdivision embeds a unique regular subdivision. We say that two regions r_1 and r_2 consecutive in $<$ with $r_1 \uparrow r_2$ are *vertically consecutive.* We then have Lemma 3.

LEMMA 3. *If r_1 and r_2 are two vertically consecutive regions of a monotone subdivision* **P**, *then the monotone chain from $HIGH(r_1)$ to $LOW(r_2)$ is unique.*

Proof. The lemma holds trivially if $HIGH(r_1) = LOW(r_2)$. Thus, assume the contrary. Since $r_1 \uparrow r_2$, there is a monotone chain γ from $HIGH(r_1)$ to $LOW\ (r_2)$.

Suppose now, for a contradiction, that there is a monotone chain γ' from $HIGH\ (r_1)$ to $LOW\ (r_2)$ distinct from γ. Then $\gamma \cup \gamma'$ defines the boundary of a partial subdivision that contains at least one region of **P**. For any region r inside this partial subdivision, there are (possibly empty) chains from $HIGH(r_1)$ to $LOW(r)$ and from $HIGH(r)$ to $LOW(r_2)$, so that $r_1 \uparrow r \uparrow r_2$, contrary to the hypothesis that r_1 and r_2 are consecutive in $<$. □

DEFINITION 2. Given two vertically consecutive regions, r_1 and r_2, in **P**, with $r_1 \uparrow r_2$, the unique monotone chain from $HIGH(r_1)$ to $LOW(r_2)$ is called a *channel.*

LEMMA 4. *All channels are pairwise vertex-disjoint.*

Proof. Assume, for a contradiction, that there are two channels γ_1 and γ_2 that are not vertex-disjoint, where γ_1 connects regions r_1 and r_2, γ_2 connects regions r_3 and r_4, and $r_1 < r_2 < r_3 < r_4$ (see Fig. 5). Since γ_1 and γ_2 share a vertex x, there is a chain from $HIGH(r_3)$ to $LOW(r_2)$ consisting of the portion of γ_2 from $HIGH(r_3)$ to x, and the portion of γ_1 from x to $LOW(r_2)$. Hence, we have $r_3 < r_2$, a contradiction. □

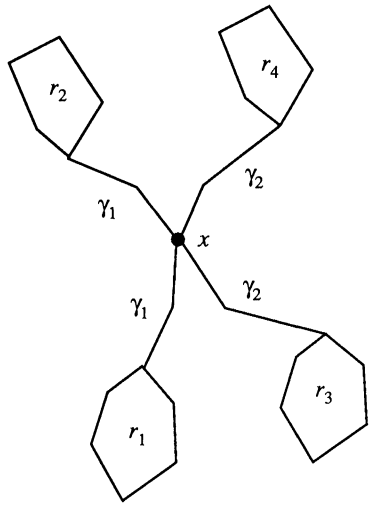


FIG. 5. *Two channels γ_1 and γ_2 that are not vertex-disjoint, where γ_1 connects regions r_1 and r_2, γ_2 connects regions r_3 and r_4, and $r_1 < r_2 < r_3 < r_4$ (for the proof of Lemma 4).*

Given two vertically consecutive regions r_1 and r_2, with $r_1 \uparrow r_2$, we imagine duplicating the channel from r_1 to r_2 and view the measure-zero region delimited by the two replicas as a degenerate polygon joining r_1 and r_2 and merging them into a new region $r_1 \cup r_2$ (see Fig. 6). Clearly, we can merge in this fashion any sequence of vertically consecutive pairs. This is formulated in the following definition.

DEFINITION 3. *Clusters* are recursively defined as follows:

(1) An individual region r is a cluster;

(2) Given two vertically consecutive clusters χ_1 and χ_2, with $\chi_1 \uparrow \chi_2$, their union is a cluster χ, denoted χ_1-χ_2 (the horizontal bar denotes the channel).

A *maximal cluster* χ is one that is not properly contained in any other cluster.

The unique subdivision resulting by forming all maximal clusters of **P** is denoted **P***. Figure 2(b) illustrates the regular subdivision **P*** corresponding to the subdivision **P** of Fig. 2(a). Note the clusters r_2-r_3, r_6-r_7, and r_9-r_{10}-r_{11}.

The above definition leads us to a convenient string notation for the order $<$, as well as for the cluster structure where appropriate. Normally, we shall use lowercase

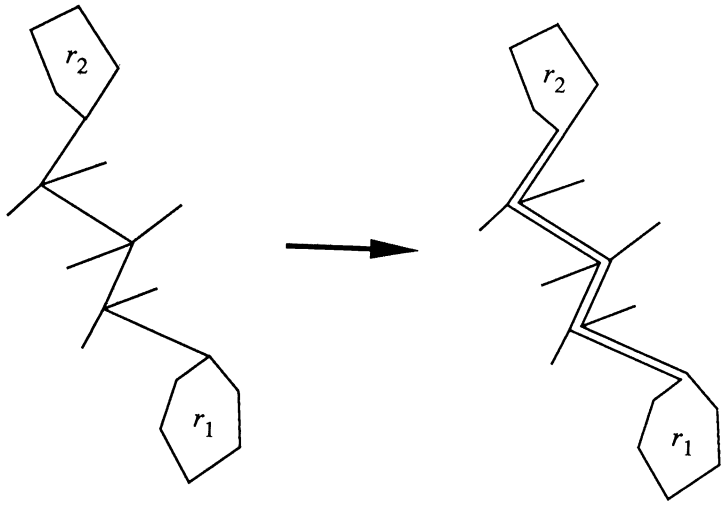


FIG. 6. *Creation of a channel between two vertically consecutive regions.*

roman letters for individual regions, lowercase Greek letters for clusters, and uppercase roman letters otherwise (i.e., for collections of consecutive regions not forming a single cluster). Specifically, we have:

(1) A cluster (possibly, a region) is a string.

(2) Given two strings A and B, such that the rightmost cluster of A and the leftmost cluster of B are consecutive (contiguity), then AB is a string.

A subdivision may be represented by means of its structural decomposition. For example, the subdivision of Fig. 2(b) is described by the following string:

$$r_1\chi_1 r_4 r_5 \chi_2 r_8 \chi_3 r_{12} r_{13} r_{14} r_{15},$$

where $\chi_1 = r_2 - r_3$, $\chi_2 = r_6 - r_7$, and $\chi_3 = r_9 - r_{10} - r_{11}$.

Later we will find it convenient to explicitly indicate that two consecutive regions r_1 and r_2 may or may not form a cluster. We shall denote this with the string notation $r_1 - - r_2$, where "--" means "potential channel."

We conclude this section with the following straightforward observation.

THEOREM 4. *The subdivision* $\mathbf{P}^*$ *obtained by forming all maximal clusters of a monotone subdivision* $\mathbf{P}$ *is regular.*

Note that in the transformation of $\mathbf{P}$ to $\mathbf{P}^*$ only the edges of channels are duplicated. By Lemma 4, each edge is duplicated at most once, thereby ensuring that the number of edges remains $O(n)$.

4. Dynamic point location in a monotone subdivision.

4.1. Data structure. In the following description, we assume that all sorted lists are stored as *red-black trees* [GS], [T]. We recall the following properties of red-black trees that are important in the subsequent time complexity analyses.

(1) Only $O(1)$ rotations are needed to rebalance the tree after an insertion/deletion.

(2) The data structure can be used to implement concatenable queues. Operations *SPLICE* and *SPLIT* of concatenable queues take $O(\log n)$ time and need $O(\log n)$ rotations each for rebalancing.

The search data structure consists of a main component, called the *augmented separator tree*, and an auxiliary component, called the *dictionary*. The *augmented*

separator tree **T** has a primary and secondary structure. The *primary structure* is a separator tree for $\mathbf{P}^*$, i.e., each of its leaves is associated with a region of $\mathbf{P}^*$ (a maximal cluster of **P**), and each of its internal nodes is associated with a separator of $\mathbf{P}^*$. (The left-to-right order of the leaves of the primary structure of **T** corresponds to the order $<$ on the regions of $\mathbf{P}^*$.) The *secondary structure* is a collection of lists, each realized as a search tree. Specifically, node σ points to the list $proper(\sigma)$ sorted from bottom to top, and the leaf associated with cluster χ (briefly called "leaf χ") points to the list $regions(\chi)$ of the regions that form cluster χ, also sorted from bottom to top.

Given two regions r_1 and r_2 consecutive in $<$, the separator σ between r_1 and r_2 is associated with the least common ancestor of the leaves associated with the respective clusters of r_1 and r_2 in **T**. By the definition of separator tree, the edges of σ are stored in the secondary structures of nodes along the path from node σ to the root of **T**; by Theorem 3, in a regular subdivision each extreme vertex of $proper(\sigma)$ splits $proper(\sigma')$, for some ancestor node σ' of node σ, into two chains. More precisely, the following simple lemma, stated without proof, makes explicit the allocation of the edges of σ to the nodes of **T**.

LEMMA 5. *Let σ be a separator of $\mathbf{P}^*$, and $\sigma_1, \cdots, \sigma_h$ be the sequence of nodes of* **T** *on the path from the root $(=\sigma_1)$ to σ. Then*

$$\sigma = (\alpha_1, \alpha_2, \cdots, \alpha_h, proper(\sigma), \beta_h, \beta_{h-1}, \cdots, \beta_1),$$

where α_i and β_i are (possibly empty) subchains of $proper(\sigma_i)$, $i = 1, \cdots, h$.

To dynamically maintain the channels, it is convenient to keep two representatives e' and e'' of each edge e that are created when e is inserted into **P**. If e does not belong to a channel, e' and e'' are joined into a *double edge* and belong to the same $proper(\sigma)$. If instead e is part of a channel, then e' and e'' are *single edges* and belong to distinct $proper(\sigma')$ and $proper(\sigma'')$. In the latter case e' and e'' are on the boundary of the same cluster χ so that nodes σ' and σ'' are on the path from leaf χ to the root of **T**. Therefore we represent $proper(\sigma)$ by means of two lists: $strand1(\sigma)$ and $strand2(\sigma)$. List $strand1(\sigma)$, called *primary strand*, stores a representative for each edge of $proper(\sigma)$ in bottom-to-top order. List $strand2(\sigma)$, called *secondary strand*, stores a representative for each double edge of $proper(\sigma)$ in bottom-to-top order.

Moreover, associated with each chain $proper(\sigma)$ there are two boolean indicators $t(\sigma)$ and $b(\sigma)$, corresponding, respectively, to the topmost and bottommost vertices of $proper(\sigma)$. Specifically, let σ' be the ancestor of σ such that the topmost vertex of $proper(\sigma)$ is an internal vertex of the chain $proper(\sigma')$ (for the special case where the topmost vertex of σ is at infinity, we let σ' be the father of σ). We define $t(\sigma) = left$ if σ is to the left of σ', and $t(\sigma) = right$ if σ is to the right of σ'. Parameter $b(\sigma)$ is analogously defined.

The *dictionary* contains the sorted lists of the vertices, edges, and regions of **P**, each sorted according to the alphabetic order of their names. With each vertex v we store pointers to the representatives of v in the (at most two) chains $proper(\sigma)$ and $proper(\sigma')$ of which v is a nonextreme vertex. With each edge e we store pointers to the two representatives of e in the data structure. Finally, with each region r we store the vertices $HIGH(r)$ and $LOW(r)$, and a pointer to the representative of r in the list $regions(\chi)$ such that χ is the maximal cluster containing r. The dynamic maintenance of the dictionary in the various operations can be trivially performed in $O(\log n)$ time, and will not be explicitly mentioned in the following.

To analyze the storage used by the data structure, we note the following: the primary structure of **T** has $O(n)$ nodes, since there are $O(n)$ regions (by Euler's formula) and therefore $O(n)$ separators; the secondary structure of **T** also has size

$O(n)$, since there are $O(n)$ edges in $\{proper(\sigma): \sigma \in \mathbf{T}\}$ and $O(n)$ regions in $\{regions(\chi): \chi \in \mathbf{T}\}$ (again, by Euler's formula); the auxiliary component has one record of bounded size per vertex, edge, and region. Therefore, we conclude with the following theorem.

THEOREM 5. *The data structure for dynamic point location has storage space* $O(n)$.

Note that the above data structure is essentially *identical* with the one originally proposed for the static version of the technique [LP]. What is remarkable is that the single-chain structure of the proper edges of any given separator, due to our specific choice of the separator family, is the key for the emergence of full dynamic capabilities.

We now show that the property expressed by Theorem 3 allows us to establish an important dynamic feature of the data structure. According to standard terminology, a *rotation at node* μ of a binary search tree is the restructuring of the subtree rooted at μ so that one of the children of μ becomes the root thereof. A rotation is either *left* or *right* depending on whether the right or left child of μ becomes the new root, respectively. We then have Lemma 6.

LEMMA 6. *A rotation at a node of* $\mathbf{T}$ *can be performed in* $O(\log n)$ *time.*

Proof. Without loss of generality, we consider a left rotation as illustrated in Fig. 7. Clearly, the separators stored at nodes outside the subtree rooted at σ_2 in Fig. 7(a) are not affected by the rotation, nor are those in the subtrees rooted at σ_1, σ_3, and σ_5. Hence, the only alterations involve separators σ_2 and σ_4. If $\sigma_4 \cap proper(\sigma_2) = \emptyset$, then the modification is trivial. Suppose then that $\sigma_4 \cap proper(\sigma_2) \neq \emptyset$. In this case the set $\sigma_4 \cap proper(\sigma_2)$ forms either the initial or the final segment of the chain $proper(\sigma_2)$, or both, for otherwise, regular subdivision $\mathbf{P}^*$ would contain vertically consecutive regions. Thus the update is accomplished by: (1) splitting $proper(\sigma_2)$ into $\gamma_2 = \sigma_4 \cap proper(\sigma_2)$ and its relative complement γ_1: (2) splicing γ_2 with $proper(\sigma_4)$ to form the updated $proper(\sigma_4)$; and (3) setting the updated $proper(\sigma_2)$ equal to γ_1. Note that the extreme vertices of $proper(\sigma_4)$ are obtained in time $O(1)$, and the splitting vertices of $proper(\sigma_2)$ are determined in time $O(\log n)$. Since data structures for $proper(\sigma_2)$ and $proper(\sigma_4)$, (i.e., the red-black trees associated with lists $strand1$ and $strand2$) are also concatenable queues, the splitting and splicing operations are executed in time $O(\log n)$ as well. The parameters $t(\sigma)$ and $b(\sigma)$ for the resulting separators are updated in $O(1)$ time by means of straightforward rules. □

Hereafter, the red-black tree $\mathbf{T}$ is assumed to be balanced. The rest of this section is devoted to the discussion of the algorithms to perform searches, insertions, and deletions.

4.2. Query. To perform point location search for a query point q, we use essentially the same method as in [LP]. The search consists of tracing a path from the root to a leaf χ of $\mathbf{T}$. At each internal node σ we discriminate q against separator σ. Three cases may occur:

(1) $q \in \sigma$: we return the edge of σ that contains q and stop;
(2) q is to the left of σ: we proceed to the left child of σ;
(3) q is to the right of σ: we proceed to the right child of σ.

Once we reach a leaf χ, we know that q belongs to a region of χ. Since the regions of χ are sorted from bottom to top, such region is determined by searching in the list $regions(\chi)$. The above technique can be viewed as a "horizontal" binary search in the set of separators of $\mathbf{P}^*$, followed by a "vertical" binary search in the set of regions of the leaf χ.

Let e be the edge of σ whose vertical span contains $y(q)$. When $e \in proper(\sigma)$, the discrimination of q against σ is a conventional search in $strand1(\sigma)$. When

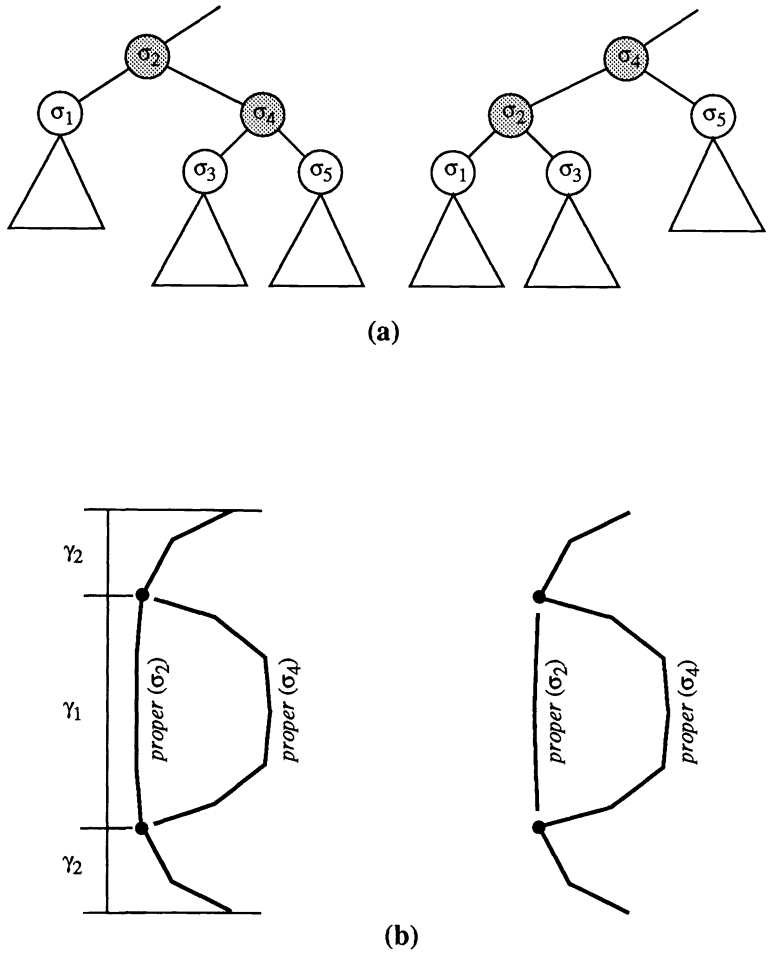


FIG. 7. *Illustration of a (left) rotation. Shaded regions are nodes involved in the update.* (a) *Separator tree.* (b) *Chains of proper edges.*

$e \notin proper(\sigma)$ then we use the pair $(t(\sigma), b(\sigma))$: for example, when e is above $proper(\sigma)$, if $t(\sigma) = left$, then q is discriminated to the right of σ, and to its left otherwise. (This is a minor variant of the criterion adopted in [LP].) The case when e is below $proper(\sigma)$ is treated analogously. This simple analysis confirms that the time spent at each node is $O(\log n)$. We have Theorem 6 [LP].

THEOREM 6. *The time complexity of the query operation is* $O(\log^2 n)$.

4.3. Insertion. We shall first show that the effect of operation *INSERT-CHAIN*$(\gamma, v_1, v_2, r; r_1, r_2)$ on the order $<$ of the regions of **P** can be expressed as a syntactical transformation between the strings expressing the order before and after the update. The situation is illustrated in Fig. 8.

On the boundary of r there are two distinguished vertices: $HIGH(r_1)$ and $LOW(r_2)$. Note that $HIGH(r_1) = HIGH(r)$ if v_2 is on the right chain of the boundary of r (and similarly $LOW(r_2) = LOW(r)$ if v_1 is on the left chain). Thus, in general, $HIGH(r_1)$ is on the left chain of the boundary of r, and $LOW(r_2)$ is on the right chain. Using the string notation introduced in § 3, let L and R be the strings corresponding to the

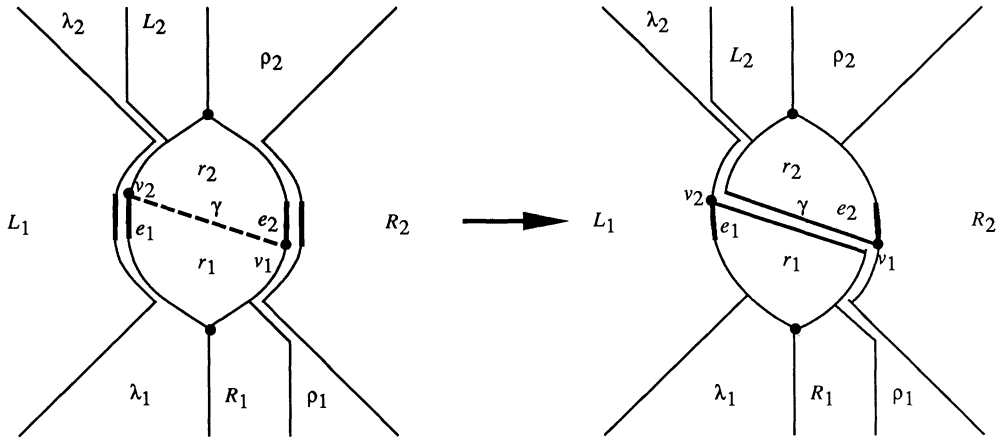


FIG. 8. (a) *Canonical partition of subdivision* $\mathbf{P}^*$ *with reference to region r and vertices* v_1 *and* v_2. (b) *The restructured subdivision after the insertion of chain* γ *between* v_1 *and* v_2.

regions that, respectively, precede and follow r in $<$. Thus, the subdivision $\mathbf{P}^*$ is described by the string LrR.

Let e_1 be the edge of $\mathbf{P}^*$ on the left boundary of r incident on $HIGH(r_1)$ from below, and let χ be the maximal cluster on the left of e_1. In general, this cluster consists of two portions, χ_1 and χ_2 (such that χ_1-χ_2), where χ_2 consists exactly of the regions q' of χ for which $y(LOW(q')) \cong y(HIGH(r_1))$. Thus, we have $L = L'\chi L''$. We now distinguish three cases and define substrings λ_1, λ_2, L_1, and L_2 as follows.

(1) $\chi_2 \neq \varnothing$. Let $\lambda_1 = \chi_1$, $\lambda_2 = \chi_2$, $L_1 = L'$, $L_2 = L''$, so that $L = L_1\lambda_1\text{-}\lambda_2 L_2$.

(2) $\chi_2 = \varnothing$. Let q be the region preceding r (note q could form a cluster with r). We further distinguish:

(2.1) $y(LOW(q)) \cong y(HIGH(r_1))$. In this case we let $L = L_1\lambda_2 L_2$, where λ_2 is the maximal cluster immediately following χ.

(2.2) $y(LOW(q)) < y(HIGH(r_1))$. In this case we let $L = L_1\lambda_1\text{--}$, where λ_1 is the rightmost maximal cluster of L (but not necessarily a maximal cluster in $\mathbf{P}^*$).

The three cases are conveniently encompassed by the notation

$$L = L_1\lambda_1\text{--}\lambda_2 L_2.$$

Note that some of the symbols may denote empty strings.

Analogously, string R can be reformulated as

$$R = R_1\rho_1\text{--}\rho_2 R_2$$

with straightforward meanings of the symbols.

Thus, in general, for any given region r and choice of v_1 and v_2 on its boundary, we have the following canonical string decomposition of $\mathbf{P}^*$:

$$L_1\lambda_1\text{--}\lambda_2 L_2 r R_1\rho_1\text{--}\rho_2 R_2.$$

The corresponding partition of the subdivision is illustrated in Fig. 8(a). Examples of configurations corresponding to cases (2.1) and (2.2) are shown in Fig. 9. Namely, Fig. 9(a) shows case (2.1) for L and case (1) for R, Fig. 9(b) shows case (2.2) for L and R, and Fig. 9(c) shows case (2.2) for L and case (1) for R.

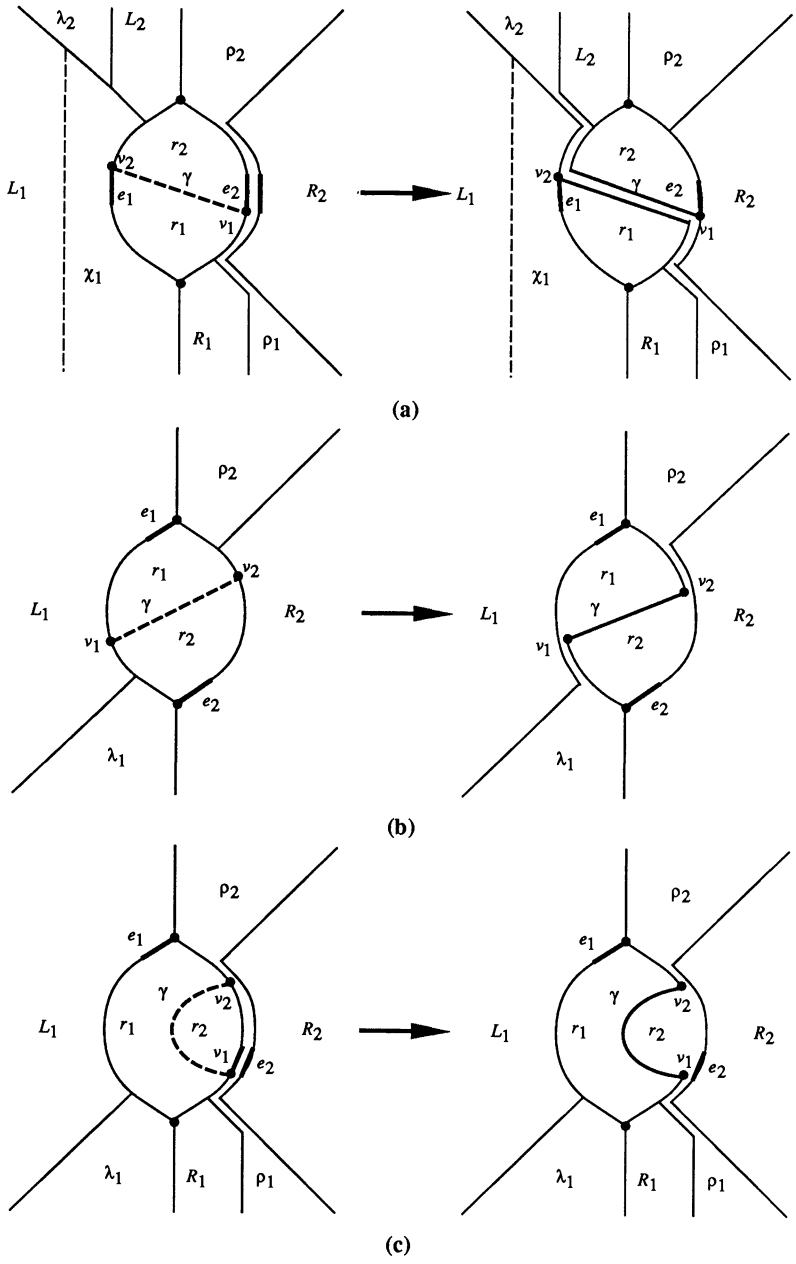


FIG. 9. *Special cases of the structural partition of Fig.* 8. (a) *Case* (2.1) *for L and case* (1) *for R.* (b) *Case* (2.2) *for both L and R.* (c) *Case* (2.2) *for L and case* (1) *for R.*

We now investigate the rearrangement of this order caused by the insertion of chain γ into r. Referring to Fig. 8(b), it is immediately observed that the order after the update is as follows:

$$L_1 < \lambda_1 < r_1 < R_1 < \rho_1 < \lambda_2 < L_2 < r_2 < \rho_2 < R_2.$$

To obtain the string description of the updated subdivision we must determine whether any new channel has been created. Any such channel can only arise in correspondence

with a new adjacency caused by the update, specifically for the following pairs: (λ_1, r_1), (ρ_1, λ_2), and (r_2, ρ_2). The channel from λ_1 to r_1 exists only if $y(HIGH(\lambda_1)) \leqq y(LOW(r_1))$, and analogously for the channel from r_2 to ρ_2. Instead, since $y(HIGH(\rho_1)) < y(LOW(\lambda_2))$, the cluster ρ_1-λ_2 always exists. Therefore, the order caused by the insertion of γ is represented by the string

$$L_1\lambda_1\text{--}r_1R_1\rho_1\text{-}\lambda_2L_2r_2\text{--}\rho_2R_2.$$

(In purely syntactic terms, this transformation corresponds to rewriting r as $r_2 - r_1$ and then exchanging substrings $r_1R_1\rho_1$ and $\lambda_2L_2r_2$.) This is summarized as follows.

THEOREM 7. *Let* $L_1\lambda_1\text{--}\lambda_2L_2rR_1\rho_1\text{--}\rho_2R_2$ *be the string description of the order of* $\mathbf{P}^*$, *where* L_1, r, *and* R_2 *are nonempty. After operation* $INSERTCHAIN(\gamma, v_1, v_2, r; r_1, r_2)$ *the new order is described by* $L_1\lambda_1\text{--}r_1R_1\rho_1\text{-}\lambda_2L_2r_2\text{--}\rho_2R_2$.

The algorithm for the $INSERTCHAIN(\gamma, v_1, v_2, r; r_1, r_2)$ operation implements the syntactical transformation of the string description by decomposing the subdivision $\mathbf{P}$ into its components L_1, λ_1, λ_2, L_2, r, R_1, ρ_1, ρ_2, and R_2, which are subsequently reassembled according to the new order given by Theorem 7.

To formally describe the algorithm, we denote by $\mathbf{P}(S)$ the partial subdivision associated with a string S of consecutive regions of $\mathbf{P}$. We can represent $\mathbf{P}(S)$ with essentially the same data structure described in § 4.1, and we denote with $\mathbf{T}(S)$ the augmented separator tree for S. Note that $\mathbf{T}(S)$ does not store the edges which form the boundary of S (in the same way as $\mathbf{T}$ does not store the edges at infinity). Partial subdivisions can be cut and merged with the same rules as for the decomposition and concatenation of the corresponding strings.

Let $\mathbf{P}(S_1)$, $\mathbf{P}(S_2)$, and $\mathbf{P}(S)$ be partial subdivisions such that $S = S_1S_2$. We show in the following how to *merge* $\mathbf{T}(S_1)$ and $\mathbf{T}(S_2)$ into $\mathbf{T}(S)$, and how to *cut* $\mathbf{T}(S)$ to produce $\mathbf{T}(S_1)$ and $\mathbf{T}(S_2)$. The merge operation needs also the separator σ forming the common boundary between the two (open) partial subdivisions $\mathbf{P}(S_1)$ and $\mathbf{P}(S_2)$; σ is represented by its primary and secondary strands. The cut operation returns the separator σ. These operations can be implemented by means of the following six primitives.

PROCEDURE $MERGE1(S_1, \sigma, S_2; S)$. (It merges partial subdivisions $\mathbf{P}(S_1)$ and $\mathbf{P}(S_2)$, with $S_1 \to S_2$; σ is the separator between $\mathbf{P}(S_1)$ and $\mathbf{P}(S_2)$.)

(1) Construct a separator tree $\mathbf{T}(S)$ for $\mathbf{P}(S)$, by placing σ at the root, and making $\mathbf{T}(S_1)$ and $\mathbf{T}(S_2)$ the left and right subtrees of σ, respectively.
($\mathbf{T}(S)$ is a legal separator tree for $\mathbf{P}(S)$, but might be unbalanced.)

(2) Rebalance $\mathbf{T}(S)$ by means of rotations.

PROCEDURE $MERGE2(\chi_1, \alpha, \chi_2; \chi)$. (It merges partial subdivisions $\mathbf{P}(\chi_1)$ and $\mathbf{P}(\chi_2)$ such that $\chi_1 \uparrow \chi_2$ into $\mathbf{P}(\chi)$, where $\chi = \chi_1\text{-}\chi_2$, and α is the channel between γ_1 and χ_2.)

(1) Separate the two strands of α, and make the secondary strand become a new primary strand.

(2) Splice $regions(\chi_1)$ and $regions(\chi_2)$ to form $regions(\chi)$.

LEMMA 7. *Operations* $MERGE1(S_1, \sigma, S_2; S)$ *and* $MERGE2(\chi_1, \alpha, \chi_2; \chi)$ *have time complexity* $O(\log^2 n)$ *and* $O(\log n)$, *respectively.*

Proof. The time bound for operation $MERGE2$ follows immediately from the properties of concatenable queues. With regard to $MERGE1$, Step (1) consists of forming $\mathbf{T}(S)$ by joining the primary structures of $\mathbf{T}(S_1)$ and $\mathbf{T}(S_2)$ through node σ, which takes $O(1)$ time. Since we use red-black trees, we can rebalance $\mathbf{T}(S)$ with

$O(\log n)$ rotations [GS], [T, pp. 52–53]. By Lemma 6, each such rotation takes $O(\log n)$ time, so that the total time complexity is $O(\log^2 n)$. □

PROCEDURE $CUT1(S, \chi_1, \chi_2; S_1, \sigma, S_2)$. (It cuts partial subdivision $\mathbf{P}(S)$ into $\mathbf{P}(S_1)$ with rightmost cluster χ_1 and $\mathbf{P}(S_2)$ with leftmost cluster χ_2, such that $\chi_1 \to \chi_2$, and also returns the separator σ between $\mathbf{P}(S_1)$ and $\mathbf{P}(S_2)$.)

(1) Find the node σ of $\mathbf{T}(S)$ that is the least common ancestor of leaves χ_1 and χ_2.
(2) Perform a sequence of rotations to bring σ to the root of $\mathbf{T}(S)$, where after each rotation we rebalance the subtree of σ involved in the rotation, namely, the left subtree for a left rotation and the right subtree for a right rotation (see Fig. 10).
(3) Set $\mathbf{T}(S_1)$ as the left subtree of σ and $\mathbf{T}(S_2)$ as the right subtree of σ. Return the chain $proper(\sigma)$.

PROCEDURE $CUT2(\chi; \chi_1, \alpha, \chi_2)$. (It cuts partial subdivision $\mathbf{P}(\chi)$ into $\mathbf{P}(\chi_1)$ and $\mathbf{P}(\chi_2)$, where $\chi = \chi_1 - \chi_2$ and α is the channel between χ_1 and χ_2.)

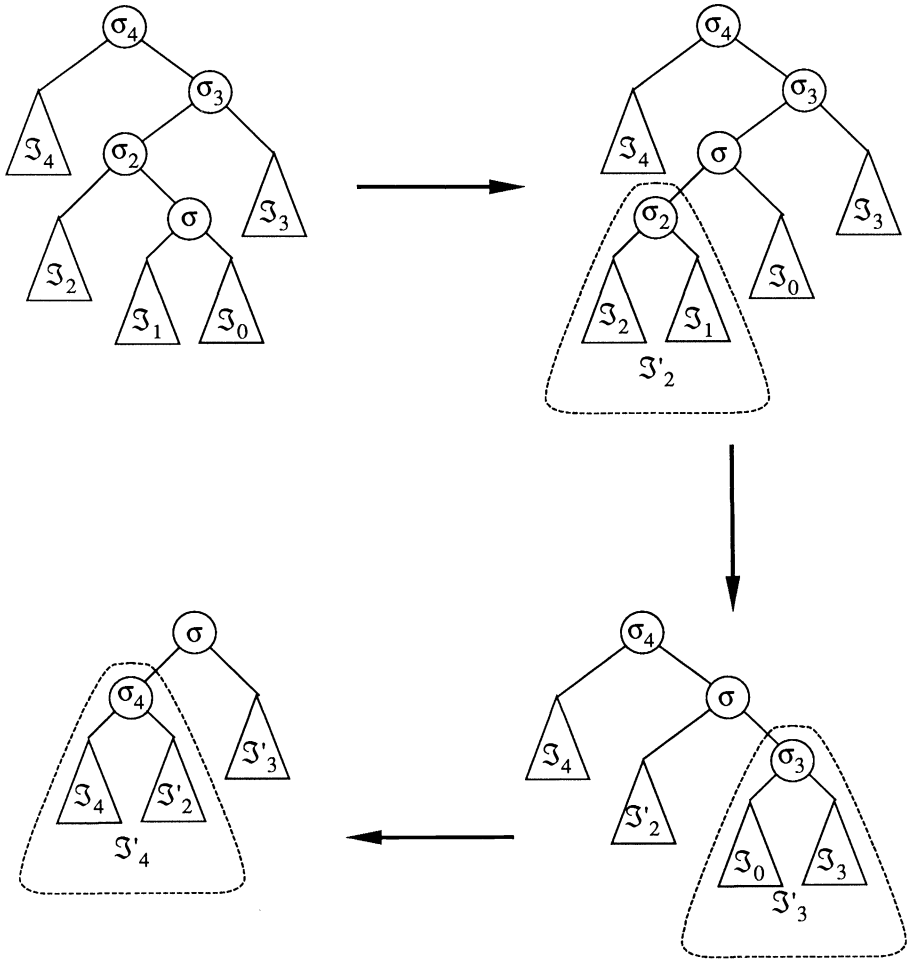


FIG. 10. *Example for step* (2) *of Procedure CUT*1.

(1) Join the two previously separated strands of α, so that the rightmost one becomes the secondary strand of the other.
(2) Split *regions*(χ) into *regions*(χ_1) and *regions*(χ_2).

LEMMA 8. *Operations* $CUT1(S, \chi_1, \chi_2; S_1, \sigma, S_2)$ *and* $CUT2(\chi; \chi_1, \gamma, \chi_2)$ *have time complexity* $O(\log^2 n)$ *and* $O(\log n)$, *respectively.*

Proof. The time bound for operation $CUT2$ is immediate. With regard to operation $CUT1$, step (1) takes $O(height(\mathbf{T}(S))) = O(\log n)$ time. In step (2), we perform no more than *height*($\mathbf{T}(S)$) rotations to bring σ to the root. After each such rotation, we have to rebalance a subtree $\mathbf{T}'$, whose left and right subtrees are already balanced, so that the number of rotations required for rebalancing is proportional to the difference of height of the subtrees of $\mathbf{T}'$. Such differences form a sequence whose sum is proportional to *height*($\mathbf{T}(S)$) [GS], [T, p. 53]. By Lemma 6, each such rotation takes $O(\log n)$ time, so that the total time complexity is $O(\log^2 n)$. □

PROCEDURE *FINDLEFT*($e; \chi$). (It finds the cluster χ to the left of edge e. If e is part of a channel, then χ is the cluster that contains such channel.)
(1) Perform a point location search for (any point of) edge e. The search will stop at a node σ of $\mathbf{T}$ that stores (a representative of) e.
(2) If e is a double edge of σ (i.e., e does not belong to a channel), resume the point location search in the left subtree of σ and return the leaf χ where the search terminates. (This corresponds to searching for a point p^- immediately to the left of edge e.)
(3) Otherwise (i.e., e is a single edge of σ and belongs to a channel) resume the point location search in both subtrees of σ. (This corresponds to searching for points p^- and p^+ immediately to the left and right of edge e, respectively.) One of the two searches, say the left one, will terminate in a leaf, while the other search, say the right one, will stop at a node σ' that stores the other representative of e. (Recall that the two nodes storing e are on the path from leaf χ to the root of $\mathbf{T}$.) We resume the search in the left subtree of σ' and return the leaf χ where the search terminates. (The case where the right search out of σ terminates in a leaf is analogous.)

PROCEDURE *FINDRIGHT*($e; \chi$). (It finds the cluster χ to the right of edge e. If e is part of a channel, then χ is the cluster that contains such channel.)
(Analogous to *FINDLEFT*.)

LEMMA 9. *Operations* $FINDLEFT(e; \chi)$ *and* $FINDRIGHT(e; \chi)$ *have each time complexity* $O(\log^2 n)$.

Proof. Since each edge has two representatives, there are at most two nodes of $\mathbf{T}$ where we proceed to both children. Hence, we visit a total of $O(\log n)$ nodes, spending $O(\log n)$ time at each node. □

The complete algorithm for operation $INSERTCHAIN(\gamma, v_1, v_2, r; r_1, r_2)$ is as follows.

ALGORITHM $INSERTCHAIN(\gamma, v_1, v_2, r; r_1, r_2)$.
(1) Find regions q and s immediately preceding and following r, respectively; also, find clusters χ_L and χ_R by means of $FINDLEFT(e_1; \chi_L)$ and $FINDRIGHT(e_2; \chi_R)$. From these obtain λ_1, λ_2, ρ_1, and ρ_2.
(2) Perform a sequence of $CUT1$ and $CUT2$ operations to decompose $\mathbf{P} = \mathbf{P}(L_1\lambda_1\text{--}\lambda_2 L_2 r R_1 \rho_1\text{--}\rho_2 R_2)$ into $\mathbf{P}(L_1)$, $\mathbf{P}(\lambda_1)$, $\mathbf{P}(\lambda_2)$, $\mathbf{P}(L_2)$, $\mathbf{P}(r)$, $\mathbf{P}(R_1)$, $\mathbf{P}(\rho_1)$, $\mathbf{P}(\rho_2)$, and $\mathbf{P}(R_2)$. The primary and secondary strands returned by each such

operation, which form the boundaries of the above partial subdivisions, are collected into a list **L**.

(3) Construct the primary and secondary strands of chain γ and add them to **L**.

(4) Destroy $\mathbf{P}(r)$ and create $\mathbf{P}(r_1)$ and $\mathbf{P}(r_2)$.

(5) Test for channels λ_1–r_1 and r_2–ρ_2, and perform a sequence of *MERGE*1 and *MERGE*2 operations to construct the updated subdivision $\mathbf{P}(L_1\lambda_1\text{--}r_1R_1\rho_1\text{-}\lambda_2L_2r_2\text{--}\rho_2R_2)$. The separators and channels needed to perform each such merge are obtained by splitting and splicing the appropriate strands of **L**.

THEOREM 8. *The time complexity of operation INSERTCHAIN*$(\gamma, v_1, v_2, r; r_1, r_2)$, *where* γ *consists of* k *edges, is* $O(\log^2 n + k)$.

Proof. In step (1), finding q and s takes $O(\log n)$ time. In fact, q is either in the cluster of r or in the cluster immediately preceding the one of r, and analogously for s. By Lemma 9, finding χ_L and χ_R takes $O(\log^2 n)$ time. The remaining computation of λ_1, λ_2, ρ_1, and ρ_2 can be done in $O(\log n)$ time. By Lemma 8, step (2) takes $O(\log^2 n)$ time. Note that the list **L** has $O(1)$ elements. Step (3) can be clearly performed in time $O(k)$. Step (4) takes $O(1)$ time since r, r_1, and r_2 are single-region structures. In step (5), testing for channels λ_1--r_1 and r_2--ρ_2 takes $O(1)$ time. Since the list **L** has $O(1)$ elements, we can construct in $O(\log n)$ time the separators and channels needed for each merge operation of step (5). By Lemma 7 the total time for such merges is $O(\log^2 n)$. □

With regard to the *INSERTPOINT* operation, we locate the edge e in the dictionary, and replace each of the two representatives of e in the data structure with the chain (e_1, v, e_2). This corresponds to performing two insertions into sorted lists, so that we have Theorem 9.

THEOREM 9. *The time complexity of operation INSERTPOINT*$(v, e; e_1, e_2)$ *is* $O(\log n)$.

A similar argument shows Theorem 10.

THEOREM 10. *The time complexity of operation MOVEPOINT*$(v; x, y)$ *is* $O(\log n)$.

4.4. Deletion. The transformations involved in a *REMOVECHAIN* operation are exactly the reverse of the ones for the *INSERTCHAIN* operation. We observe that all the updates performed in the latter case are totally reversible, which establishes Theorem 11.

THEOREM 11. *The time complexity of operation REMOVECHAIN*$(\gamma; r)$, *where* γ *consists of* k *edges, is* $O(\log^2 n + k)$.

The same situation arises with respect to the *INSERTPOINT* and *REMOVEPOINT* operations, so that we have Theorem 12.

THEOREM 12. *The time complexity of operation REMOVEPOINT*$(v; e)$ *is* $O(\log n)$.

Theorem A stated in § 1 results from the combination of the above Theorems 5, 6, 8, 9, 11, and 12.

5. Conclusion and open problems. The above technique represents a reasonably efficient solution of the dynamic point location problem. It requires no new sophisticated or bizarre data structures, and it appears eminently practical.

It remains an open problem whether $O(\log n)$ optimal performance is achievable for query/update times; in particular, whether the technique of fractional cascading, which achieved optimality for its suboptimal static predecessor [EGS], can also be successfully applied to the presented technique.

Another challenging open question is to extend our technique to general planar subdivisions.

REFERENCES

[B] G. BIRKHOFF, *Lattice Theory*, American Mathematical Society Colloquium Publications, Vol. 25, American Mathematical Society, Providence, RI, 1979.

[DL] D. P. DOBKIN AND R. J. LIPTON, *Multidimensional searching problems*, SIAM J. Comput., 5 (1976), pp. 181-186.

[EKA] 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.

[EGS] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317-340.

[F] O. FRIES, *Zerlegung einer planaren Unterteilung der Ebene und ihre Anwendungen*, M.S. thesis, Inst. Angew. Math. und Inform., Univ. Saarlandes, Saarbrücken, FRG, 1985.

[FMN] O. FRIES, K. MEHLHORN, AND S. NAEHER, *Dynamization of geometric data structures*, in Proc. ACM Symposium on Computational Geometry, 1985, pp. 168-176.

[GS] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, 1978, pp. 8-21.

[Ka] T. KAMEDA, *On the vector representation of the reachability in planar directed graphs*, Inform. Process. Lett., 3 (1975), pp. 75-77.

[KR] D. KELLY AND I. RIVAL, *Planar lattices*, Canadian J. Math., 27 (1975), pp. 636-665.

[Ki] D. G. KIRKPATRICK, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28-35.

[LP] 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.

[LEC] A. LEMPEL, S. EVEN, AND I. CEDERBAUM, *An Algorithm for Planarity Testing of Graphs*, Theory of Graphs, Internat. Symposium, Rome, Italy, 1966, P. Rosenstiehl ed., Gordon and Breach, New York, 1967, pp. 215-232.

[LT] R. J. LIPTON AND R. E. TARJAN, *Applications of a planar separator theorem*, in Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, 1977, pp. 162-170.

[M] K. MEHLHORN, *Data Structures and Algorithms* 3: *Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, New York, 1984.

[O] M. OVERMARS, *Range searching in a set of line segments*, in Proc. ACM Symposium on Computational Geometry, 1985, pp. 177-185.

[P1] F. P. PREPARATA, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473-483.

[P2] ———, *Planar point location revisited: A guided tour of a decade of research* (invited paper), Lecture Notes in Computer Science, Vol. 338 (Proc. 1988 FST & TCS Symposium, Pune, India), Springer-Verlag, Berlin, New York, 1988, pp. 11-17.

[PS] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, Springer-Verlag, Berlin, New York, 1985.

[ST] N. SARNAK AND R. E. TARJAN, *Planar point location using persistent search trees*, Comm. ACM, 29 (1986), pp. 669-679.

[TP] R. TAMASSIA AND F. P. PREPARATA, *Dynamic maintenance of planar digraphs, with applications*, *Algorithmica*, to appear. Tech. Report ACT-92, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL, 1988.

[T] R. E. TARJAN, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, 44, Society for Industrial Applied Mathematics, Philadelphia, PA, 1983.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 831–842, August 1989

014

ON THE LIMIT SETS OF CELLULAR AUTOMATA*

KAREL CULIK II†, JAN PACHL‡, AND SHENG YU§

Abstract. The limit sets of cellular automata, defined by Wolfram, play an important role in applications of cellular automata to complex systems. A number of results on limit sets are proved, considering both finite and infinite configurations of cellular automata. The main concern of this paper is with testing membership and (essential) emptiness of limit sets for linear and two-dimensional cellular automata.

Key words. cellular automaton, limit set, limit language, decidability, bi-infinite words

AMS(MOS) subject classification. 68D20

1. Introduction. Recently, cellular automata (CA) have been intensively studied as models of complex systems, especially systems containing a large number of simple components with local interactions [11], [12], [13]. An important role in these models is played by the limit sets of CA. The limit set of a CA consists of those configurations that might occur after arbitrarily many computation steps of the automaton. They correspond to the concept of attractors in the chaos theory in physics.

Many problems concerning CA limit sets are open [6], [11], [13]. In this paper, we study several versions of the "emptiness problem" for CA limit sets. We show that for $k \geq 2$ it is undecidable whether the limit set of a given k-dimensional cellular automaton consists of the quiescent configuration only. We also show that for all $k \geq 1$ it is undecidable whether the limit set of a given k-dimensional cellular automaton contains a finite configuration. The first problem for $k = 1$ is still open. However, we show that if the limit set of a given cellular automaton contains the quiescent configuration only, then all the configurations map to the quiescent configuration in a bounded number of steps. The methodology of this work is of interest. Besides automata theoretical techniques we also use, as Hurd did [6], the product topology on the space of configurations; we use the fact that this topological space is compact.

In §2 we define cellular automata and their limit sets. In §3 we endow the set of states of a CA cell with the discrete topology and observe that the space of all the configurations of the CA with the product topology is compact by Tychonoff's theorem. We then use compactness to prove several properties of the limit sets, including nonemptiness. We also use Baire's category theorem to derive a classification of cellular automata.

In the following section we prove two results about the relationship between the limit sets and the limit languages of linear (one-dimensional) automata.

In §5 we use the undecidability of the tiling problem to prove that for $k \geq 2$ it is undecidable whether the limit set of a given k-dimensional cellular automaton is a singleton.

* Received by the editors January 12, 1988; accepted for publication (in revised form) December 14, 1988. This research was supported by the Natural Science and Research Council of Canada grants A-7403 and A-0952, and National Science Foundation grant CCR-8702752.

† Department of Computer Science, University of South Carolina, Columbia, South Carolina 29208.

‡ Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. Present address, IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland.

§ Department of Mathematical Sciences, Kent State University, Kent, Ohio 44242.

A configuration of a CA is called finite if all but finite (nonzero) number of cells are quiescent. In §6 we study the limit sets of finite configurations, i.e., the intersection of the limit set with the set of finite configurations. Our main result here is that it is undecidable whether the limit set of finite configurations is empty even for linear CA. The set of all finite subwords of the configurations in a limit set is called a limit language. We show that the membership problem for CA limit languages, i.e., whether a given string is in the limit language of a given linear CA, is undecidable. Using the existence of a universal CA [1], we prove that there exists a CA whose limit language is not recursive. Similar results were proved by Hurd [6]. In §6 it is explained how our results relate to those in [6].

There are simple regular languages that are not CA limit languages. On the other hand, since the complement of any CA limit language is recursively enumerable, a limit language is recursive if and only if it is recursively enumerable. Thus the result mentioned above implies that not all limit languages are recursively enumerable [6]. Moreover, there is no obvious effective translation between the description of a recursive limit set by its CA and the description by a Turing machine. For more detailed proofs of the results in §§4–6 see [4].

2. Cellular automata: Basic definitions. Let Z be the set of integers, and Z^k the set of k-tuples of integers. A *cellular automaton*, abbreviated CA (or, more specifically, a *k-dimensional cellular automaton*, kD CA), is an infinite array, indexed by Z^k, of *cells*. Each cell is identified by its *location* $I \in Z^k$.

At any time, each cell has a *state*, which belongs to a finite set S. The dynamic behavior of the CA is determined by a rule that describes the state of each cell at time $t+1$ as a function of the states of some neighboring cells at time t. The rule is invariant with respect to translations (shifts) of Z^k.

Formally, a cellular automaton is a quadruple $A = (k, S, N, f)$, where $k \geq 1$ is the dimension, S is the finite set of states, N is the *neighborhood*, and f is the *local function* of A. The dimension k is an integer, $k \geq 1$. The (relative) neighborhood N is a sequence $(I_1, I_2, \cdots, I_h)$ of relative locations $I_j \in Z^k$, $1 \leq j \leq h$. The local function is a total function $f : S^h \to S$.

A *configuration* c of the CA is a function $c : Z^k \to S$, which assigns a state in S to each cell of the CA. The set of configurations is denoted S^{Z^k}. The local function f is extended to the *global function*

$$G_f : S^{Z^k} \to S^{Z^k}$$

of the set of configurations into itself. By definition, for $c_1, c_2 \in S^{Z^k}$,

$$G_f(c_1) = c_2$$

if and only if

$$c_2(I) = f(c_1(I + I_1), c_1(I + I_2), \cdots, c_1(I + I_h))$$

for all $I \in Z^k$.

The function G_f describes the dynamic behavior of the CA: The CA moves from the configuration c at time t to the configuration $G_f(c)$ at time $t+1$. The state of the cell I at time $t+1$ depends only on the states of the cells in the neighborhood $(I + I_1, I + I_2, \cdots, I + I_h)$ at time t. Notice that besides being locally defined, the global function G_f is total and translation invariant.

Example 1. Let $A = (1, S, N, f)$ be a one-dimensional CA, where $S = \{0, 1\}$, $N = (-2, -1, 0, 1, 2)$, and

$$f(x_1, x_2, x_3, x_4, x_5) = \begin{cases} 1 & \text{if } x_1 + x_2 + x_3 + x_4 + x_5 = 4; \\ 0 & \text{otherwise.} \end{cases}$$

If c is a configuration consisting of all 1's and c' is a configuration consisting of all 0's, then $G_f(c) = c'$.

For $c \in S^{Z^k}$, the sequence $(c, G_f(c), {G_f}^2(c), {G_f}^3(c), \cdots)$ is called the *orbit of* c.

Frequently, a state $\tilde{q}$ with the property

$$f(\tilde{q}, \tilde{q}, \cdots, \tilde{q}) = \tilde{q}$$

is distinguished and called the *quiescent state*. In a CA, there may be more than one state with the above property, but at most one of them is distinguished as the quiescent state. The configuration with all cells in the quiescent state is called the *quiescent configuration*, denoted by $\tilde{Q}$.

Let $A = (k, S, N, f)$ be a CA. Define

$$\begin{aligned} \Omega^{(0)} &= S^{Z^k}, \text{ and} \\ \Omega^{(i)} &= G_f(\Omega^{(i-1)}) \text{ for } i \geq 1 . \end{aligned}$$

Then

$$\Omega = \bigcap_{i=0}^{\infty} \Omega^{(i)}$$

is called the *limit set* of A.

Define

$$\Phi = \{ c \in S^{Z^k} \mid G_f(c) = c \}$$

(this is the set of "fixed points" of G_f). Obviously $\Phi \subseteq \Omega$.

3. The product topology on configurations. The configuration space S^{Z^k} is a product of infinitely many finite sets S. When S is endowed with the discrete topology, the *product topology* on S^{Z^k} is compact by Tychonoff's theorem [7, Thm. 5.13]. A subbasis of open sets for the product topology consists of all sets of the form

$$\{ c \in S^{Z^k} \mid c(i) = a \}, \tag{1}$$

where $i \in Z^k$ and $a \in S$. A subset of S^{Z^k} is open if and only if it is a union of finite intersections of sets of the form (1). It is easy to show that the global function G_f defined in the previous section is continuous from S^{Z^k} to S^{Z^k}. (Thus the pair (S^{Z^k}, G_f) is a classical dynamical system, in the sense of [3].)

THEOREM 3.1. *For every G_f the limit set Ω is nonempty.*

Proof. Since G_f is continuous, each $\Omega^{(i)}$, $i \geq 0$ is a continuous image of the compact space S^{Z^k}. Hence $\Omega^{(i)}$ are nonempty compact subsets of S^{Z^k}, and $\Omega^{(0)} \supseteq \Omega^{(1)} \supseteq \Omega^{(2)} \supseteq \cdots$. Therefore the intersection $\Omega = \bigcap_{i=0}^{\infty} \Omega^{(i)}$ is nonempty. □

The theorem has also an easy nontopological proof.

An alternative proof. Let $c : Z^k \to S$ be a constant function (i.e., there is $a \in S$ such that $c(I) = a$ for all $I \in Z^k$). In the orbit $(c, c_1, c_2, \cdots)$ of c, each c_j is a constant function. Since the set S is finite, there are only finitely many constant functions from Z to S, and thus there exists m such that $c_m = c_j$ for infinitely many j. Hence $c_m \in \Omega^{(i)}$ for all $i \geq 0$, and $c_m \in \Omega$. □

For some CA, the limit set Ω contains only one configuration. In particular, for a CA with a special quiescent state $\tilde{q}$, it is possible that the limit set contains only the quiescent configuration $\tilde{Q}$. It was open whether the problem $\Omega \stackrel{?}{=} \{\tilde{Q}\}$ is decidable for k-dimensional CA for $k \geq 1$. It is still open for $k = 1$.

Now we are going to use Baire's category theorem to classify cellular automata by the limit behavior of their orbits. A subset of a topological space is called a G_δ set if it is the intersection of a countable family of open sets.

THEOREM 3.2. *Let C be a closed translation-invariant subset of S^{Z^k}. Exactly one of these two conditions is true:*

(i) *There exists an integer $i \geq 0$ such that $G_f{}^i(S^{Z^k}) \subseteq C$.*

(ii) *There exists a dense G_δ set $D \subseteq S^{Z^k}$ such that*

$$C \cap \bigcup_{i=0}^{\infty} G_f{}^i(D) = \emptyset.$$

Proof. For $i = 0, 1, 2, \cdots$, let

$$F_i = \{\, c \in S^{Z^k} \mid G_f{}^i(c) \in C \,\}.$$

The sets F_i are closed and translation invariant. Let

$$D = S^{Z^k} - \bigcup_{i=0}^{\infty} F_i \ .$$

Thus D is a G_δ set in S^{Z^k} and

$$C \cap \bigcup_{i=0}^{\infty} G_f{}^i(D) = \emptyset.$$

If D is dense in S^{Z^k}, then condition (ii) holds.

If D is not dense, then $\bigcup_{i=0}^{\infty} F_i$ contains a nonempty open subset E of S^{Z^k}. The set E is locally compact; therefore, by Baire's theorem [7, Thm. 6.34], there exists i such that F_i contains a nonempty open subset of E, which is an open subset of S^{Z^k}. Since F_i is translation invariant, it follows that F_i contains a nonempty open *translation-invariant* subset of S^{Z^k}. However, every nonempty open translation-invariant subset of S^{Z^k} is dense in S^{Z^k}. Since F_i is closed, it follows that $F_i = S^{Z^k}$, and therefore (i) holds. □

In the notation of §2, $\Omega^{(i)} = G_f{}^i(S^{Z^k})$. The set $\{\tilde{Q}\}$ (the singleton set containing only the quiescent configuration) and the limit set Ω are closed and translation invariant. Thus we obtain Corollaries 1 and 2.

COROLLARY 3.3. *If $\Omega \neq \{\tilde{Q}\}$, then there exists a configuration whose orbit does not contain $\tilde{Q}$.*

Proof. If there exists i such that $\Omega^{(i)} \subseteq \{\tilde{Q}\}$, then $\Omega = \{\tilde{Q}\}$. Therefore, by Theorem 3.2, if $\Omega \neq \{\tilde{Q}\}$, then condition (ii) holds with $C = \{\tilde{Q}\}$. If $c \in D$, then the orbit of c does not meet $\{\tilde{Q}\}$. □

COROLLARY 3.4. *For each* CA, *exactly one of these two conditions is true:*

(i) *There exists an integer* $i \geq 0$ *such that* $\Omega^{(i)} = \Omega$.
(ii) *There exists a dense* G_δ *set* $D \subseteq S^{Z^k}$ *such that*

$$\Omega \cap \bigcup_{i=0}^{\infty} {G_f}^i(D) = \emptyset.$$

It is easy to find CA satisfying condition (i) in Corollary 3.4. For instance, (i) holds whenever G_f is surjective (because in that case $\Omega = \Omega^{(i)} = S^{Z^k}$ for every i). On the other hand, the CA in the following example does not satisfy (i) (and therefore it satisfies (ii)).

Example 2. Let $A = (1, S, N, f)$ be a one-dimensional CA such that $S = \{0, 1\}$, $N = (-1, 0, 1)$, and

$$f(a_{-1}, a_0, a_1) = \begin{cases} 1 & \text{if } a_{-1} = a_0 = a_1 = 1; \\ 0 & \text{otherwise.} \end{cases}$$

In this example,

$$\Omega = \{\, {}^{\omega}1^{\omega} \,\} \cup \{\, {}^{\omega}01^n0^{\omega} \mid n = 0, 1, 2, \cdots \}.$$

However,

$${G_f}^i({}^{\omega}01^{2i+1}01^{2i+1}0^{\omega}) = {}^{\omega}010^{2i+1}10^{\omega}$$

and therefore $\Omega^{(i)} \neq \Omega$ for every i, which means that condition (i) in Corollary 3.4 does not hold.

Now we prove that condition (ii) in Corollary 3.4 never holds when $\Omega = \{\tilde{Q}\}$.

THEOREM 3.5. $\Omega = \{\tilde{Q}\}$ *if and only if there exists an integer* $i \geq 0$ *such that* $\Omega^{(i)} = \{\tilde{Q}\}$.

Proof. The *if* part is trivially true. To prove the *only if* part, assume that $\Omega = \{\tilde{Q}\}$. Choose one cell $I_0 \in Z^k$, and define

$$C = \{\, c \in S^{Z^k} \mid c(I_0) \neq \tilde{q} \,\}.$$

Then C is a closed set and

$$\bigcap_{i=0}^{\infty} \left(\Omega^{(i)} \cap C\right) = \{\tilde{Q}\} \cap C = \emptyset.$$

By compactness, $\Omega^{(i)} \cap C = \emptyset$ for some i. Since $\Omega^{(i)}$ is translation invariant,

$$\Omega^{(i)} \cap \{\, c \in S^{Z^k} \mid c(I) \neq \tilde{q} \,\} = \emptyset$$

for every $I \in Z^k$. Hence $\Omega^{(i)} = \{\tilde{Q}\}$. □

Theorem 3.5 yields a semi-procedure for demonstrating that Ω contains only the quiescent configuration. To define the semi-procedure, we extend the global function G_f to operate on partial configurations. If $W \subseteq Z^k$, define

$$N^{-1}(W) = \{\, I \in Z^k \mid I + I_j \in W \text{ for } 1 \leq j \leq h \,\}$$

and for a function $c_1 : W \to S$ define

$$G_f(c_1) = c_2,$$

where $c_2 : N^{-1}(W) \to S$ is such that

$$c_2(I) = f(c_1(I + I_1), c_1(I + I_2), \cdots, c_1(I + I_h))$$

for all $I \in N^{-1}(W)$.

For $r \geq 0$, define the k-dimensional interval W_r to be the product of one-dimensional intervals $[-r, r]$; that is,

$$W_r = \{ (i_1, i_2, \cdots, i_k) \in Z^k \mid -r \leq i_j \leq r \text{ for } j = 1, 2, \cdots, k \}.$$

Denote by I_0 the origin in Z^k, i.e., the k-tuple of zeros.

COROLLARY 3.6. *Let $r \geq 0$ be such that $N \subseteq W_r$. Then $\Omega = \{\tilde{Q}\}$ if and only if there exists an integer $i \geq 0$ such that for every function $c : W_{ir} \to S$ the function ${G_f}^i(c)$ maps I_0 to $\tilde{q}$.*

Proof. The corollary follows from the theorem and from this observation, which can be proved by induction in i: If $c : W_{ir} \to S$ and $c' : Z^k \to S$ agree on W_{ir}, then ${G_f}^i(c)$ and ${G_f}^i(c')$ agree at I_0. □

The following semi-procedure determines that $\Omega = \{\tilde{Q}\}$: Let $A = (k, S, N, f)$ be the given CA. Find r such that $N \subseteq W_r$. For $i = 1, 2, \cdots$, generate all functions $c : W_{ir} \to S$, and for each such c compute the value of ${G_f}^i(c)$ at I_0. Stop when, for some i, all the values are $\tilde{q}$. This is only a semi-procedure, because it never halts when $\Omega \neq \{\tilde{Q}\}$.

We conjecture that the problem $\Omega \stackrel{?}{=} \{\tilde{Q}\}$ is decidable for *linear* (i.e., one-dimensional) CA. In §5 we show that the problem is undecidable for two-dimensional CA, and therefore also for k-dimensional CA when $k \geq 2$. In the remainder of this section, we show that the same problem for Φ (the set of fixed points of G_f, defined at the end of §2) is decidable for linear CA, although (as will be proved in Theorem 5.3) it is undecidable for dimensions $k \geq 2$.

A configuration $c : Z \to S$ is called *periodic* if there is $m > 0$ (called a *period of* c) such that $c(j + m) = c(j)$ for every $j \in Z$.

LEMMA 3.7. *For a linear CA $(1, S, N, f)$, let $r > 0$ be such that $N \subseteq [-r, r]$, and let n be the cardinality of S. If $\Phi \neq \{\tilde{Q}\}$ then there exists a periodic configuration $c \in \Phi$, $c \neq \tilde{Q}$, with period at most n^{2r+1}.*

Proof. Choose any $c' \in \Phi - \{\tilde{Q}\}$. Find the smallest $j > 0$ such that for some j_0 the restrictions of c' to the intervals $[j_0, j_0 + 2r]$ and $[j_0 + j, j_0 + j + 2r]$ are identical (modulo a shift) and $c'(j_1) \neq \tilde{q}$ for some j_1 in $[j_0, j_0 + j]$. Since there are n^{2r+1} different partial configurations $d : [-r, r] \to S$, it follows that $j \leq n^{2r+1}$. Define c to be the (unique) configuration that is equal to c' on the interval $[j_0, j_0 + j + 2r]$ and periodical with period j. □

THEOREM 3.8. *The problem $\Phi \stackrel{?}{=} \{\tilde{Q}\}$ is decidable for $k = 1$.*

Proof. In view of Lemma 3.7, the following algorithm decides whether $\Phi = \{\tilde{Q}\}$: Generate all partial configurations $c : [-r, n^{2r+1} + r] \to S$, and for each such c check whether there exists an integer m, $0 < m \leq n^{2r+1}$, such that $G_f(c)(j) = c(j)$ for $0 \leq j \leq n^{2r+1}$, $c(j) = c(j + m)$ for $-r \leq j \leq r$, and $c(0) \neq \tilde{q}$. If there is at least one c for which the test is positive, then $\Phi \neq \{\tilde{Q}\}$. Otherwise, $\Phi = \{\tilde{Q}\}$. □

4. The limit languages of linear cellular automata. In this section we assume that $A = (1, S, N, f)$; that is, A is a linear CA.

We treat a configuration of A as a bi-infinite word over the alphabet S. With every set of configurations we associate a set of finite words (strings) over S, as follows:

For a bi-infinite word $c \in S^Z$, define (as in [6])

$$L[c] = \{ w \in S^* \mid w \text{ is a finite subword of } c\},$$

and, for $C \subseteq S^Z$, define

$$L[C] = \bigcup_{c \in C} L[c].$$

$L[C]$ is called the *language of* C. If Ω is the limit set of the CA A, then we call $L[\Omega]$ the *limit language of* A.

The next theorem gives an alternative definition of the limit language $L[\Omega]$.

THEOREM 4.1. $L[\Omega] = \bigcap_{i=0}^{\infty} L[\Omega^{(i)}]$.

Proof. Since $\Omega \subseteq \Omega^{(i)}$ for every i, it follows that $L[\Omega] \subseteq \bigcap_i L[\Omega^{(i)}]$. To prove the opposite inclusion, choose any $w \in \bigcap_i L[\Omega^{(i)}]$. Let j be the length of w. Define

$$C = \{ c \in S^Z \mid c(1)c(2)\cdots c(j) = w \}.$$

Then C is a closed set and $C \cap \Omega^{(i)} \neq \emptyset$ for every i, by the choice of w and the translation invariance of $\Omega^{(i)}$. By compactness,

$$\Omega \cap C = \bigcap_{i=0}^{\infty} \left(\Omega^{(i)} \cap C\right) \neq \emptyset ,$$

which means that w is a subword of some $c \in \Omega$, hence $w \in L[\Omega]$. □

Wolfram [12] shows that the set $L[\Omega^{(i)}]$ is regular for each $i \geq 0$. This result can be proved using the fact that regular sets are closed under general sequential machine (GSM) mappings [5]. Indeed for each local function f it is easy to construct the GSM T_f that maps each word w in $L[\Omega^{(i)}]$ with $|w| \geq r$, where r is the span of the neighborhood, into the successor string of length $|w| - r$ in $L[\Omega^{(i+1)}]$. The set $L[\Omega^{(0)}] = S^*$ is regular, therefore for each $i > 0$, $L[\Omega^{(i+1)}] = T_f(L[\Omega^{(i)}])$ is regular as well. We omit the details of this proof.

A natural extension of finite automata (FA) to bi-infinite words ($\omega\omega$-words), called $\omega\omega$-FA, is described in [4]. The set of bi-infinite words recognized by an $\omega\omega$-FA is called an *$\omega\omega$-regular set.*

In [4] we prove that

(i) the sets $\Omega^{(i)}$ are $\omega\omega$-regular;

(ii) the set Ω is $\omega\omega$-regular if and only if the language $L[\Omega]$ is regular.

Hurd [6] shows that $L[\Omega]$ need not be regular.

As a corollary of the next theorem, we shall give a characterization of Ω in terms of $L[\Omega]$.

THEOREM 4.2. *If $C \subseteq S^Z$ is translation invariant, then the set $\{ c \in S^Z \mid L[c] \subseteq L[C] \}$ is the closure of C in the product topology.*

Proof. Let $D = \{ c \in S^Z \mid L[c] \subseteq L[C] \}$. The complement of D in S^Z is open. Indeed, if $c' \notin D$ then $c'(i)\cdots c'(j) \notin L[C]$ for some $i, j \in Z, i \leq j$. In that case the set

$$\{ c \in S^Z \mid c(i)\cdots c(j) = c'(i)\cdots c'(j) \},$$

which is a neighborhood of c' in the product topology, does not intersect D.

Since D is closed and $C \subseteq D$, it follows that the closure $\bar{C}$ of C is a subset of D. To prove that $D \subseteq \bar{C}$, choose any $d \in D$. Then for every $j \geq 0$ the word

$d(-j)\cdots d(j)$ is a subword of some $c_j \in C$. Since C is translation invariant, we can choose c_j so that $d(-j)\cdots d(j) = c_j(-j)\cdots c_j(j)$. But then d is the limit of the sequence $(c_j \mid j = 0, 1, \cdots)$ in the product topology, which proves that $d \in \bar{C}$. □

COROLLARY 4.3. *A configuration $c \in S^Z$ belongs to the limit set Ω if and only if $L[c] \subseteq L[\Omega]$.*

Proof. By Theorem 4.2, $\Omega = \{ c \in S^Z \mid L[c] \subseteq L[\Omega] \}$. □

5. The limit sets of two-dimensional cellular automata. In contrast to the conjecture we made for linear CA in §3, we are now going to show that it is undecidable, whether or not the limit set of a given two-dimensional CA consists of the quiescent configuration only. Consequently, the same problem for k-dimensional CA limit sets is undecidable for any $k \geq 2$. The proof is based on a well-known deep result, the undecidability of the tiling problem. The tiling problem was raised by Wang [10], and proved to be undecidable five years later by Berger [2]. Robinson gave a more readable proof in [9].

THEOREM 5.1. *It is recursively undecidable whether or not the limit set Ω of a given two-dimensional* CA *consists of $\tilde{Q}$ only.*

Proof. We show that the tiling problem can be transformed into our problem. We are given a set of tiles

$$T = \{ (l_i, r_i, u_i, d_i) \mid 1 \leq i \leq n \},$$

where l_i, r_i, u_i, and d_i denote the colors of the left, right, upper, and lower edges, respectively. In the following, we use $l(t), r(t), u(t)$, and $d(t)$ to denote the colors of the four edges of a tile t; that is, $t = (l(t), r(t), u(t), d(t))$. We construct a CA $A = (2, Q, N, f)$, where

$Q = T \cup \{\tilde{q}\}$;

$N = ((0,0), (-1,0), (1,0), (0,1), (0,-1))$;

and

$$f(t_o, t_l, t_r, t_u, t_d) = \begin{cases} t_o & \text{if } l(t_o) = r(t_l), r(t_o) = l(t_r), \\ & u(t_o) = d(t_u), \text{ and } d(t_o) = u(t_d); \\ \tilde{q} & \text{otherwise.} \end{cases}$$

Now we show that there is a valid tiling of the plane if and only if the limit set of A is not $\{\tilde{Q}\}$, where $\tilde{Q}$ denotes the quiescent configuration. If the plane can be tiled with the given tiles, then all the valid tilings are configurations in the limit set of A. If the plane cannot be tiled with the given tiles, then there is an integer $i \geq 1$ such that the square of size i cannot be tiled (this follows from König's infinity lemma [8, pp. 381-383]). By the definition of f, $G_f{}^i(c) = \tilde{Q}$ for all $c \in S^{Z^2}$. This implies that $\Omega^{(i)} = \{\tilde{Q}\}$, and the limit set Ω is equal to $\{\tilde{Q}\}$.

Since the tiling problem is undecidable, the problem $\Omega \stackrel{?}{=} \{\tilde{Q}\}$ for two-dimensional CA is also undecidable. □

COROLLARY 5.2. *It is recursively undecidable, whether or not the limit set of a given k-dimensional* CA *consists of the quiescent configuration only, for any $k \geq 2$.*

Proof. Define the local function such that only two dimensions are actually effective. A cell remains in the same nonquiescent state if its four neighbors in two specific dimensions satisfy the rule of tiling. □

The same technique does not work for linear CA because the tiling problem in one dimension is trivially decidable.

Observe that, for the given set of tiles and the cellular automaton constructed in the proof of Theorem 5.1, there is a valid tiling of the plane if and only if the set $\Phi \subseteq S^{Z^k}$ (defined at the end of §2) contains some configuration different from the quiescent configuration $\tilde{Q}$. Thus the proof of Theorem 5.1 also proves the following result.

THEOREM 5.3. *For $k \geq 2$ it is recursively undecidable whether $\Phi = \{\tilde{Q}\}$.*

6. Limit sets of finite configurations. A configuration is *finite* if the number of nonquiescent cells is finite but not zero. Let $\mathcal{F}$ denote the set of all finite configurations of a CA A. We define the *limit set of finite configurations* of A as

$$\Omega_F = \Omega \cap \mathcal{F}.$$

In this section, we show that, given an arbitrary CA, it is undecidable whether Ω_F is empty. The difficulty in transforming the Turing machine halting problem into this problem is that CA do not distinguish input symbols from working symbols. Note also that the limit set of finite configurations may be nonempty, even if every finite configuration eventually becomes quiescent. This is shown by Example 3.

Example 3. Let $A = (1, S, N, f)$ be the linear CA defined in Example 2. That is, $S = \{0, 1\}$, $N = (-1, 0, 1)$, and

$$f(a_{-1}, a_0, a_1) = \begin{cases} 1 & \text{if } a_{-1} = a_0 = a_1 = 1; \\ 0 & \text{otherwise.} \end{cases}$$

In this example, either "0" or "1" can be distinguished as the quiescent state. If "0" is the quiescent state, then the limit set of finite configurations is the set of all configurations that have exactly one substring of the form "011 $\cdots$ 10". If "1" is the quiescent state, then the limit set of finite configurations is empty.

Now, we show that given a CA it is undecidable whether Ω_F is empty. To prove this is much harder than it appears at first. One might think that since a CA can easily simulate a Turing machine (TM), we can reduce the TM halting problem to this problem using a rather standard approach. However, this does not work since the CA we are considering do not distinguish input symbols from work symbols (i.e., tape symbols for a TM). Assume that a CA A simulates a TM M. Then every configuration of M may appear as an initial string of A including the halting configurations which may never be reached by M. Two distinctive technique are used in the following proof. One is the repeated initialization of the simulation of a TM on a blank tape. The other is the continuous decreasing, rather than increasing, of the size of the simulation.

THEOREM 6.1. *Given a* CA, *it is undecidable whether $\Omega_F = \emptyset$.*

Proof. In this proof we consider linear CA only. The result can be easily extended to multidimensional CA.

Given a Turing machine M operating on a one-way infinite tape, we construct a CA A as follows. Besides the quiescent state, the state set of A consists of a left boundary state, l; a right boundary state, r; the left- and right-moving signals s_l and s_r; a yellow state, y; a destroyer, d; and a set of blue states, $B = \{b_1, b_2, \cdots, b_t\}$. The blue states are used to encode the computation of M. Before describing how A operates, we first introduce the notion of valid segments. A *segment* is the finite, consecutive, nonquiescent part of a configuration, that is surrounded by quiescent cells. A segment is *valid* if

(i) its left and right boundaries are l and r, respectively;
(ii) it has a signal symbol $s \in \{s_l, s_r\}$ between l and r ;
(iii) every cell between l and s is in a blue state;
(iv) all cells between s and r are in the yellow state.

A segment is *invalid* otherwise. The validity of a segment can be checked locally in one step. If a configuration contains an invalid segment, the destroyer d is generated and spreads. It is not difficult to prove that every finite configuration that contains an invalid segment is not in the limit set. This statement is even true for infinite configurations. A valid segment evolves as follows. See Fig. 1. The left boundary l

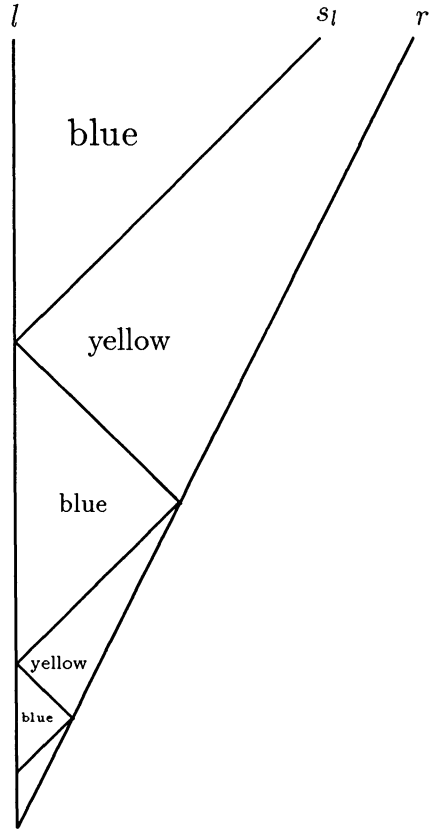


FIG. 1. *A computation of* CA *A*.

stays unchanged. The right boundary moves to the left at half speed (one cell each two steps). The signal s_l moves left at full speed (one cell each step) until it meets l. Then it changes to s_r and moves right. The signal s_r changes back to s_l when it meets r. And this repeats. Each time s_l meets l, a simulation of M starts. Each simulation is restricted to a triangle-shaped region labeled *blue* in the time-space diagram in Fig. 1. The simulation may repeatedly start and terminate until l and r meet. If a halting configuration of M is simulated, the destroyer d is generated and spreads. Then the

simulation is aborted by the spreading destroyers.

Now, we show that M starting with the blank tape halts if and only if the limit set of A does not contain a finite configuration.

If M never halts, then we can have arbitrarily large blue triangles. The configuration

$$c_0 = \cdots q\, q\, q\, l\, r\, q\, q\, q \cdots,$$

where q is the quiescent state, has an infinite history, i.e., for any integer $i \geq 0$, there exists a configuration of A such that it maps to c_0 in exactly i steps. Therefore, c_0 is in the limit set of A.

If M halts in n steps, then no blue triangle that allows the simulation of M to operate for more than n steps exists. Only the first blue triangle which has a missing upper corner can be an exception. So, it is not difficult to show that no configuration with a valid segment is in the limit set. Since, as we mentioned, no other finite configurations are possible in the limit set, the limit set of A contains no finite configurations. □

The following theorem was proved by Hurd [6, Thm. 4].

THEOREM 6.2. *Given a* CA $A = (1, S, N, f)$ *and a string* $w \in S^*$, *it is undecidable whether* w *is in the limit language of* A.

Proof. For the cellular automaton A constructed in the proof of Theorem 6.1, the string lr is in the limit language if and only if $\Omega_F \neq \emptyset$. □

Our next result (Corollary 6.3) has been stated in [6] as a direct consequence of a theorem equivalent to our Theorem 6.2. However, we feel that Corollary 6.3 does not *immediately* follow from Theorem 6.2. In order to find a nonrecursive limit language, one must show that for *one particular* CA A it is undecidable whether a given string is in the limit language of A.

COROLLARY 6.3. *There exists a linear cellular automaton such that its limit language is not recursive.*

Proof. The corollary follows from Theorem 6.2 with the help of a universal CA. A universal CA is given in [1], where any CA is simulated by encoding its local function in the states of the universal CA. Given a CA A and a string w of A, there is a string w' of the universal CA such that w' encodes both A and w. Now, the problem of whether w is in the limit language of A is transformed to the problem of whether w' (the encoding of A and w) is in the limit language of the universal CA. Since the former is undecidable, the latter is undecidable, too. □

REFERENCES

[1] J. ALBERT AND K. CULIK II, *A simple universal cellular automaton and its one-way and totalistic version*, Complex Systems, 1 (1987), pp. 1-16.

[2] R. BERGER, *The undecidability of the domino problem*, Mem. Amer. Math. Soc., 66 (1966), pp. 1-72.

[3] J. R. BROWN, *Ergodic Theory and Topological Dynamics*, Academic Press, New York, (1976).

[4] K. CULIK II, J. PACHL, AND S. YU, *On the limit sets of cellular automata*, Res. Report CS-87-47, Dept. of Computer Science, Univ. of Waterloo, 1987.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[6] L. P. HURD, *Formal language characterizations of cellular automaton limit sets*, Complex Systems, 1 (1987), pp. 69-80.

[7] J. L. KELLEY, *General Topology*, Van Nostrand, New York, 1955.

[8] D. E. KNUTH, *The Art of Computer Programming*, Vol. I, second edition, Addison-Wesley, Reading, MA, 1973.

[9] R. M. ROBINSON, *Undecidability and nonperiodicity for tiling of the plane*, Invent. Math., 12 (1971), pp. 177-209.

[10] H. WANG, *Proving theorems by pattern recognition* II, Bell System Tech. J., 40 (1961), pp. 1-41.

[11] S. WOLFRAM, *Universality and complexity in cellular automata*, Physica, 10D (1984), pp. 1-35.

[12] ———, *Computation theory of cellular automata*, Commun. Math. Phys., 96 (1984), pp. 15-57.

[13] ———, *Twenty problems in the theory of cellular automata*, Physica Scripta, T9 (1985), pp. 170-183.

SIAM J. COMPUT.
Vol. 18, No. 4, pp. 843-857, August 1989

015

EFFICIENT MESSAGE ROUTING IN PLANAR NETWORKS *

GREG N. FREDERICKSON † AND RAVI JANARDAN ‡

Abstract. The problem of routing messages along near-shortest paths in a distributed network without using complete routing tables is considered. It is assumed that the nodes of the network can be assigned suitable short names at the time the network is established. Two space-efficient near-shortest-path routing schemes are given for the class of planar networks. Both schemes use the separator property of planar networks in assigning the node names and performing the routings. For an n-node network, the first scheme uses $O(\log n)$-bit names and a total of $O(n^{4/3})$ items of routing information, each $O(\log n)$ bits long, to generate routings that are only three times longer than corresponding shortest routings in worst case[1]. For any constant ϵ, $0 < \epsilon < 1/3$, the second scheme achieves the better space bound of $O(n^{1+\epsilon})$ items, each $O((1/\epsilon)\log n)$ bits long, but at the expense of $O((1/\epsilon)\log n)$-bit node names and a worst-case bound of 7 on the routings.

Key words. distributed network, graph theory, planar graph, routing, separator, shortest paths

AMS(MOS) subject classifications. 68M10, 68Q20, 68R10, 94C15

1. Introduction.

One of the primary functions in a distributed network is the routing of messages between pairs of nodes. Assuming that a nonnegative cost, or distance, is associated with each edge, it is desirable to route along shortest paths. While this can be accomplished using a complete routing table at each of the n nodes in the network, such tables are expensive for large networks, storing a total of $\Theta(n^2)$ items of routing information, where each item is a node name. Thus, recent research has focused on identifying classes of network topologies for which the shortest paths information at each node can be stored succinctly. It is assumed that the nodes can be assigned suitable short names at the time the network is established. The idea behind naming nodes is to encode useful information about the network into the node names and then to make use of this information when performing the routing. Shortest-path routing schemes that use $O(\log n)$-bit node names and a total of $\Theta(n)$ items of routing information have been given for networks such as trees, unit-cost rings [12],[13], unit-cost complete networks, unit-cost grids [14], and networks at the lower end of a hierarchy identified in [5] (the simplest of which are the outerplanar networks [7]). Unfortunately, the approach in the research cited above becomes expensive even for very simply defined classes of networks such as, for instance, the series-parallel networks [3]. However, by shifting our focus to consider schemes that route along near-shortest paths, we have been able to design space-efficient routing schemes for much broader classes of network topologies.

The issue of saving space in routing tables by settling for near-shortest path routings was first raised in [8]. (Indeed, this is the first reported work on the problem

* Received by the editors May 21, 1987; accepted for publication (in revised form) November 6, 1988. A preliminary version of this paper appeared as a part of *Separator-based strategies for efficient message routing*, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, October 1986, pp. 428–437.

† Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. The research of this author was partially supported by National Science Foundation grants CCR-86202271 and DCR-8320124 and by Office of Naval Research Contract N 00014-86-K-0689.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455. The research of this author was partially supported by National Science Foundation grants CCR-8808574 and DCR-8320124 and by a grant-in-aid of reasearch from the Graduate School of the University of Minnesota.

[1] Unless stated otherwise, all logarithms are to the base 2.

of space-efficient routing.) Networks of general topology were studied in [8], and a clustering approach was proposed for naming the nodes. Unfortunately, no indication was given of how to do the clustering. Furthermore, the routings produced depended crucially on certain strong assumptions about the structure of the clusters, and, in worst case, could be $O(n)$ times longer than shortest routings. In this paper, and in related work [6], we consider various classes of networks that exhibit a certain separator property and show how to take advantage of this property to design space-efficient near-shortest routing schemes. All our schemes achieve routings that are, in worst case, only a small constant times longer than corresponding shortest routings. More recently, general networks with unit-cost edges have been considered in [11], and a trade-off has been established between the space used and the quality of the routings generated. Both upper and lower bounds are given for this trade-off.

In this paper we present near-shortest-path routing schemes for planar networks. A *planar network* is a network that can be embedded in the plane such that edges do not cross [7]. We measure the quality of the routings achieved by our schemes on a network by the *performance bound*, defined as the maximum ratio $\hat{\rho}(u,v)/\rho(u,v)$ taken over all pairs of nodes u, v in the network, where $\rho(u,v)$ is the length of a shortest path from u to v and $\hat{\rho}(u,v)$ is the length of the routing from u to v. We give a routing scheme that for any constant ϵ, $0 < \epsilon < 1/3$, can be set up to use $O((1/\epsilon)\log n)$-bit names and $O(n^{1+\epsilon})$ items of routing information, each $O((1/\epsilon)\log n)$ bits long, and achieve a performance bound of 7. Our approach makes use of separator strategies [9],[10] to decompose the network hierarchically and generate names for the nodes. However, using only the Lipton–Tarjan separator algorithm [9], the best we are able to achieve is a scheme that uses $O(n^{4/3})$ items, each $O(\log n)$ bits long, although with a better performance bound of 3. To reduce the storage to $O(n^{1+\epsilon})$ items, we employ a combination of very sparse routing tables and interval routing [5] to route in succession to a number of intermediate destinations carefully placed at higher levels of the decomposition. We show how Miller's algorithm [10] can be applied to generate the structured separators necessary for encoding the interval routing information.

In [6], we give a near-shortest-path routing scheme for any class of *c-decomposable networks*, i.e., networks that can be decomposed recursively by separators of size at most a constant c, where $c \geq 2$. Examples of such networks are the series-parallel networks [3], for which $c = 2$, and the k-outerplanar networks [1], for $k > 1$ a constant, for which $c = 2k$. A basic scheme is given which uses $O(cn \log n)$ items of routing information, each $O(\log n)$ bits long, and $O(\log n)$-bit names, generated from a separator-based hierarchical decomposition of the network, to achieve a performance bound of 3. We then show how to generate improved routings by including in the node names $O(c \log c \log n)$ additional bits of information about relative distances in the network. The resulting scheme has a performance bound of $(2/\alpha) + 1$, where α, $1 < \alpha \leq 2$, is the root of the equation $\alpha^{\lceil (c+1)/2 \rceil} - \alpha - 2 = 0$. Thus, the performance bound is 2 for $c \leq 3$ and ranges up to strictly less than 3 for increasing values of c.

The decomposition technique used in [6] for c-decomposable networks does not yield a space-efficient solution for planar networks, since the latter have separators of size $O(\sqrt{n})$. Instead, we take advantage of a different approach for planar graph decomposition, which is presented in [4]. This result is reviewed briefly in §2.1. The remainder of §2 describes the $O(n^{4/3})$-space scheme, called Scheme I, while §3 discusses the $O(n^{1+\epsilon})$-space scheme, called Scheme II.

2. A basic routing scheme: Scheme I.

2.1. Multilevel division and naming in Scheme I. Throughout the paper we model our network by an undirected planar graph G. (For graph-theoretic terms not defined here, see [2],[7].) For the purposes of assigning the node names and setting up the routing information, we perform a multilevel division of G into regions. A *division* of a planar graph is a grouping of its nodes into subsets called *regions*. A region contains two types of nodes, namely, interior nodes and boundary nodes. An *interior node* is contained in exactly one region and is adjacent only to nodes contained in the region, whereas a *boundary node* is contained in two or more regions.

For any parameter $f(n) < n$, an $f(n)$-*division* of a planar graph is a division of the graph into $\Theta(n/f(n))$ regions with a total of $O(n/\sqrt{f(n)})$ boundary nodes, where each region contains no more than $f(n)$ nodes and $O(\sqrt{f(n)})$ boundary nodes. An $f(n)$-division of a planar graph exists for any $f(n) < n$. An algorithm for performing an $f(n)$-division of a planar graph, based on the planar separator algorithm of [9], is given in [4]. Briefly, the $f(n)$-division algorithm involves careful, repeated application of the planar separator algorithm to G until no region has more than $f(n)$ nodes. We take the boundary nodes of the regions to be the separator nodes generated by the separator algorithm. The division has the property that any path between boundary nodes that are interior to different regions must contain a boundary node of each of these regions. We will make use of this property to do the routing. The reader is referred to [4] for more details of the $f(n)$-division algorithm. We note that although the algorithm in [4] is based on the planar separator algorithm of Lipton and Tarjan [9], it can easily be adapted to use the separator algorithm of Miller [10] as well.

The regions at various levels in the multilevel division are defined inductively as follows. The level 0 region R_1 consists of the nodes of G, with all nodes interior. In general, the name of a region is of the form R_γ, where γ is a sequence of positive integers. Let $f(n) < n$ be a parameter to be specified later. For $i \geq 1$, let R_γ be a level $i-1$ region with n' nodes. If R_γ has a nonzero number of interior nodes, then the $f(n')$-division algorithm of [4], based on the separator algorithm of [9], is applied to it to generate the level i regions $R_{\gamma 1}, R_{\gamma 2}, \cdots, R_{\gamma r}$, for some positive integer $r > 1$.

While performing the division of R_γ, we treat the boundary nodes of R_γ as boundary nodes of the resulting level i regions also. From the arguments in [4] it can be shown that, for the choice of $f(\cdot)$ to be made, the division is still an $f(\cdot)$-division. A node v that is interior to R_γ and first becomes a boundary node during its division is a *level i node*. Any other level i node generated by the division of R_γ is a *sibling* of v. A boundary node u of a level j region, $j < i$, to which v is interior is an *ancestor of v for level j*. We call v a *descendant* of u. Two nodes are *related* if one is an ancestor of the other or if they are siblings. Otherwise, they are *unrelated*.

Each level i node resulting from the division of R_γ is assigned the name γ, with an integer distinguisher appended to make the names distinct. This naming has the property that for unrelated nodes v and u, the length l of the longest common prefix of the distinguisher-free portions of their names is the smallest integer for which the nodes are in different level l regions. We call level l the *separating level for v and u*.

For the purposes of doing the routing, additional information is encoded into the name of a level i node v, identifying the closest ancestor of v for each level $j < i$. An integer, called a *level j designator*, is associated with each boundary node of the level j regions that result from the division of the level $j-1$ region to which v is interior. The level j designator of the closest ancestor of v for level j is recorded in v's name, in the jth field following the distinguisher.

The length of the names depends on the parameter $f(\cdot)$. We will show in §2.2

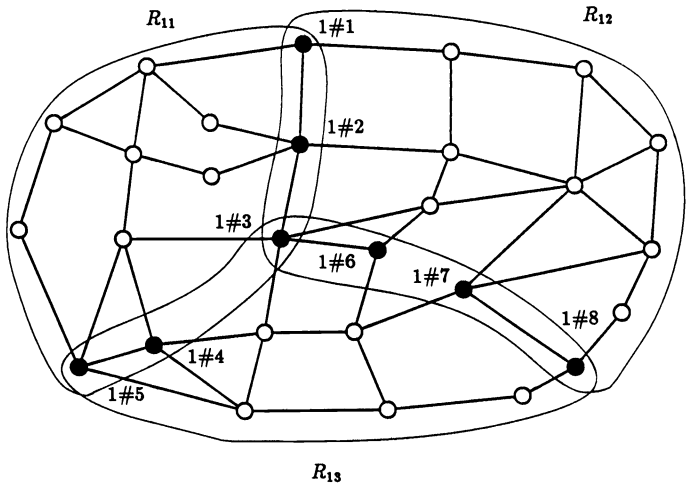


FIG. 1. *Illustration of the first level in the multilevel division and naming of a planar graph.*

that, for the choice of $f(\cdot)$ to be made, the names are $O(\log n)$ bits long.

We illustrate the decomposition and naming using Fig. 1. The given graph, constituting the level 0 region R_1, is divided into three level 1 regions: R_{11}, R_{12}, and R_{13}. The level 1 nodes, which are the boundary nodes resulting from the division, are shown filled in. The names assigned to these nodes are also shown. The "#" symbol is a delimiter and the integer following it is the distinguisher. No designators are encoded into the names of the level 1 nodes, as they do not have any ancestors.

2.2. Routing information stored at the nodes in Scheme I. Once the nodes have been assigned suitable names, based on their position in the decomposition, appropriate information is stored at them to do the routing. We first give an overview of the routing strategy in order to motivate the routing information stored. Let s be any source and d any destination. The strategy for routing from s to d depends on whether or not s and d are related. As we will see, it is not too expensive, in terms of total storage used in the network, to store a routing table at s to route to all related nodes d. However, this approach is expensive for routing between unrelated nodes, as the total number of pairs of unrelated nodes is large. Instead, the routing from s to an unrelated node d is done in three stages, as follows. Let l be the separating level for s and d; let $\hat{s}$ be the closest ancestor of s for level l; and let $\hat{d}$ be the closest ancestor of d for level l. In the first stage, the message is routed from s to $\hat{s}$, in the second stage from $\hat{s}$ to $\hat{d}$, and in the third stage from $\hat{d}$ to d. Note that since the source and destination for each stage are related nodes, routing information for each stage is available in the routing tables stored. For this approach to work, s must have available the name $\hat{s}$, and $\hat{s}$ must have available the name $\hat{d}$. The former is accomplished by storing at s the name of its closest ancestor for each level. The latter is accomplished by storing at $\hat{s}$ a table mapping the designators of certain nodes to their names. The name $\hat{d}$ is determined by $\hat{s}$ by indexing into this table using the designator of the closest ancestor of d for level l, which has been encoded in d's name.

One problem that can arise in the routing strategy described is the following.

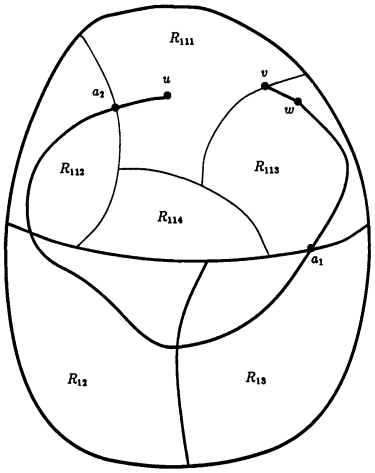


FIG. 2. *Illustration of the next milestone, a_1, and the final milestone, a_2, for u at v.*

Consider the routing from some node v to a related node u during some phase in the routing from s to d, where v and u are possibly s and d, respectively. Let w be the node to which v must send the message, as determined by v from its routing table. Now, if w and u are unrelated, then w will not be able to continue the routing to u. The problem is overcome by having v make available to w, in the message header, the names a_1 and a_2 of a pair of related nodes on a shortest (v,u)-path through w, where a_1 is an ancestor of w and a_2 an ancestor of u. The routing from w to u proceeds through a_1 and a_2, in that order. The nodes a_1 and a_2, called *the next milestone for u at v* and *the final milestone for u at v*, respectively, are defined as follows. Let R be the region such that w and u are both in R but are in different regions R' and R'' that result from the division of R. Then a_1 and a_2 are, respectively, the first boundary nodes of R' and R'' on a shortest (v,u)-path through w.

Figure 2 illustrates schematically a two-level division of a planar graph. A shortest (v,u)-path is shown in bold, where the neighbor w of v on this path and node u are unrelated. Both w and u are in region R_{11}, but in different regions, R_{113} and R_{111}, which result from the division of R_{11}. Nodes a_1 and a_2 are the first boundary nodes of R_{113} and R_{111}, respectively, on the shortest (v,u)-path.

We can now describe the routing information stored at any node v in the network. A routing table is maintained at v, giving for each related node u the name $next\_node_v(u)$ of the next node on a shortest (v,u)-path in G. To route to u, v sends the message to $next\_node_v(u) = w$ over edge $\{v,w\}$. In addition, v also has two tables containing the next and final milestone information. If w and u are unrelated, then the names of the next milestone for u at v and the final milestone for u at v are stored at v in $next\_milestone_v(u)$ and $final\_milestone_v(u)$, respectively.

The following information is stored at v to enable it to route to unrelated nodes. Let v be a level i node. Then, for each level $j < i$, the name of the closest ancestor of v for level j is stored at v in a table. Furthermore, consider the division of the level $i-1$ region to which v is interior into level i regions. A table is stored at v, mapping the

level i designator of each boundary node of the level i regions to its name.

The amount of routing information in the network depends on $f(\cdot)$. As the following theorem shows, the appropriate choice for $f(n)$ is $n^{2/3}$.

THEOREM 2.1. *For any n-node planar graph, Scheme* I *uses a total of $O(n^{4/3})$ items of routing information, where each item is $O(\log n)$ bits long.*

Proof. Each item of routing information held at a node is a node name. As will be seen in Theorem 2.2, each node name in Scheme I is $O(\log n)$ bits long.

We bound the number of items of routing information as follows. We first bound the total number of items of routing information held by all level 1 nodes for the other nodes, as well as the number of items of routing information held by all the other nodes for the level 1 nodes. We then use this to develop a recurrence that gives the total number of items of routing information held by all the nodes in the network.

Since there are $O(n/\sqrt{f(n)})$ level 1 nodes, the level 1 nodes together maintain a total of $O((n/\sqrt{f(n)})^2)$ items of shortest paths information for siblings. As there are $\Theta(n/f(n))$ level 1 regions, each containing $O(f(n))$ nodes and $O(\sqrt{f(n)})$ boundary nodes, the level 1 nodes store a total of $O(\sqrt{f(n)}f(n)n/f(n))$, i.e., $O(n\sqrt{f(n)})$ items of shortest paths information for descendants. The descendants of the level 1 nodes store $O(\sqrt{f(n)}f(n)n/f(n))$, i.e., $O(n\sqrt{f(n)})$ items of shortest paths information overall for the level 1 nodes (ancestors for level 1). Thus the number of items of routing information stored by level 1 nodes for related nodes and vice versa is $O((n/\sqrt{f(n)})^2 + n\sqrt{f(n)})$. The number of items of milestone information held by the level 1 nodes for the other nodes and vice versa is of this same order.

The size of the table of designators of a level 1 node is $O(n/\sqrt{f(n)})$, so that the space used by the designator tables of all level 1 nodes is $O((n/\sqrt{f(n)})^2)$. Finally, the descendants of the level 1 nodes store $O(f(n)n/f(n))$, i.e., $O(n)$ items of information identifying nearest level 1 nodes.

Let $S(n)$ be the total number of items of information stored in an n-node network. Then, for positive constants a, b, and c, we have $S(n) \leq an^2/f(n) + bn\sqrt{f(n)} + c(n/f(n))S(f(n))$, where the last term accounts for the information stored at lower levels in the decomposition. We choose $f(n) = n^{2/3}$ to make the opposing terms $an^2/f(n)$ and $bn\sqrt{f(n)}$ equal to within a constant factor. Thus $S(n) \leq dn^{4/3} + cn^{1/3}S(n^{2/3})$ for some positive constant d. We choose the range of n, for which the above recurrence holds, as $n \geq (2c)^9$. Thus we have,

$$S(n) \leq dn^{4/3} + cn^{1/3}S(n^{2/3}), \quad \text{for } n \geq (2c)^9.$$

For some positive constant e, we may write the basis cases as

$$S(n) \leq e, \quad \text{for } 1 \leq n < (2c)^9.$$

Then we claim that $S(n) \leq gn^{4/3}$, where $g = \max\{e, 2d\}$. The claim can be shown by induction on n. The claim clearly holds for the basis. For the induction step, for which $n \geq (2c)^9$, we require $dn^{4/3} + cn^{1/3}gn^{8/9} \leq gn^{4/3}$, i.e., $d + cgn^{-1/9} \leq g$, i.e., $d + g/2 \leq g$, which is true. The theorem follows.

Note that our choice of the threshold for n as $(2c)^9$ was arbitrary. A threshold of $((1+\delta)c)^9$ for any $\delta > 0$ will do. Correspondingly, $g = \max\{e, (1+1/\delta)d\}$. □

We next show that for the above choice of $f(n) = n^{2/3}$ the node names are only $O(\log n)$ bits long. With Scheme II in mind, we prove a more general result. We show that for $f(n) = n^{1-\epsilon}$, where ϵ is any constant, $0 < \epsilon < 1$, the node names are $O((1/\epsilon)\log n)$ bits long.

THEOREM 2.2. *Consider the naming of the nodes of an n-node planar graph from a multilevel division, performed with respect to a parameter $f(n) = n^{1-\epsilon}$, where ϵ is any constant, $0 < \epsilon < 1$. The node names are $O((1/\epsilon)\log n)$ bits long.*

Proof. At level 0 in the decomposition there is just one region of size n and no boundary nodes. It is easy to show by induction that the division of a level $j-1$ region into level j regions, $j \geq 1$, results in $O(n^{(1-\epsilon)^{j-1}\epsilon})$ level j regions and $O(n^{(1-\epsilon)^{j-1}(1+\epsilon)/2})$ boundary nodes of these level j regions. Furthermore, the highest level number L in the decomposition is at most $1 + \log_{1/(1-\epsilon)} \log n$, which is $1 + \log\log n/\log(1/(1-\epsilon))$. This can be seen as follows. Since a level j region has at most $n^{(1-\epsilon)^j}$ nodes, the level number at which all regions have at most two nodes is at most $\log_{1/(1-\epsilon)} \log n$. At most one node of each region with two nodes can be an interior node. Thus at most one additional level is needed to get regions with no interior nodes, at which point the decomposition terminates.

For $i \geq 1$, the name of a level i node v consists of $2i$ fields: i integers that constitute the name of the level $i-1$ region to which v is interior; an integer distinguisher; and $i-1$ integers, each a level j designator, $0 \leq j \leq i-1$. These fields are separated by $2i-1$ delimiters. The delimiter and the bits 0 and 1 used in the binary representation of the fields can be encoded using two bits each.

The number of bits needed to encode the region name is at most

$$\begin{aligned} 2\Big(1 + \sum_{j=1}^{i-1}\big(\lceil \log n^{(1-\epsilon)^{j-1}\epsilon}\rceil + O(1)\big)\Big) &= O\Big(i + \sum_{j=1}^{i-1} \log n^{(1-\epsilon)^{j-1}\epsilon}\Big) \\ &= O(i + (1-(1-\epsilon)^{i-1})\log n) \\ &= O(i + \log n). \end{aligned}$$

The number of bits needed for the distinguisher is at most

$$\begin{aligned} 2\lceil \log n^{(1-\epsilon)^{i-1}(1+\epsilon)/2}\rceil + O(1) &= O(\log n^{(1-\epsilon)^{i-1}(1+\epsilon)/2}) \\ &= O((1-\epsilon)^{i-1}((1+\epsilon)/2)\log n) \\ &= O(\log n), \quad \text{since } \epsilon < 1. \end{aligned}$$

The number of bits needed to encode all the designators is at most

$$\begin{aligned} 2\sum_{j=1}^{i-1}\big(\lceil \log n^{(1-\epsilon)^{j-1}(1+\epsilon)/2}\rceil + O(1)\big) &= O\Big(i + \sum_{j=1}^{i-1} \log n^{(1-\epsilon)^{j-1}(1+\epsilon)/2}\Big) \\ &= O(i + (((1+\epsilon)/2)(1-(1-\epsilon)^{i-1})/\epsilon)\ \log n) \\ &= O(i + (1/\epsilon)\log n). \end{aligned}$$

Finally, the delimiters can be encoded using $2(2i-1)$, i.e., $O(i)$ bits in all.

Summing these and simplifying, the total number of bits needed to encode the name of v is

$$\begin{aligned} O(i + (1 + (1/\epsilon))\log n) &= O(i + (1/\epsilon)\log n), \quad \text{since } 0 < \epsilon < 1 \text{ implies } (1/\epsilon) > \epsilon \\ &= O((1/\epsilon)\log n + \log\log n/\log(1/(1-\epsilon))), \quad \text{since } i \leq L \\ &= O((1/\epsilon)\log n), \quad \text{noting that } \log(1/(1-\epsilon)) > \epsilon. \end{aligned}$$

This proves the theorem. □

2.3. The routing strategy in Scheme I. A message is routed from a source s to a destination d as follows. The message header contains separate fields for the next milestone, final milestone, and the destination, all initially set to d. The next and final milestone fields alone are reset, as necessary, during the routing. Let d' and d'', respectively, denote the current names in the next milestone and final milestone fields. Each node v participating in the routing performs a *routing action* as follows. It sets $w = next\_node_v(d')$, and resets d'' to $final\_milestone_v(d')$, and d' to $next\_milestone_v(d')$ if these entries for d' are stored at v. It then sends the message to w.

The routing begins with s searching its routing table for d', which is initially d. If found, then s and d are related, and s performs a routing action. Otherwise, let l be the separating level for s and d (l can be determined from the names s and d), and let $\hat{s}$ be the closest ancestor of s for level l. Then s resets d' and d'' to $\hat{s}$ and performs a routing action.

Let v be any node that the message arrives at subsequently. If $v \neq d'$, then v performs a routing action.

If $v = d' \neq d''$, then v sets d' to d'' and performs a routing action.

Suppose that $v = d' = d'' \neq d$. If v and d are related, then v sets d' and d'' to d and performs a routing action. Otherwise, v must be $\hat{s}$, and $\hat{s}$ must be a level l node (Lemma 2.3 below). Using its table of level l designators and the lth field designator in d's name, v determines the closest ancestor $\hat{d}$ of d for level l. It then sets d' and d'' to $\hat{d}$ and performs a routing action.

If $v = d' = d'' = d$, then the routing terminates.

LEMMA 2.3. *Let l be the separating level for source s and destination d, and let $\hat{s}$ be the closest ancestor of s for level l. In the routing from s to d, let v be any final milestone different from d. If v and d are unrelated, then v must be $\hat{s}$, and $\hat{s}$ must be a level l node.*

Proof. Clearly, if s and d are related, then so are v and d. Thus, assume that s and d are unrelated. We first show that for each v, v and d are related, except possibly when v is $\hat{s}$.

In the routing from s to $\hat{s}$, v is always $\hat{s}$, since $\hat{s}$ is an ancestor of every node in the routing. If $\hat{s}$ and d are related, the routing is from $\hat{s}$ to d. Thus every v in this routing is an ancestor of d. However, if $\hat{s}$ and d are unrelated, then the routing is from $\hat{s}$ to $\hat{d}$. Every v in this phase is an ancestor of $\hat{d}$, and hence an ancestor of d. The message eventually reaches a final milestone that is either $\hat{d}$, or an ancestor of $\hat{d}$. Thus, in the routing from this node to d, every v is an ancestor of d. Every v in this routing is an ancestor of d.

Thus, if v and d are unrelated, then v must be $\hat{s}$. Suppose that $\hat{s}$ is a level $j < l$ node. Thus $\hat{s}$ is a boundary node of the level $l-1$ region to which s is interior. But, since s and d are interior to the same level $l-1$ region, $\hat{s}$ must be an ancestor of d, a contradiction. Thus $\hat{s}$ must be a level l node. □

We now establish an upper bound on the length of the routings generated by Scheme I. First, we obtain in the following lemma a lower bound on the distance between nodes s and d in the case that they are unrelated.

LEMMA 2.4. *Let s and d be unrelated nodes in the multilevel division of a planar graph. Let $\hat{s}$ be the closest ancestor for s, and let $\hat{d}$ be the closest ancestor of d for the separating level of s and d. Then $\rho(s,d) \geq \rho(s,\hat{s}) + \rho(d,\hat{d})$.*

Proof. Let s' and d' be the ancestors of s and d, respectively, on a shortest (s,d)-path. Thus $\rho(s,d) \geq \rho(s,s') + \rho(d,d')$. But $\rho(s,s') \geq \rho(s,\hat{s})$ and $\rho(d,d') \geq \rho(d,\hat{d})$, by

our choice of $\hat{s}$ and $\hat{d}$. The lemma follows. □

THEOREM 2.5. *For any planar graph, the performance bound of Scheme* I *is* 3.

Proof. Let s be any source and d any destination. If s and d are related, then the routing is along a shortest (s,d)-path. This is because every node participating in the routing performs a routing action with respect to d', which is always on a shortest (s,d)-path. Otherwise, s routes to ancestor $\hat{s}$, and if $\hat{s}$ and d are related, then $\hat{s}$ routes to d. As both routings are along shortest paths, we have

$$\begin{aligned}\hat{\rho}(s,d) &= \rho(s,\hat{s}) + \rho(\hat{s},d)\\ &\leq \rho(s,\hat{s}) + \rho(\hat{s},s) + \rho(s,d)\\ &\leq 3\rho(s,d), \text{ as } \rho(s,\hat{s}) \leq \rho(s,d) \quad \text{by Lemma 2.4.}\end{aligned}$$

If $\hat{s}$ and d are unrelated, then $\hat{s}$ routes to $\hat{d}$, where $\hat{d}$ is a sibling or an ancestor. Consider the first occasion that a final milestone $\hat{d}'$ is reached, where $\hat{d}'$ is either $\hat{d}$, or an ancestor of $\hat{d}$, and hence an ancestor of $\hat{s}$ and d. The message is routed from $\hat{d}'$ to d. The routings from $\hat{s}$ to $\hat{d}'$ and from $\hat{d}'$ to d are both along shortest paths. Thus,

$$\begin{aligned}\hat{\rho}(s,d) &= \rho(s,\hat{s}) + \rho(\hat{s},\hat{d}') + \rho(\hat{d}',d)\\ &\leq \rho(s,\hat{s}) + \rho(\hat{s},\hat{d}') + \rho(\hat{d}',\hat{d}) + \rho(\hat{d},d)\\ &= \rho(s,\hat{s}) + \rho(\hat{s},\hat{d}) + \rho(\hat{d},d), \text{ since } \hat{d}' \text{ is on a shortest } (\hat{s},\hat{d})\text{-path}\\ &\leq \rho(s,\hat{s}) + \rho(\hat{s},s) + \rho(s,d) + \rho(d,\hat{d}) + \rho(\hat{d},d)\\ &\leq 3\rho(s,d), \text{ as } \rho(s,\hat{s}) + \rho(\hat{d},d) \leq \rho(s,d) \quad \text{by Lemma 2.4.}\end{aligned}$$

Thus $\hat{\rho}(s,d)/\rho(s,d) \leq 3$ for any nodes s and d, and the theorem follows. □

3. Improving the space bound: Scheme II.

3.1. Multi-level division and naming in Scheme II. We now give Scheme II, in which the storage is reduced to $O(n^{1+\epsilon})$ items, where ϵ is any constant, $0 < \epsilon < 1/3$. The scheme uses $O((1/\epsilon)\log n)$-bit names and has a performance bound of 7. To reduce the storage, we maintain at each node a routing table for only certain closest ancestors and descendants. However, the previous routing strategy will not work now, since the routing tables are very sparse. To overcome this problem, we introduce an additional phase in the routing, in which the message is routed to a pair of intermediate destinations carefully placed at a higher level in the decomposition. We show how to choose a good, though not necessarily optimal, path for this phase, for which the multi-interval labeling scheme from [5] can be used to encode succinctly the routing information in interval form.

The network is decomposed essentially as in Scheme I. However, in order to set up the multi-interval routing information, the boundary nodes of each region must lie on one or more cycles. For a triangulated planar graph, Miller's algorithm [10] yields an $O(\sqrt{n})$-separator that is a simple cycle. The desired regions can be generated by using this, instead of the Lipton–Tarjan separator algorithm [9], in the $f(\cdot)$-division algorithm. The graphs induced on the regions at each level are first triangulated. The cost of the triangulating edges is chosen large enough (for instance, greater than the sum of the costs of all edges in G) so that shortest paths are unaffected. (We remark that the triangulation is done only so that Miller's algorithm can be applied. These edges will not be used in the routing itself, and they do not have to be added physically to the network.) The faces of each graph are then assigned zero weight (as Miller's algorithm requires that faces be weighted), and the $f(\cdot)$-algorithm is then

applied to generate the regions at the next level. As in Scheme I, the boundary nodes of each region are considered to be boundary nodes of the regions resulting from its division. The nodes are named as in Scheme I. As seen later, in Scheme II we choose $f(n) = n^{1-\epsilon}$, where ϵ is any constant, $0 < \epsilon < 1/3$. Thus Theorem 2.2 applies and the node names are $O((1/\epsilon)\log n)$ bits long.

3.2. Routing information in Scheme II. Let v be a level j node, $j \geq 1$. For each level $i \geq j$, shortest paths information is maintained at v for only those of its descendants for which it is the closest ancestor for level i. Let T be a tree of shortest paths from v to these descendants. Starting at v, depth-first numbers are assigned to the nodes of T, and at each node, the edge joining it to a child is labeled by a subinterval of depth-first numbers, representing all nodes in the subtree rooted at the child. A table is stored at v, mapping node names to depth-first numbers. A shortest routing from v to any node u in T is performed by having each node on the path use the depth-first number of u, recorded in the message header by v, to choose the appropriate edge over which to route.

For each level $i < j$, the closest ancestor of v for level i is identified, and the name of the parent of v in the tree rooted at that ancestor is stored at v. Thus v can perform a shortest routing to this ancestor.

For any level $i \geq j$, let R be a level i region for which v is a boundary node. Let R' be the level $i-1$ region containing R; let B be the set of boundary nodes associated with the division of R'. The following information maintained at v enables it to route to the nodes in B.

A table of designators is stored at v, mapping the level i designator of each node u in B to its name.

For each u, a level number $\bar{i}$ is maintained at v, where $\bar{i} \leq i$ is the largest integer for which there is a shortest (v,u)-path in G wholly in the level $\bar{i}-1$ region $\bar{R}$ containing R.

Furthermore, consider each u for which there is at least one (v,u)-path in G wholly in R', and let P be a least-cost such path. The name $next\_milestone_v(u)$ of the first node from B on P (in the direction from v to u) is stored at v. The routing from v to u is performed along P. Path P consists of segments, each of which is wholly in some level i region resulting from the division of R', and whose endpoints are boundary nodes of the level i region. Furthermore, each segment is a shortest such segment. For instance, the first segment has endpoints v and $next\_milestone_v(u)$, and, without loss of generality, lies wholly in region R. Each intermediate node on this segment routes to $next\_milestone_v(u)$. The routing information for this segment can be set up using the multi-interval labeling scheme from [5], as follows.

In the decomposition, the boundary nodes of each region lie on cycles. Let region R have t boundary nodes lying on $p \geq 1$ cycles. Associate an integer between 1 and t, called an *interval name*, with each boundary node by proceeding around each cycle in turn, as described in [5].

The following lemma shows that the routing information for the boundary nodes of R can be encoded succinctly at each node of R as subintervals of interval names labeling each incident edge.

LEMMA 3.1. *At any node w of R, the ends of all the edges incident with w can be labeled with at most $3p + degree(w) - 2$ subintervals of $[1,t]$ such that the following is true. Let z be any boundary node of R reachable from w by a path in G that is wholly contained in R. Then, the first edge on a shortest such (w,z)-path is the one whose label at w contains the interval name of z.*

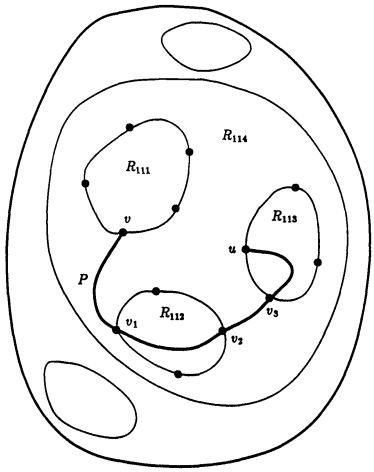


FIG. 3. *Multi-interval routing from v to u in Scheme* II.

Proof. Consider the graph G_R defined by the nodes and edges of R. A shortest (w, z)-path in G that is wholly contained in R is a shortest (w, z)-path in G_R. The lemma then follows from Corollary 5.1 in [5], since all the boundary nodes z lie on at most p faces in G_R. □

The edges incident with each node of R are labeled with subintervals of interval names. At boundary node v, a table mapping the names of the other boundary nodes of R to their respective interval names with respect to R, is stored. The routing from v to $next\_milestone_v(u)$ is performed by having each participating node use the interval name of $next\_milestone_v(u)$, recorded in the message header by v, to choose the appropriate edge over which to route.

The multi-interval routing is illustrated schematically in Fig. 3, which shows four regions, R_{111}, R_{112}, R_{113}, and R_{114}, resulting from the division of level 1 region R_{11}. (For clarity, the figure is not drawn to scale and not all regions and region names are shown. Also, all interior nodes have been omitted.) The boundary nodes resulting from the division are shown filled in. For boundary nodes v and u, a least cost (v, u)-path P contained wholly in R_{11} is shown bold. The routing over P consists of four phases of multi-interval routing, as follows: from v to $v_1 = next\_milestone_v(u)$; from v_1 to $v_2 = next\_milestone_{v_1}(u)$; from v_2 to $v_3 = next\_milestone_{v_2}(u)$; and finally, from v_3 to u.

The following theorem bounds the number of items of routing information used by Scheme II.

THEOREM 3.2. *For any n-node planar graph, Scheme* II *can be set up to use $O(n^{1+\epsilon})$ items of routing information, where ϵ is any constant, $0 < \epsilon < 1/3$. Each item is $O((1/\epsilon)\log n)$ bits long.*

Proof. Each item of routing information stored at a node in Scheme II is one of the following: a node name, a level number, a depth-first number, or an interval name. From Theorem 2.2, the length of a node name is $O((1/\epsilon)\log n)$ bits. Each depth-first number and interval name is an integer in the range 1 to n, and so $O(\log n)$ bits suffice

for these. Each level number is an integer of magnitude $O(\log\log n/\log(1/(1-\epsilon)))$, i.e., $O((1/\epsilon)\log\log n)$. Thus, $O(\log((1/\epsilon)\log\log n))$, i.e., $O(\log(1/\epsilon)+\log\log\log n)$ bits are needed for a level number. It follows that an item is $O((1/\epsilon)\log n+\log(1/\epsilon))$ bits long, which is $O((1/\epsilon)\log n)$, since $(1/\epsilon)>\log(1/\epsilon)$.

We count the number of items of routing information by levels. The level 1 nodes claim, in the role of closest ancestors for level 1, disjoint subsets of the level $j>1$ nodes. Thus the tables at the level 1 nodes, which map node names to depth-first search numbers, use a total of $O(n)$ space. Furthermore, the total number of subintervals of depth-first search numbers maintained for level 1 is proportional to the number of edges of G, which is $O(n)$.

Each level $j>1$ node maintains a constant number of items for its closest ancestor for level 1. Thus the level j nodes maintain a total of $O(n)$ items about nearest ancestors for level 1.

Each boundary node of a level 1 region maintains a constant number of items (a level 1 designator, a level number, and *next_milestone*$(\cdot)$) for each of the other boundary nodes. Since there are $O(n/\sqrt{f(n)})$ boundary nodes of level 1 regions, a total of $O((n/\sqrt{f(n)})^2)$, i.e., $O(n^2/f(n))$ items is stored.

The storage used by the multi-interval routing scheme for level 1 is as follows. Each boundary node of a level 1 region maintains, for each of the other boundary nodes of the region, an entry in the table that maps node names to interval names. Since there are $O(\sqrt{f(n)})$ boundary nodes per level 1 region, a total of $O((\sqrt{f(n)})^2)$, i.e., $O(f(n))$ items are stored per level 1 region. Thus, as there are $\Theta(n/f(n))$ level 1 regions, a total of $O(f(n)n/f(n))$, i.e, $O(n)$ items are stored for all level 1 regions. By Lemma 3.3 below, $O(n)$ intervals are used to encode the multi-interval routing information.

Thus the total number of items of routing information associated with level 1 is $O(n^2/f(n)+n)$, i.e., $O(n^2/f(n))$, since $f(n)<n$. Let $S(n)$ be the total number of items stored in an n-node network. Then, for positive constants c and d we have $S(n)\le dn^2/f(n)+c(n/f(n))S(f(n))$. We choose $f(n)=n^{1-\epsilon}$, for any constant ϵ, $0<\epsilon<1/3$, and the range of n for which the above recurrence holds as $n\ge(2c)^{1/\epsilon^2}$. Thus,

$$S(n)\le dn^{1+\epsilon}+cn^{\epsilon}S(n^{1-\epsilon}),\quad \text{for } n\ge(2c)^{1/\epsilon^2}.$$

For e a positive constant, we may write the basis cases as

$$S(n)\le e,\quad \text{for } 1\le n<(2c)^{1/\epsilon^2}.$$

We claim that $S(n)\le gn^{1+\epsilon}$, where $g=\max\{e,2d\}$. The proof is by induction and is similar to that in Theorem 2.1. Remarks analogous to those in Theorem 2.1 apply here as well, for the choice of the threshold for n. □

The multi-interval routing scheme at level 1 enables routing between the boundary nodes of the various level 1 regions that result from the division of the level 0 region, which has n nodes. We show in the following lemma that this scheme uses a total of $O(n)$ intervals. (In general, for any level $i\ge1$, there are a number of multi-interval routing schemes. Each scheme is associated with a level $i-1$ region and enables routing between certain boundary nodes of the level i regions resulting from the division of the level $i-1$ region. The number of intervals used for each scheme is proportional to the size of the corresponding level $i-1$ region.)

LEMMA 3.3. *The multi-interval labeling scheme at level* 1 *uses a total of* $O(n)$ *intervals.*

Proof. We first derive an upper bound on the number of cycles on the boundaries of the level 1 regions, counting separately each occurrence of a cycle on a level 1 region boundary. In worst case the cycles are all vertex-disjoint, so that the number of cycles is one less than the number of level 1 regions, which is $\Theta(n/f(n))$. Since each cycle is on the boundary of two level 1 regions, the desired upper bound is $\Theta(n/f(n))$.

Let R be any level 1 region. Let p_R be the number of cycles on the boundary of R, and let w be any node of R. From Lemma 3.1 it follows that the total number of intervals maintained for R by all nodes w of R is less than

$$\sum_{w \in R} (3p_R + degree(w)) \leq 3p_R cf(n) + 2(3cf(n) - 6),$$

since R has at most $cf(n)$ nodes for some constant c, and the induced subgraph of G on R is planar. Thus the total number of intervals for all level 1 regions R is less than

$$\sum_{\text{all } R} (3p_R cf(n) + 6cf(n)) = 3cf(n) \sum_{\text{all } R} p_R + 6cf(n) \sum_{\text{all } R} 1,$$

which is $O(n)$, since, from the first part of the lemma, $\sum_{\text{all } R} p_R$ is $\Theta(n/f(n))$, and since there are $\Theta(n/f(n))$ level 1 regions. □

3.3. The routing strategy in Scheme II. The routing from s to d is as follows. Irrespective of whether or not s and d are related, the routing is always performed via the closest ancestor $\hat{s}$ of s and the closest ancestor $\hat{d}$ of d for level l, where l is the length of the longest common prefix of the distinguisher-free portions of the names of s and d. The routing from s to $\hat{s}$ is along a shortest-path tree rooted at $\hat{s}$. Using its table of designators and the lth field designator in d's name, $\hat{s}$ determines $\hat{d}$. Unfortunately, unlike Scheme I, it is now not possible to perform shortest paths routing from $\hat{s}$ to $\hat{d}$, since, in general, this information will not be available at $\hat{s}$. Instead, the routing from $\hat{s}$ to $\hat{d}$ is performed along a near-shortest path as follows.

Let $\bar{l} \leq l$ be the level number maintained at $\hat{s}$ for $\hat{d}$. Thus there is a shortest $(\hat{s}, \hat{d})$-path in G that is wholly contained in the enveloping level $\bar{l} - 1$ region. Let $\hat{\hat{s}}$ and $\hat{\hat{d}}$ be the closest ancestors of $\hat{s}$ and $\hat{d}$, respectively, for level $\bar{l}$. If $\bar{l} = l$, then we take $\hat{\hat{s}}$ and $\hat{\hat{d}}$ to be just $\hat{s}$ and $\hat{d}$, respectively. The routing from $\hat{s}$ to $\hat{d}$ is performed in three stages, as follows. Node $\hat{s}$ records $\bar{l}$ and $\hat{d}$ in the message header and routes to $\hat{\hat{s}}$ along a shortest $(\hat{s}, \hat{\hat{s}})$-path. Using its table of designators and the $\bar{l}$th field designator in $\hat{d}$'s name, $\hat{\hat{s}}$ determines $\hat{\hat{d}}$. It then uses interval routing information to route to $\hat{\hat{d}}$ along a path P of least cost from among those that are wholly contained in the level $\bar{l} - 1$ region. Note that at least one such path exists, namely, the one consisting of the shortest paths from $\hat{\hat{s}}$ to $\hat{s}$, from $\hat{s}$ to $\hat{d}$, and from $\hat{d}$ to $\hat{\hat{d}}$. Node $\hat{\hat{d}}$ then routes to $\hat{d}$ along a shortest $(\hat{\hat{d}}, \hat{d})$-path. Finally, the message is routed from $\hat{d}$ to d along a shortest $(\hat{d}, d)$-path.

An example of a routing from s to d is shown schematically in Fig. 4. (Again, the figure is not drawn to scale, and not all regions and region names are shown.) Since s and d are in different level 4 regions, R_{11111} and R_{11112}, but in the same level 3 region, R_{1111}, l is 4. As level 1 region R_{11} is the first enveloping region to completely contain a shortest $(\hat{s}, \hat{d})$-path in G, shown dashed, $\bar{l}$ is 2. The message path is shown in bold.

The following theorem establishes the performance bound of the routing.

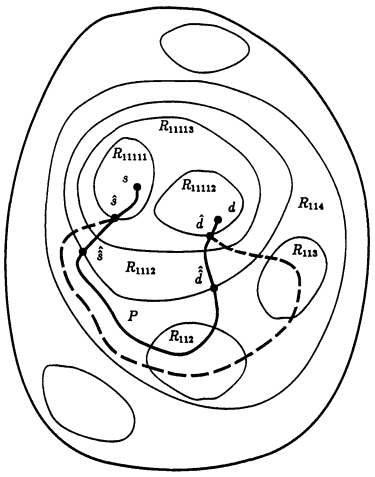


FIG. 4. *An example of a routing from s to d in Scheme* II. *The message path is shown bold.*

THEOREM 3.4. *For any planar graph, the performance bound of Scheme* II *is* 7.

Proof. Let s be any source and d any destination. If $\bar{l} = l$, then $\hat{s}$ and $\hat{\hat{d}}$ are just $\hat{s}$ and $\hat{d}$, respectively, and it follows that P is a shortest $(\hat{s}, \hat{d})$-path in G. Thus

$$\begin{aligned}\hat{\rho}(s,d) &= \rho(s,\hat{s}) + \rho(\hat{s},\hat{d}) + \rho(\hat{d},d) \\ &\leq \rho(s,\hat{s}) + \rho(\hat{s},s) + \rho(s,d) + \rho(d,\hat{d}) + \rho(\hat{d},d) \\ &\leq 3\rho(s,d), \text{ as } \rho(s,\hat{s}) + \rho(d,\hat{d}) \leq \rho(s,d) \quad \text{by Lemma 2.4.}\end{aligned}$$

Otherwise, $\bar{l} < l$ is the highest-numbered level for which a shortest $(\hat{s}, \hat{d})$-path in G is not contained in the enveloping level $\bar{l}$ region, but is contained in the enveloping level $\bar{l} - 1$ region. The path thus leaves the level $\bar{l}$ region for the first time and reenters it for the last time via two of its boundary nodes, b_1 and b_2, respectively. Thus $\rho(\hat{s},\hat{d}) \geq \rho(\hat{s},b_1) + \rho(\hat{d},b_2) \geq \rho(\hat{s},\hat{\hat{s}}) + \rho(\hat{d},\hat{\hat{d}})$, by our choice of $\hat{\hat{s}}$ and $\hat{\hat{d}}$ as the closest ancestors of $\hat{s}$ and $\hat{d}$ for level $\bar{l}$, respectively. Let $| P |$ be the length of P. Then $| P | \leq \rho(\hat{\hat{s}},\hat{s}) + \rho(\hat{s},\hat{d}) + \rho(\hat{d},\hat{\hat{d}}) \leq 2\rho(\hat{s},\hat{d})$. The length of the routing from $\hat{s}$ to $\hat{d}$ is then $\rho(\hat{s},\hat{\hat{s}}) + | P | + \rho(\hat{\hat{d}},\hat{d}) \leq 3\rho(\hat{s},\hat{d})$. Thus

$$\begin{aligned}\hat{\rho}(s,d) &\leq \rho(s,\hat{s}) + 3\rho(\hat{s},\hat{d}) + \rho(\hat{d},d) \\ &\leq \rho(s,\hat{s}) + 3(\rho(\hat{s},s) + \rho(s,d) + \rho(d,\hat{d})) + \rho(\hat{d},d) \\ &= 4(\rho(s,\hat{s}) + \rho(d,\hat{d})) + 3\rho(s,d) \\ &\leq 7\rho(s,d), \quad \text{by Lemma 2.4.}\end{aligned}$$

Thus $\hat{\rho}(s,d)/\rho(s,d) \leq 7$ for any nodes s and d, and the theorem follows. □

REFERENCES

[1] B. S. BAKER, *Approximation algorithms for NP-complete problems on planar graphs*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, Tucson, Arizona, October 1983, pp. 265–273.

[2] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, American Elsevier, New York, 1976.

[3] R. J. DUFFIN, *Topology of series-parallel networks*, J. Math. Appl., 10 (1965), pp. 303–318.

[4] G. N. FREDERICKSON, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[5] G. N. FREDERICKSON AND R. JANARDAN, *Designing networks with compact routing tables*, Algorithmica, 3 (1988), pp. 171–190.

[6] ———, *Space-efficient message routing in c-decomposable networks*, SIAM J. Comput., 19 (1990), to appear.

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

[8] L. KLEINROCK AND F. KAMOUN, *Hierarchical routing for large networks – performance evaluation and optimization*, Comput. Networks, ISDN Systems, 1 (1977), pp. 155–174.

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

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

[11] D. PELEG AND E. UPFAL, *A trade-off between space and efficiency for routing tables*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, May 1988, pp. 43–52.

[12] N. SANTORO AND R. KHATIB, *Labelling and implicit routing in networks*, Comput. J., 28 (1985), pp. 5–8.

[13] J. VAN LEEUWEN AND R. TAN, *Computer networks with compact routing tables*, in The Book of L, G. Rozenberg and A. Salomaa, eds., Springer-Verlag, Berlin, New York, 1986, pp. 259–273.

[14] ———, *Interval routing*, Comput. J., 30 (1987), pp. 298–307.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 859-881, October 1989

POLYNOMIAL TIME ALGORITHMS FOR FINDING INTEGER RELATIONS AMONG REAL NUMBERS*

J. HASTAD†, B. JUST‡¶, J. C. LAGARIAS§, AND C. P. SCHNORR‡

Abstract. This paper considers variants and generalizations of the following computational problem. Given a real input $\mathbf{x} \in \mathbb{R}^n$, find a small integer relation $\mathbf{m}$ for x that is a nonzero vector $\mathbf{m} \in \mathbb{Z}^n$ orthogonal to $\mathbf{x}$, or prove that no integer relation $\mathbf{m}$ exists with $\|\mathbf{m}\| \leqq 2^k$. An algorithm is presented that solves this problem in $O(n^3(k+n))$ arithmetic operations over real numbers. The algorithm is a variation of the multidimensional Euclidean algorithm proposed by Ferguson and Forcade [*Bull. Amer. Math. Soc.*, 1 (1979), pp. 912-914] and Bergman [*Notes on Ferguson and Forcade's Generalized Euclidean Algorithm*, University of California, Berkeley, CA, 1980]. A connection between such multidimensional Euclidean algorithms and the Lattice Basis Reduction Algorithm of Lenstra, Lenstra Jr., and Lovász [*Math. Ann.*, 21 (1982), pp. 515-534] is shown. Polynomial time solutions are also established for finding linearly independent sets of small integer relations and for finding small simultaneous integer relations for several real vectors, using real input vectors and counting arithmetic operations over real numbers at unit cost. For integer input vectors $\mathbf{x}$ a different algorithm is given for finding integer relations (that always exist) that uses at most $O(n^3 \log \|x\|)$ arithmetic operations on $O(n + \log \|\mathbf{x}\|)$ bit integers.

Key words. multidimensional continued fraction algorithm, lattice basis reduction, integer relations, diophantine approximation, generalized Euclidean algorithms

AMS(MOS) subject classifications. 11J71, 11Y16, 68Q25

1. Introduction.

Given a real vector $\mathbf{x} = (x_1, \cdots, x_n) \in \mathbb{R}^n$, an *integer relation* for $\mathbf{x}$ is a nonzero vector $\mathbf{m} = (m_1, \cdots, m_n) \in \mathbb{Z}^n$ such that $\langle \mathbf{m}, \mathbf{x} \rangle = \sum_{i=1}^n m_i x_i = 0$. This paper studies the following computational problem. Given a real vector $\mathbf{x} \in \mathbb{R}^n$, either find a small integer relation $\mathbf{m}$ for $\mathbf{x}$ or prove that no small integer relation exists.

The problem of finding integer relations for two numbers (x_1, x_2) can be solved by applying the Euclidean algorithm to x_1, x_2, or, equivalently, by computing the ordinary continued fraction expansion of the real number x_1/x_2. The problem of finding good algorithms for $n \geqq 3$ has been studied under the names *generalized Euclidean algorithm* and *multidimensional continued fraction algorithm.* Quite a few of these algorithms have been proposed, many of which are surveyed in Brentjes (1981) and Bernstein (1971).

In geometric terms the task is to approximate a line $\mathbf{x}\mathbb{R} \subset \mathbb{R}^n$, consisting of scalar multiples of $\mathbf{x}$, by a sequence of integer lattice bases of $\mathbb{Z}^n$. The convergence of a sequence of lattice bases $\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}$ for $\nu = 1, 2, \cdots$ to the line $\mathbf{x}\mathbb{R}$ has the following consequences. The $(n-1)$-vector $(b_{i2}^{(\nu)}/b_{i1}^{(\nu)}, \cdots, b_{in}^{(\nu)}/b_{i1}^{(\nu)})$ associated with $\mathbf{b}_i^{(\nu)} = (b_{i1}^{(\nu)}, \cdots, b_{in}^{(\nu)})$ is a good simultaneous Diophantine approximation to the vector $(x_2/x_1, \cdots, x_n/x_1)$ associated with $\mathbf{x} = (x_1, \cdots, x_n)$. The lattice basis $\mathbf{c}_1^{(\nu)}, \cdots, \mathbf{c}_n^{(\nu)}$ that is *dual* to $\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}$ consists of the row vectors of the inverse of the matrix

* Received by the editors June 2, 1986; accepted for publication (in revised form) December 5, 1988. A preliminary version of this paper was presented at the 1986 Symposium of Theoretical Aspects of Computer Science in Paris, France.

† Royal Institute of Technology, Stockholm, Sweden. This research was performed while the first author visited AT&T Bell Laboratories, Murray Hill, New Jersey, 07974, and Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This author was supported by an IBM fellowship.

‡ Universität Frankfurt, Fachbereich Mathematik und Informatik, 6000 Frankfurt, Federal Republic of Germany.

§ AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

¶ The maiden name of B. Just is B. Helfrich.

$[\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}]$ with column vectors $\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}$, i.e., $[\mathbf{c}_1^{(\nu)}, \cdots, \mathbf{c}_n^{(\nu)}]^\top = [\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}]^{-1}$. If the basis $\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}$ converges (in various ways) to $\mathbf{x}\mathbb{R}$ then the vectors $\mathbf{c}_i^{(\nu)}$ for $1 \leqq i \leqq n$ converge to $(\mathbf{x}\mathbb{R})^\perp$. If in particular $\mathbf{c}_i^{(\nu)}$ is orthogonal to $\mathbf{x}$ then $\mathbf{c}_i^{(\nu)}$ is an *integer relation* for $\mathbf{x}$; if such a vector $\mathbf{c}_i^{(\nu)}$ occurs it is said that an integer relation has been detected.

Jacobi (1868) proposed a continued fraction algorithm for $n = 3$ and Perron (1907) extended it to general n. The Jacobi–Perron algorithm generates a sequence of integer lattice bases $\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}$ of $\mathbb{Z}^n$ that *weakly converges* to $\mathbf{x}\mathbb{R}$, i.e., the angles between $\mathbf{b}_i^{(\nu)}$ and $\mathbf{x}$ converge to 0, which means $\lim_\nu (\mathbf{b}_i^{(\nu)}/\|\mathbf{b}_i^{(\nu)}\| - \mathbf{x}/\|\mathbf{x}\|) = \mathbf{0}$ for $1 \leqq i \leqq n$. Ferguson and Forcade (1979) presented the first continued fraction algorithm for arbitrary n that is guaranteed to detect integer relations when they exist. If no integer relations exist this algorithm is *strongly convergent*, i.e., the distances of $\mathbf{b}_i^{(\nu)}$ to $\mathbf{x}\mathbb{R}$ converge to 0, which means $\lim_\nu (\mathbf{b}_i^{(\nu)}\|\mathbf{x}\| - \mathbf{x}\|\mathbf{b}_i^{(\nu)}\|) = 0$ for $1 \leqq i \leqq n$. Ferguson and Forcade's algorithm is inductive. The n-dimensional version of the algorithm repeatedly uses lower-dimensional versions of the algorithm as a subroutine (see Ferguson and Forcade (1982), Ferguson (1986).) A noninductive algorithm that incorporates the basic Ferguson–Forcade ideas but which is different in detail was developed by Bergman (1980). Ferguson (1987) presents a similar noninductive algorithm and proves an exponential running time bound for it. Bergman's algorithm is surprisingly similar to the Lovász algorithm for lattice basis reduction. Both algorithms perform the same type of basis transformations, reduction in size and exchange steps according to similar but slightly different exchange rules.

This paper studies the problem of finding integer relations via generalized Euclidean algorithms. It considers the following computational problem, the *integer relation problem*. Given a real vector $\mathbf{x} \in \mathbb{R}^n$ and a bound 2^k, either find an integer relation $\mathbf{m}$ for $\mathbf{x}$ with $\|\mathbf{m}\| \leqq 2^{n+k}$ or prove there is no integer relation $\mathbf{m}$ with $\|\mathbf{m}\| < 2^k$. A gap between the upper bounds 2^{n+k} and 2^k is unavoidable because efficient lattice basis reduction algorithms only find a nearly shortest lattice vector. We distinguish two versions of this problem: one in which the input vector $\mathbf{x}$ consists of arbitary real numbers, and the other in which the input vector is an integer vector. The algorithms for the real input version use the following arithmetic operations on real numbers at unit cost: addition, subtraction, multiplication, division, comparison of two numbers ($<$), and the nearest integer function ($\lceil \rfloor$). On the other hand, the algorithms for the integer input version are studied using the bit complexity model in which we count bit operations. We present algorithms for both the real and the integer versions of the problem, and we prove running time bounds that are polynomial in n, k, and the bit length of the integer inputs. The algorithms for the real input version of the problem are similar to the algorithms of Bergman (1980) and Ferguson (1987). For the integer input version of the problem we present an algorithm that is close to the Lovász basis reduction algorithm.

The problem dual to the integer relation problem is the problem of finding good simultaneous Diophantine approximations. Algorithms for the integer relation problem do not necessarily find good Diophantine approximations. Just (1987) has established a class of continued fraction algorithms that use both the Bergman and Lovász exchange rule and that find reasonably good simultaneous Diophantine approximations in all dimensions.

Now we describe the contents of the paper in more detail. Section 2 presents a version of the generalized Euclidean algorithm of Bergman (1980) and Ferguson (1987) and the Lovász Lattice Basis Reduction Algorithm of Lenstra, Lenstra, Jr., and Lovász

(1982). These algorithms are presented in a form that illustrates their striking similarity; this was not apparent in the original papers. We call our version of the "Bergman, Ferguson, Forcade-type" algorithm the *Basic Integer Relation Algorithm.* This algorithm exhibits the main features of the polynomial time algorithms to solve the real input version of various integer relation problems presented in §§ 3–5. The Basic Integer Relation Algorithm and the Lovász Lattice Basis Reduction Algorithm use different rules for exchanging basis vectors. This difference is crucial to obtaining a polynomial running time bound for the real input version of the Basic Integer Relation Algorithm. The Lovász exchange rule does not find integer relations among real numbers in polynomial time.

Section 3 presents the *Small Integer Relation Algorithm* that is derived from the Basic Relation Algorithm by adding a suitable termination test. The Small Integer Relation Algorithm takes as input a real vector $\mathbf{x}=(x_1,\cdots,x_n)\neq\mathbf{0}$ and a positive integer k. We prove it has the following properties.

(1) It either finds an integer relation $\mathbf{m}$ for $\mathbf{x}$ with $\|\mathbf{m}\|^2\leqq 2^{n-2}\min\{\lambda(\mathbf{x})^2, 2^{2k}\}$ or proves $\lambda(\mathbf{x})\geqq 2^k$, where $\lambda(\mathbf{x})$ is the length of the shortest, nonzero integer relation for $\mathbf{x}$.

(2) It halts after at most $O(n^3(k+n))$ arithmetic operations on real numbers.

The arithmetic model of computation on real numbers, with the operations addition, subtraction, multiplication, division, comparison ($<$), and the nearest integer function ($\lceil\,\rfloor$) at unit cost, is not compatible with the usual Turing machine model since it allows infinite precision arithmetic at each step. In the Turing machine model we cannot prove the existence of integer relations for real numbers since this would require infinite precision arithmetic. However in the Turing machine model we can prove, by computations on rational numbers, the nonexistence of small integer relations for given real numbers $x_1,\cdots,x_n$. For this we apply the Small Integer Relation Algorithm to a rational vector $\bar{\mathbf{x}}$ that is sufficiently close to $\mathbf{x}=(x_1,\cdots,x_n)$. We cannot say beforehand how close $\bar{\mathbf{x}}$ has to be to $\mathbf{x}$ but Theorem 3.5 defines the term "sufficiently close" a posteriori.

Section 4 describes an algorithm used to find several small linearly independent integer relations if they exist. The resulting algorithm, the *Several Relations Algorithm,* takes as input a vector $\mathbf{x}\neq 0$ in $\mathbb{R}^n$, and positive integers k and r satisfying $1\leqq r\leqq n-1$. Its output is either a set of r linearly independent integer relations $\{\mathbf{m}_i: 1\leqq i\leqq r\}$ with all $\|\mathbf{m}_i\|\leqq 2^{n+k}$ or a proof that there do not exist r linearly independent integer relations with all $\|\mathbf{m}_i\|\leqq 2^k$. Its running time is $O(n^3(k+n))$ arithmetic operations. Let $L_{\mathbf{x}}\subset\mathbb{Z}^n$ be the lattice of integer relations for $\mathbf{x}$. The Several Relations Algorithm can be used to approximate the successive minima of $L_{\mathbf{x}}$. It can also be used to find a basis of a sublattice L of the lattice $L_{\mathbf{x}}$ that contains all shortest vectors in $L_{\mathbf{x}}$.

Section 5 presents an algorithm that finds simultaneous integer relations for linearly independent vectors $\mathbf{x}_1,\cdots,\mathbf{x}_q\in\mathbb{R}^n$. The *Simultaneous Relations Algorithm* incorporates all the previous algorithms as special cases. When given as input $\mathbf{x}_1,\cdots,\mathbf{x}_q\in\mathbb{R}^n$ and r, $k\in\mathbb{N}$ this algorithm halts after at most $O(n^3(k+n))$ arithmetic operations with real numbers and either finds r independent simultaneous integer relations $\mathbf{m}$ for $\mathbf{x}_1,\cdots,\mathbf{x}_q$ or proves there does not exist such a set of relations $\mathbf{m}$ with $\|\mathbf{m}\|<2^k$. This algorithm can be used to find r independent integer dependencies among a set of real vectors $\mathbf{y}_1,\cdots,\mathbf{y}_t\in\mathbb{R}^n$.

Section 6 studies integer relation algorithms in the bit complexity model of computation. In this case the input is a vector $\mathbf{x}$ in $\mathbb{Z}^n$, and integer relations always exist. We describe an algorithm based on ideas from the Lattice Basis Reduction

Algorithm. This algorithm finds a basis for the $(n-1)$-dimensional lattice $L_{\mathbf{x}}$ of integer relations for $\mathbf{x}$. It essentially applies the Lovász algorithm to the linearly dependent vectors $\mathbf{b}_0 = \mathbf{x}, \mathbf{b}_1, \cdots, \mathbf{b}_n$, where $\mathbf{b}_1, \cdots, \mathbf{b}_n$ is the standard basis of $\mathbb{Z}^n$. This algorithm finds short basis vectors for the lattice $L_{\mathbf{x}}$, and it terminates after at most $O(n^3 \log \|\mathbf{x}\|)$ arithmetic steps using $O(n + \log \|\mathbf{x}\|)$-bit integers. The performance of this algorithm should be compared with that of the efficient reduction algorithms by Schönhage (1984) and Schnorr (1986, 1988).

The results of this paper were obtained independently by Hastad and Lagarias and by Just and Schnorr. Just and Schnorr rediscovered the Basic Integer Relation Algorithm independently of Bergman, Ferguson, and Forcade as a modification of the Lovász algorithm.

2. The Basic Integer Relation Algorithm and the Lovász Lattice Basis Reduction Algorithm. We will use the following notation throughout the paper. Let $\mathbb{R}^n$ be the n-dimensional real vector space, $n > 1$ with inner product $\langle , \rangle$ and let $\|\mathbf{y}\| = \langle \mathbf{y}, \mathbf{y} \rangle^{1/2}$ be the length of the vector $\mathbf{y} \in \mathbb{R}^n$. All vectors are column vectors unless otherwise specified. For a linear subspace $E \subset \mathbb{R}^n$ we let $E^{\perp} \subset \mathbb{R}^n$ be the orthogonal complement of E, i.e., the subspace consisting of all vectors that are orthogonal to E. For $\mathbf{b}_1, \cdots, \mathbf{b}_r \in \mathbb{R}^n$ let $\langle \mathbf{b}_1, \cdots, \mathbf{b}_r \rangle$ be the additive closure of $\mathbf{b}_1, \cdots, \mathbf{b}_r$, i.e., the set of all vectors $\sum_{i=1}^{r} m_i \mathbf{b}_i$ with $m_i \in \mathbb{Z}$. The $n \times r$ matrix with column vectors $\mathbf{b}_1, \cdots, \mathbf{b}_r \in \mathbb{R}^n$ is denoted $[\mathbf{b}_1, \cdots, \mathbf{b}_r]$. Let span $(\mathbf{b}_1, \cdots, \mathbf{b}_r)$ be the linear space generated by the vectors $\mathbf{b}_1, \cdots, \mathbf{b}_r$. The transpose of matrix A is $A^{\top}$.

A lattice $L \subset \mathbb{R}^n$ is a discrete, additive subgroup of $\mathbb{R}^n$. Every lattice is the additive closure of a set $\mathbf{b}_1, \cdots, \mathbf{b}_r$ of linearly independent vectors in $\mathbb{R}^n$, i.e., $L = \langle \mathbf{b}_1, \cdots, \mathbf{b}_r \rangle$. The vectors $\mathbf{b}_1, \cdots, \mathbf{b}_r$ are called a *basis* of L, and r is the *rank* of the lattice L. The *determinant* of L is the volume of the r-dimensional parallelepiped generated by the basis $\mathbf{b}_1, \cdots, \mathbf{b}_r$. The rank and the determinant do not depend on the choice of basis. The ith *successive minimum* $\lambda_i(L)$ of lattice L is the smallest radius of a ball with center $\mathbf{0}$ that contains i linearly independent lattice vectors.

All algorithms of this paper that take real numbers for input will use the following arithmetic operations on real numbers at unit cost: addition, subtraction, multiplication, division, comparison ($<$), and the nearest integer function ($\lceil \rfloor$). We call this the *arithmetic model of computation.* (It differs from the arithmetic model of computation used in Frank and Tardös (1985) because it includes $\lceil \rfloor$ as a basic arithmetic operation.)

An *integer relation* $\mathbf{m} \in \mathbb{Z}^n$ for $\mathbf{x}$ is a nonzero vector $\mathbf{m} \in \mathbb{Z}^n$ satisfying $\langle \mathbf{x}, \mathbf{m} \rangle = 0$. We associate with a nonzero vector $\mathbf{x} \in \mathbb{R}^n$ the lattice $L_{\mathbf{x}} \subset \mathbb{Z}^n$ of all integer relations for $\mathbf{x}$ together with $\mathbf{0}$, i.e.,

$$L_{\mathbf{x}} = \{\mathbf{m} \in \mathbb{Z}^n : \langle \mathbf{x}, \mathbf{m} \rangle = 0\}.$$

For general real vectors $\mathbf{x}$ the lattice $L_{\mathbf{x}}$ may have rank from 0 to $n-1$, while for rational vectors its rank is always $n-1$. For real vectors $\mathbf{x}$ the rank of $L_{\mathbf{x}}$ cannot be computed in the arithmetic model of computation. Babai, Just, and Meyer auf der Heide (1988) have shown that it cannot even be decided in the arithmetic model of computation whether there exists an integer relation for $\mathbf{x}$, i.e., whether rank $(L_{\mathbf{x}}) \geqq 1$. Let $\lambda(\mathbf{x}) = \lambda_1(\mathbf{x}) \leqq \lambda_2(\mathbf{x}) \leqq \cdots \leqq \lambda_r(\mathbf{x})$, with $r = \text{rank}\,(L_{\mathbf{x}})$, be the successive minima of the lattice $L_{\mathbf{x}}$. Then $\lambda(\mathbf{x})$ is the length of the shortest integer relation for $\mathbf{x}$ provided that some integer relation exists. If there is no integer relation for $\mathbf{x}$ we let $\lambda(\mathbf{x}) = \infty$.

With a sequence of vectors $\mathbf{b}_1, \cdots, \mathbf{b}_n \in \mathbb{R}^n$ and the fixed vector $\mathbf{b}_0 = \mathbf{x} \in \mathbb{R}^n$, we associate the orthogonal system $\mathbf{b}_0^*, \cdots, \mathbf{b}_n^*$ where $\mathbf{b}_i^*$ is the component of $\mathbf{b}_i$ that is orthogonal to $\mathbf{b}_1, \cdots, \mathbf{b}_{i-1}$ and $\mathbf{b}_0$. The vectors $\mathbf{b}_0^*, \cdots, \mathbf{b}_n^*$ can be computed by the

process of Gram-Schmidt orthogonalization using $O(n^3)$ arithmetic operations:

$$\mathbf{b}_0^* = \mathbf{x},$$

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=0}^{i-1} \mu_{i,j}\mathbf{b}_j^*, \qquad i = 1, \cdots, n,$$

where $\mu_{i,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle \|\mathbf{b}_j^*\|^{-2}$ if $\mathbf{b}_j^* \neq 0$, and $\mu_{i,j} = 0$ if $\mathbf{b}_j^* = 0$. This process gives the usual Gram-Schmidt orthogonalization for $\mathbf{b}_1, \cdots, \mathbf{b}_n$ when $\mathbf{b}_0 = \mathbf{x} = \mathbf{0}$. We denote by $\lceil r \rfloor$ the integer nearest to $r \in \mathbb{R}$, with $\lceil r \rfloor = r - \frac{1}{2}$ if r is a half-integer.

Both the Basic Integer Relation Algorithm and the Lovász Lattice Basis Reduction Algorithm perform a sequence of elementary basis exchange operations on a current ordered basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of a given lattice. These consist of two types of steps:

(1) *Exchange steps.* Interchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ for some i.

(2) *Size-reduction step.* Replace $\mathbf{b}_i$ with $\mathbf{b}_i - l\mathbf{b}_j$ where $l \in \mathbb{Z}$ for some i and j with $1 \leqq j < i$.

Both types of steps produce a new basis matrix B^{new} with $B^{\text{new}} = BU$ for some matrix $U \in \mathrm{GL}(n, \mathbb{Z})$, where $B = [\mathbf{b}_1, \cdots, \mathbf{b}_n]$.

The purpose of a size-reduction step is to make the Gram-Schmidt quantity $\mu_{i,j}^{\text{new}}$ in the new basis satisfy $|\mu_{i,j}^{\text{new}}| \leqq \frac{1}{2}$. This uniquely determines the integer l in the size-reduction step by $l = \lceil \mu_{i,j}^{\text{old}} \rfloor$. A basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of a lattice is *size-reduced* if all $|\mu_{i,j}| \leqq \frac{1}{2}$ in its Gram-Schmidt orthogonalization. Given any basis we can obtain a size-reduced basis having the same Gram-Schmidt orthogonal system $\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*$ by applying a sequence of $n(n-1)/2$ size-reduction steps. These steps have to be applied in a suitable order that does not change the $|\mu_{i,j}| \leqq \frac{1}{2}$ produced in previous steps.

The Basic Integer Relation Algorithm is a variant of the algorithms of Bergman (1980) and Ferguson (1987), which are themselves based on ideas of the Ferguson-Forcade Algorithm (1979), (1982).

BASIC INTEGER RELATION ALGORITHM. (On input $\mathbf{x} \in \mathbb{R}^n$ this algorithm produces infinite sequences $M^{(t)}$, $P^{(t)}$ of matrices in $\mathrm{GL}(n, \mathbb{Z})$. The column vectors $\mathbf{b}_1^{(t)}, \cdots, \mathbf{b}_n^{(t)}$ of $P^{(t)}$ form a basis of $\mathbb{Z}^n$ and these bases approach span $(\mathbf{x})$ as t increases. The row vectors of $M^{(t)} = (P^{(t)})^{-1}$ are the dual basis, and these converge to span $(\mathbf{x})^{\perp}$.)

1. *Initiation.* For the vectors $\mathbf{b}_0 = \mathbf{x}$ and standard basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of $\mathbb{Z}^n$ compute the Gram-Schmidt quantities $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|^2$ for $0 \leqq j$, $i \leqq n$. Set iteration count $t := 0$.

2. *Exchange step.* $t := t+1$. Choose for $1 \leqq i \leqq n$ that i that maximizes $2^i \|\mathbf{b}_i^*\|^2$. Size-reduce $\mathbf{b}_{i+1}$ with respect to $\mathbf{b}_i$ by setting $\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$. Exchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$. Now update the Gram-Schmidt data $\|\mathbf{b}_r^*\|^2$, $\mu_{r,j}$, $\mu_{j,r}$ for $r = i$, $i+1$ and $1 \leqq j \leqq n$. Output $P^{(t)} = [\mathbf{b}_1, \cdots, \mathbf{b}_n]$ and $M^{(t)} = [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$. Go to 2.

For completeness we include formulae for updating the Gram-Schmidt numbers $\mu_{i,j}$, $\|\mathbf{b}_i^*\|^2$. For the transformation $\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$ we obtain the new $\mu_{i+1,j}$ by setting

$$\mu_{i+1,j} := \mu_{i+1,j} - \lceil \mu_{i+1,i} \rfloor \mu_{i,j} \quad \text{for } j = 1, \cdots, i,$$

and the other Gram-Schmidt quantities do not change. Explicit formulae for updating an exchange step $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ with $\|\mathbf{b}_i^*\|$, $\|\mathbf{b}_{i+1}^*\| \neq 0$ are given by Lenstra, Lenstra Jr., and Lovász (1982). We extend their formulae to include the case $\|\mathbf{b}_{i+1}^*\| = 0$.

Updating $\|\mathbf{b}_\nu^*\|$, $\mu_{\nu,j}$, $\mu_{j,\nu}$ *for* $\nu = i$, $i+1$ *and* $j = 1, \cdots, n$ *in case of an exchange* $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$.

$\mu := \mu_{i+1,i}$;

$\|\mathbf{b}_i^*\|_{\text{new}}^2 := \|\mathbf{b}_{i+1}^*\|^2 + \mu^2 \|\mathbf{b}_i^*\|^2$;

if $\|\mathbf{b}_i^*\|_{\text{new}} \neq 0$ *then* $(\|\mathbf{b}_{i+1}^*\|^2 := \|\mathbf{b}_i^*\|^2 \|\mathbf{b}_{i+1}^*\|^2 / \|\mathbf{b}_i^*\|_{\text{new}}^2,\ \mu_{i+1,i} := \mu \|\mathbf{b}_i^*\| / \|\mathbf{b}_i^*\|_{\text{new}}^2)$
else $(\|\mathbf{b}_{i+1}^*\|^2 := \|\mathbf{b}_i^*\|^2,\ \mu_{i+1,i} := 0)$.
$\|\mathbf{b}_i^*\|^2 := \|\mathbf{b}_i^*\|_{\text{new}}^2$;

$$\begin{pmatrix} \mu_{i,j} \\ \mu_{i+1,j} \end{pmatrix} := \begin{pmatrix} \mu_{i+1,j} \\ \mu_{i,j} \end{pmatrix} \quad \text{for } j = 1, \cdots, i-1.$$

$$\begin{pmatrix} \mu_{j,i} \\ \mu_{j,i+1} \end{pmatrix} := \begin{pmatrix} 1 & \mu_{i+1,i} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -\mu \end{pmatrix} \begin{pmatrix} \mu_{j,i} \\ \mu_{j,i+1} \end{pmatrix} \quad \text{for } j = i+2, \cdots, n.$$

We remark that this algorithm keeps $\mathbf{b}_0 = \mathbf{x}$ fixed throughout; all the operations are on the remaining $n \times n$ matrix of basis vectors $\mathbf{b}_1, \cdots, \mathbf{b}_n$. Theorem 3.2 below proves that this algorithm eventually detects an integer relation if one exists. If no integer relation exists the algorithm does not stop and the sequence of output bases $[\mathbf{b}_1^{(\nu)}, \cdots, \mathbf{b}_n^{(\nu)}]$ converges strongly to $\mathbf{x}\mathbb{R}$, i.e., $\lim_\nu (\mathbf{b}_i^{(\nu)} \|\mathbf{x}\| - \mathbf{x} \|\mathbf{b}_i^{(\nu)}\|) = \mathbf{0}$ for $1, \cdots, n$. The strong convergence has been proved by Ferguson and Forcade (1979). It follows from $\lim_\nu \|\mathbf{b}_i^{(\nu)*}\| = 0$ for $i = 1, \cdots, n$.

The Basic Integer Relation Algorithm coincides essentially with the algorithms of Bergman (1980) and Ferguson (1987). The particular exchange rule used in the Basic Integer Relation Algorithm was proposed by Bergman (1980), so we call it the *Bergman exchange rule.* We have however used a somewhat different language than Ferguson (1987) to describe the algorithm, and our algorithm also differs from his in that in the exchange step it only size-reduces $\mu_{i+1,i}$ and not $\mu_{i+1,1}, \cdots, \mu_{i+1,i-1}$. The size-reduction of $\mu_{i+1,1}, \cdots, \mu_{i+1,i-1}$ is not necessary for finding integer relations in the real number model of computation. This size-reduction keeps the entries in the matrix $M^{(t)}$ small, which is important in the bit complexity model.

It is interesting to compare the Basic Integer Relation Algorithm with the Lovász Lattice Basis Reduction Algorithm that appears in Lentra, Lenstra Jr., and Lovász (1982). We outline a variant of the latter algorithm that takes for input an arbitrary sequence $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of vectors in $\mathbb{R}^n$, possibly linearly dependent. In this description $\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*$ is the ordinary Gram-Schmidt orthogonalization system arising when $\mathbf{b}_0 = \mathbf{x} = \mathbf{0}$.

BASIC LOVÁSZ ALGORITHM.

1. *Initiation.* Input $\mathbf{b}_1, \cdots, \mathbf{b}_n$ in $\mathbb{R}^n$ and compute the Gram-Schmidt quantities $\mu_{i,j}$ and $\|\mathbf{b}_i^*\|$ for $1 \leqq j,\ i \leqq n$.
2. *Termination condition.* If $\|\mathbf{b}_i^*\|^2 \leqq 2\|\mathbf{b}_{i+1}^*\|^2$ for $1 \leqq i \leqq n-1$, then size-reduce the basis, output it, and halt.
3. *Exchange step.* Choose the smallest $i < n$ with $\|\mathbf{b}_i^*\|^2 > 2\|\mathbf{b}_{i+1}^*\|^2$. Size reduce $\mathbf{b}_{i+1}$ with respect to $\mathbf{b}_i$, setting $\mathbf{b}_{i+1} = \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$. Exchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$. Update the Gram-Schmidt quantities. Go to 2.

This algorithm differs from the usual Lattice Basis Reduction Algorithm by omitting extra size-reduction steps used to keep integers small, which are not relevant in the real-number model of computation. It also differs in that we have replaced in the Lovász algorithm the reduction condition $\frac{3}{4}\|\mathbf{b}_i^*\|^2 \leqq \|\mathbf{b}_{i+1}^*\|^2 + \mu_{i+1,i}^2 \|\mathbf{b}_i^*\|^2$ by the essentially equivalent *Siegel reduction condition* $\|\mathbf{b}_i^*\|^2 \leqq 2\|\mathbf{b}_{i+1}^*\|^2$. If the input vectors are linearly independent, then this algorithm eventually halts and produces a basis reduced in the Lovász sense. (The arguments in Lenstra, Lenstra Jr., and Lovász (1982) are easily adapted to prove this.) If the input vectors are linearly dependent the algorithm may never halt.

A great similarity in the form of the Basic Integer Relation Algorithm and the Basic Lovász Algorithm is apparent. Both algorithms have an exchange rule that

exchanges $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ where $\|\mathbf{b}_i^*\|^2 > 2\|\mathbf{b}_{i+1}^*\|^2$: the Lovász algorithm chooses the smallest such i while the Basic Integer Relation Algorithm chooses an i that maximizes $\|\mathbf{b}_i^*\|^2 2^i$. The subtle difference in the form of the Bergman exchange rule relative to the Lovász exchange rule is critical in obtaining a polynomial running time bound for the Basic Integer Relation Algorithm. If the Lovász exchange rule is inserted in the Basic Integer Relation Algorithm, the resulting algorithm can be proved to detect an integer relation if one exists, but in the real number model the number of steps it takes to find a relation of size less than or equal to 2^k cannot be bounded in terms of n and k alone.

We can also use the Lovász algorithm to eliminate integer dependencies from a generator system $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of a lattice $L = \langle \mathbf{b}_1, \cdots, \mathbf{b}_n \rangle$. Because L is discrete all sufficiently small vectors in L must be $\mathbf{0}$. We have the following theorem.

THEOREM 2.1. *Suppose $\bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_s \in \mathbb{R}^n$ is a set of generators for the lattice $L = \langle \bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_s \rangle$ of rank r. Then the Lovász Algorithm transforms the input vectors $\bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_s$ into vectors $\mathbf{b}_1, \cdots, \mathbf{b}_s$ such that $\mathbf{b}_1 = \cdots = \mathbf{b}_{s-r} = \mathbf{0}$ and $\mathbf{b}_{s-r+1}, \cdots, \mathbf{b}_s$ form a basis of the lattice L. This basis has the properties*

$$\|\mathbf{b}_i^*\|^2 \leqq 2\|\mathbf{b}_{i+1}^*\|^2 \quad \text{for } s-r+1 \leqq i \leqq s-1,$$
$$|\mu_{i,j}| \leqq \tfrac{1}{2} \quad \text{for } s-r+1 \leqq j < i \leqq s.$$

For integer input vectors with $B = \max_i \|\bar{\mathbf{b}}_i\|$ this algorithm terminates after at most $O(n^4 \log B)$ arithmetic steps on $O(n \log B)$-integers.

The proof is a straightforward extension of the analysis of the Lovász algorithm (see Lenstra, Lenstra, Jr., and Lovász (1982)) and is left to the reader. Theorem 2.1 asserts that the Lovász algorithm finds integer dependencies between the generators of a lattice whenever an integer dependency exists. On the other hand, if the input vectors $\bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_s$ do not generate a lattice then the Lovász algorithm may fail to find integer dependencies. This occurs, e.g., if $\bar{\mathbf{b}}_2 = \alpha \bar{\mathbf{b}}_1$ for some irrational α and $\bar{\mathbf{b}}_1 = \bar{\mathbf{b}}_3$. Then the Lovász algorithm successively reduces and exchanges $\bar{\mathbf{b}}_1, \bar{\mathbf{b}}_2$ an infinite number of times, so that the vector $\bar{\mathbf{b}}_1$ converges to $\mathbf{0}$ but it does not achieve $\bar{\mathbf{b}}_1 = \mathbf{0}$. It never detects the integer dependency $\mathbf{b}_1 = \mathbf{b}_3$. We present an algorithm to find integer dependencies in § 5.

3. Finding integer relations. The Basic Integer Relation Algorithm can be adapted to detect small integer relations and also to rule out the existence of small integer relations by adding a suitable termination test to the algorithm. We call the resulting algorithm the Small Integer Relation Algorithm.

SMALL INTEGER RELATION ALGORITHM. (On input $\mathbf{x} \in \mathbb{R}^n$ and $k \in \mathbb{N}$ this algorithm either finds an integer relation $\mathbf{c}_n$ for $\mathbf{x}$ satisfying $\|\mathbf{c}_n\|^2 \leqq 2^{n-2}\lambda(\mathbf{x})^2$ or it proves $\lambda(\mathbf{x}) \geqq 2^k$.)

1. *Initiation.* For the vectors $\mathbf{b}_0 = x$ and standard basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of $\mathbb{Z}^n$ compute the Gram-Schmidt quantities μ_{ij} and $\|\mathbf{b}_i^*\|^2$ for $0 \leqq i, j \leqq n$.
2. *Termination test.* If $\|\mathbf{b}_n^*\| \neq 0$ then an integer relation is found. Compute the matrix $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top = [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$, output the integer relation $\mathbf{c}_n$, and stop. If $\|\mathbf{b}_i^*\| \leqq 2^{-k}$ for $1 \leqq i \leqq n$, then no small integer relation exists. Output "$\lambda(\mathbf{x}) \geqq 2^k$" and stop.
3. *Exchange step.* Choose for $1 \leqq i \leqq n$ that i that maximizes $2^i\|\mathbf{b}_i^*\|^2$. Size-reduce $\mathbf{b}_{i+1}$ with respect to $\mathbf{b}_i$ by setting $\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$. Exchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$. Now update $\|\mathbf{b}_\nu^*\|^2$, $\mu_{\nu,j}$, $\mu_{j,\nu}$ for $\nu = i, i+1$ and $1 \leqq j \leqq n$. Go to 2.

Instead of computing, on termination, the inverse matrix $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top = [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$, we can compute it incrementally. Initially $[\mathbf{c}_1, \cdots, \mathbf{c}_n]$ is the unit matrix. A reduction step $\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$ changes the vectors $\mathbf{c}_1, \cdots, \mathbf{c}_n$ as $\mathbf{c}_i := \mathbf{c}_i + \lceil \mu_{i+1,i} \rfloor \mathbf{c}_{i+1}$. For an exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ we have to exchange $\mathbf{c}_i$ and $\mathbf{c}_{i+1}$.

In analyzing this algorithm, we first prove that the termination test gives a correct answer when it applies. (We defer until later a proof that the termination condition will eventually hold for any input.)

PROPOSITION 3.1 (Correctness of Small Integer Relation Algorithm).

(1) *The output* $\mathbf{c}_n$ *in step* 2 *is an integer relation for* $\mathbf{x}$.

(2) $\lambda(\mathbf{x}) \geqq 1/\max_i \|\mathbf{b}_i^*\|$ *holds for every basis* $\mathbf{b}_1, \cdots, \mathbf{b}_n$ *of the lattice* $\mathbb{Z}^n$.

(3) *The output* $\mathbf{c}_n$ *satisfies* $\|\mathbf{c}_n\|^2 \leqq 2^{n-2} \min\{\lambda(\mathbf{x})^2, 2^{2k}\}$.

Proof. In the proof i ranges over $1 \leqq i \leqq n$ only.

(1) Let $\mathbf{b}_n^* \neq \mathbf{0}$; then $\mathbf{b}_i^* = \mathbf{0}$ holds for some $i < n$. The vectors $\mathbf{b}_1, \cdots, \mathbf{b}_i$ are linearly independent but linearly dependent mod $(\mathbf{x}\mathbb{R})$, and thus $\mathbf{x} \in \text{span}(\mathbf{b}_1, \cdots, \mathbf{b}_i)$. Since $\langle \mathbf{b}_j, \mathbf{c}_k \rangle = 0$ holds for $k > j$ this implies $\langle \mathbf{x}, \mathbf{c}_k \rangle = 0$ for $k > i$ and in particular $\langle \mathbf{x}, \mathbf{c}_n \rangle = 0$. The vector $\mathbf{c}_n$ is integral since it is part of the inverse of the unimodular matrix $[\mathbf{b}_1, \cdots, \mathbf{b}_n]$.

(2) Let $\mathbf{m}$ be any integer relation for $\mathbf{x}$. Since $\mathbf{m} \in (\mathbf{x}\mathbb{R})^{\perp} = \text{span}(\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*)$ there exists i with $\langle \mathbf{m}, \mathbf{b}_i^* \rangle \neq 0$. For the smallest such i we have $\langle \mathbf{m}, \mathbf{b}_i^* \rangle = \langle \mathbf{m}, \mathbf{b}_i \rangle \in \mathbb{Z}$, and hence $|\langle \mathbf{m}, \mathbf{b}_i^* \rangle| \geqq 1$, and thus $\|\mathbf{m}\| \geqq \|\mathbf{b}_i^*\|^{-1}$. This shows that the Small Integer Relation Algorithms correctly claims "$\lambda(\mathbf{x}) \geqq 2^k$" in step 2.

(3) The integer relation $\mathbf{c}_n$ found by the algorithm is determined by

$$[\mathbf{c}_1, \cdots, \mathbf{c}_n]^{\top} = [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$$

where $\mathbf{b}_1, \cdots, \mathbf{b}_n$ is the terminal basis of lattice $\mathbb{Z}^n$. Since both $\mathbf{b}_n^*$ and $\mathbf{c}_n$ are orthogonal to $\mathbf{x}, \mathbf{b}_1, \cdots, \mathbf{b}_{n-1}$ we have $\text{span}(\mathbf{b}_n^*) = \text{span}(\mathbf{c}_n)$, and it follows from $\langle \mathbf{b}_n, \mathbf{c}_n \rangle = 1$ that

$$\mathbf{c}_n = \pm \mathbf{b}_n^* \|\mathbf{b}_n^*\|^{-2}, \qquad \|\mathbf{c}_n\| = \|\mathbf{b}_n^*\|^{-1}. \tag{3.1}$$

Let $\bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_n$ be the basis of lattice $\mathbb{Z}^n$ before the last exchange $\mathbf{b}_n \leftrightarrow \mathbf{b}_{n-1}$. (If there is no such exchange then $\|\mathbf{c}_n\| = 1$ and the claim holds.) We have $\max_i \|\bar{\mathbf{b}}_i^*\|^2 2^i = \|\bar{\mathbf{b}}_{n-1}^*\|^2 2^{n-1}$, which implies

$$\|\bar{\mathbf{b}}_i^*\|^2 \leqq 2^{n-i-1} \|\bar{\mathbf{b}}_{n-1}^*\|^2 = 2^{n-i-1} \|\mathbf{b}_n^*\|^2 \quad \text{for } i = 1, \cdots, n-1.$$

From this we conclude that

$$\|\mathbf{c}_n\|^2 = \|\mathbf{b}_n^*\|^{-2} \leqq 2^{n-2} / \max_i \|\bar{\mathbf{b}}_i^*\|^2.$$

From (2) and since the algorithm did not terminate previously we see

$$1/\max_i \|\bar{\mathbf{b}}_i^*\|^2 \leqq \min\{\lambda(\mathbf{x})^2, 2^{2k}\}.$$

Thus the claim follows from the two latter inequalities. □

This proposition reveals the underlying motivation for Bergman's exchange rule: it is designed to prove there is no short integer relation as quickly as possible. Since we have $\lambda(\mathbf{x}) \geqq 1/\max_i \|\mathbf{b}_i^*\|$ it is reasonable to minimize $\max_i \|\mathbf{b}_i^*\|$. Bergman's exchange rule strives to minimize $\max_i 2^i \|\mathbf{b}_i^*\|^2$, where the extra powers of two are included so that we can make a profitable exchange.

Now we prove a running time bound for the Small Integer Relation Algorithm.

THEOREM 3.2. *The Small Integer Relation Algorithm halts after at most* $O(n^3(k+n))$ *arithmetic steps on real numbers. It either finds an integer relation* $\mathbf{c}_n$ *for* $\mathbf{x}$ *satisfying* $\|\mathbf{c}_n\|^2 \leqq 2^{n-2} \min\{\lambda(\mathbf{x})^2, 2^{2k}\}$ *or else it proves* $\lambda(\mathbf{x}) \geqq 2^k$.

Proof. We measure the progress of the algorithm using the quantity

$$D = \prod_{i=1}^{n-1} \alpha(\mathbf{b}_i^*)^{n-i} \quad \text{where } \alpha(\mathbf{b}_i^*) = \max\{\|\mathbf{b}_i^*\|^2 2^n, 2^{-2k}\}.$$

LEMMA 3.3. *Every exchange step of the Small Integer Relation Algorithm achieves*

$$D_{\text{new}} \leqq \tfrac{3}{4} D_{\text{old}}.$$

The lemma extends the time analysis of the Lovász algorithm to the Small Integer Relation Algorithm. It holds for arbitrary positive numbers k. Lemma 3.3 holds for Bergman's exchange rule, but does not hold if, in the algorithm, the Lovász exchange rule is used instead. However in the particular case $k=\infty$, i.e., $\alpha(\mathbf{b}_i^*) = \|\mathbf{b}_i^*\|^2 2^n$, the lemma does hold for the Lovász exchange rule. The Lovász exchange rule performs well provided that the numbers $\|\mathbf{b}_i^*\|$ cannot become arbitrarily small and nonzero, as is the case for reduction of a generator system of a lattice (Theorem 2.1). The lemma shows that Bergman's exchange rule performs well in some sense even if some of the numbers $\|\mathbf{b}_i^*\|$ become arbitrarily small and nonzero during the computation. There is a subtle interplay between the termination condition $\max_{i=1,\cdots,n-1} \|\mathbf{b}_i^*\| \leqq 2^{-k}$ and Bergman's exchange rule. Bergman's exchange rule strives to make this termination condition valid as soon as possible.

Proof. Since i has been chosen to maximize the number $2^i \|\mathbf{b}_i^*\|^2$, we have

$$\|\mathbf{b}_{i+1}^{\text{old}^*}\|^2 \leqq \tfrac{1}{2} \|\mathbf{b}_i^{\text{old}^*}\|^2.$$

From this and $|\mu_{i+1,i}| \leqq \frac{1}{2}$ we see

$$\|\mathbf{b}_i^{\text{new}^*}\|^2 = \|\mathbf{b}_{i+1}^{\text{old}^*}\|^2 + \mu_{i+1,i}^2 \|\mathbf{b}_i^{\text{old}^*}\|^2 \leqq \tfrac{3}{4} \|\mathbf{b}_i^{\text{old}^*}\|^2. \tag{3.2}$$

Since the algorithm did not previously terminate in step 2 some j satisfies $\|\mathbf{b}_j^{\text{old}^*}\| > 2^{-k}$. Since i has been chosen to maximize $2^i \|\mathbf{b}_i^*\|^2$ we have

$$2^n \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2^i \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2^j \|\mathbf{b}_j^{\text{old}^*}\|^2 \geqq 2^{-2k+j} \geqq 2^{-2k+1}. \tag{3.3}$$

We conclude from $\|\mathbf{b}_i^{\text{old}^*}\| \geqq \|\mathbf{b}_{i+1}^{\text{new}^*}\|$ that

$$\alpha(\mathbf{b}_i^{\text{old}^*}) \geqq \alpha(\mathbf{b}_{i+1}^{\text{new}^*}). \tag{3.4}$$

We are going to prove the inequality

$$\frac{\alpha(\mathbf{b}_i^{\text{new}^*})\alpha(\mathbf{b}_{i+1}^{\text{new}^*})}{\alpha(\mathbf{b}_i^{\text{old}^*})\alpha(\mathbf{b}_{i+1}^{\text{old}^*})} \leqq 1. \tag{3.5}$$

If $\alpha(\mathbf{b}_i^{\text{new}^*}) = 2^{-2k}$, then (3.5) follows from (3.4) and $\alpha(\mathbf{b}_{i+1}^{\text{old}^*}) \geqq 2^{-2k}$. If $\alpha(\mathbf{b}_{i+1}^{\text{new}^*}) = 2^{-2k}$, the inequality (3.5) follows from $\alpha(\mathbf{b}_{i+1}^{\text{old}^*}) \geqq 2^{-2k}$ and $\alpha(\mathbf{b}_i^{\text{new}^*}) \leqq \alpha(\mathbf{b}_i^{\text{old}^*})$, which follows from (3.2). If $\alpha(\mathbf{b}_i^{\text{new}^*}) = \|\mathbf{b}_i^{\text{new}^*}\|^2 2^n$ and $\alpha(\mathbf{b}_{i+1}^{\text{new}^*}) = \|\mathbf{b}_{i+1}^{\text{new}^*}\|^2 2^n$ the inequality (3.5) holds by virtue of

$$\|\mathbf{b}_i^{\text{new}^*}\| \, \|\mathbf{b}_{i+1}^{\text{new}^*}\| = \|\mathbf{b}_i^{\text{old}^*}\| \, \|\mathbf{b}_{i+1}^{\text{old}^*}\|.$$

From the inequality (3.5) we finally conclude

$$\frac{D_{\text{new}}}{D_{\text{old}}} = \frac{\alpha(\mathbf{b}_i^{\text{new}^*})^{n-i} \alpha(\mathbf{b}_{i+1}^{\text{new}^*})^{n-i-1}}{\alpha(\mathbf{b}_i^{\text{old}^*})^{n-i} \alpha(\mathbf{b}_{i+1}^{\text{old}^*})^{n-i-1}}$$

$$\overset{(3.5)}{\leqq} \alpha(\mathbf{b}_i^{\text{new}^*}) / \alpha(\mathbf{b}_i^{\text{old}^*}) \overset{(3.3)}{=} 2^{-n} \alpha(\mathbf{b}_i^{\text{new}^*}) \|\mathbf{b}_i^{\text{old}^*}\|^{-2} \overset{(3.2),(3.3)}{\leqq} \frac{3}{4}. \qquad \square$$

To complete the proof of Theorem 3.2, it suffices to observe that the algorithm starts with $D \leqq 2^{n^3}$, and $D \geqq 2^{-kn^2}$ holds on termination. Thus by Lemma 3.3 there can be at most $O(n^2(k+n))$ exchange steps. Each exchange step uses at most $O(n)$

arithmetic steps; this includes the steps to update $\mu_{i,j}$, $\mu_{i+1,j}\mu_{j,i}$, $\mu_{j,i+1}$ for $j=1,\cdots,n$ and $\|\mathbf{b}_i^*\|^2$, $\|\mathbf{b}_{i+1}^*\|^2$. The initial computation of the numbers $\mu_{i,j}$, $\|\mathbf{b}_i^*\|^2$ for $1<j<i\leqq n$ can be done using $O(n^3)$ arithmetic operations. This shows that the algorithm halts after at most $O(n^3(k+n))$ arithmetic operations on real numbers.

The second part of the theorem follows immediately from Proposition 3.1. □

Theorem 3.2 implies that the Basic Integer Relation Algorithm always detects an integer relation in $O(n^3 \log \lambda(\mathbf{x})+n)$ arithmetic operations if one exists; simply apply the Small Integer Relation Algorithm with $k=2^j$ for $j=1, 2, \cdots$, until an integer relation is found.

An interesting feature of the integer relation problem is that it contains a gap between the largest size 2^{n+k} of an integer relation found and the size 2^k of an integer relation proved not to exist. Some sort of gap like this seems necessary in order to obtain polynomial time algorithms. It is unreasonable to expect that there exists a fast algorithm to answer the question: "Does there exist an integer relation $\mathbf{m}$ for $\mathbf{x}$ with $\|\mathbf{m}\|\leqq 2^k$?" This problem contains the shortest vector problem for integer lattices, i.e., the problem to decide for a given lattice basis whether there exists a lattice vector $\mathbf{y}$ with $\|\mathbf{y}\|\leqq 2^k$. This problem is believed to be hard and is known to be NP-complete for the version in which the Euclidean norm $\|\mathbf{y}\|$ is replaced by the sup-norm $\|\mathbf{y}\|_{\sup}$ (see Van Emde Boas (1981)).

The Small Integer Relation Algorithm can also be applied starting with a basis $\mathbf{b}_1,\cdots,\mathbf{b}_n\in\mathbb{R}^n$ of an arbitrary lattice $M=\langle\mathbf{b}_1,\cdots,\mathbf{b}_n\rangle$ of rank n, possibly distinct from $\mathbb{Z}^n$. In this case the row vectors $\mathbf{c}_1,\cdots,\mathbf{c}_n$ of the matrix

$$[\mathbf{c}_1,\cdots,\mathbf{c}_n]^{\top}=[\mathbf{b}_1,\cdots,\mathbf{b}_n]^{-1}$$

form a basis of the *dual (or reciprocal, or polar) lattice* M^*, defined as

$$M^*=\{\mathbf{y}\in \operatorname{span}(\mathbf{b}_1,\cdots,\mathbf{b}_n)\,|\,\langle\mathbf{b}_i,\mathbf{y}\rangle\in\mathbb{Z} \text{ for } i=1,\cdots,n\}.$$

(In the case $M=\mathbb{Z}^n$ we have $M=\mathbb{Z}^n=M^*$.) In this case we consider the lattice

$$L_{\mathbf{x},M}=\{\mathbf{y}\in M^*\,|\,\langle\mathbf{y},\mathbf{x}\rangle=0\}$$

of relations $\mathbf{y}$ for $\mathbf{x}$ that are in M^*. Let $\lambda_M(\mathbf{x})$ be the length of the shortest, nonzero relation in $L_{\mathbf{x},M}$. Proposition 3.1 holds for arbitrary lattices $M\subset\mathbb{R}^n$ of rank n provided that relations for $\mathbf{x}$ are in $M^*$and $\lambda(\mathbf{x})$ is replaced by $\lambda_M(\mathbf{x})$. The time analysis of Lemma 3.3 remains valid, too. This observation proves the following result.

THEOREM 3.4. *Suppose the Small Integer Relation Algorithm is run with input* $\mathbf{x}\in\mathbb{R}^n$ *and starting with a basis* $\mathbf{b}_1,\cdots,\mathbf{b}_n$ *of lattice* $M\subset\mathbb{R}^n$. *Then the Small Integer Relation Algorithm halts after at most* $O(n^3(k+n))$ *arithmetic steps. It either finds a relation* $\mathbf{c}_n\in M^*$ *for* $\mathbf{x}$ *satisfying* $\|\mathbf{c}_n\|^2\leqq 2^{n-2}\min\{\lambda_M(\mathbf{x})^2, 2^{2k}\}$ *or it proves* $\lambda_M(\mathbf{x})\geqq 2^k$.

The arithmetic complexity model describes computations with real numbers although on actual computers we can only approximate computations with real numbers. In particular, it is an undecidable problem to test the equality of two computable real numbers specified by computer programs that compute them to arbitrary accuracy. In consequence on an actual computer (Turing machine) we cannot prove the existence of an integer relation for a set of real numbers. However we *can* prove the nonexistence of a small integer relation using only computations on rational numbers. We can prove $\lambda(\bar{\mathbf{x}})\geqq 2^k$ for a given real vector $\bar{\mathbf{x}}$ by applying the Small Integer Relation Algorithm to a sufficiently close rational approximation $\mathbf{x}$ to $\bar{\mathbf{x}}$. We cannot say beforehand how close $\mathbf{x}$ has to be to $\bar{\mathbf{x}}$, but the term "sufficiently close" can be defined a posteriori. We can certify that a given $\mathbf{x}$ is "sufficiently close" to $\bar{\mathbf{x}}$ by the following theorem.

THEOREM 3.5. *Suppose that the Small Integer Relation Algorithm claims "*$\lambda(\mathbf{x}) \geqq 2^k$*." Then* $\lambda(\bar{\mathbf{x}}) \geqq 2^{k-1}$ *holds for every vector* $\bar{\mathbf{x}} \in \mathbb{R}^n$ *such that* $\max_i \|\mathbf{b}_i\| \|\mathbf{x} - \bar{\mathbf{x}}\| \leqq 2^{-k} \min\{\|\mathbf{x}\|, \|\bar{\mathbf{x}}\|\}$ *where* $\mathbf{b}_1, \cdots, \mathbf{b}_n$ *is the basis on termination.*

Proof. Let $\mathbf{b}^*_{i,\mathbf{x}}$, ($\mathbf{b}^*_{i,\bar{\mathbf{x}}}$, respectively) be the component of $\mathbf{b}_i$ that is orthogonal to $\mathbf{b}_1, \cdots, \mathbf{b}_{i-1}$, $\mathbf{x}$ (to $\mathbf{b}_1, \cdots, \mathbf{b}_{i-1}$, $\bar{\mathbf{x}}$, respectively), and let $\pi_{\mathbf{x}}$, $\pi_{\bar{\mathbf{x}}}$ be the orthogonal projection to $(\mathbf{x}\mathbb{R})^\top$, $(\bar{\mathbf{x}}\mathbb{R})^\top$. It is sufficient to prove $\|\mathbf{b}^*_{i,\bar{\mathbf{x}}}\| \leqq 2^{-k+1}$ for $i = 1, \cdots, n$, which implies $\lambda(\bar{\mathbf{x}}) \geqq 2^{k-1}$ by Proposition 3.1. Below we show the inequality

$$\|\pi_{\mathbf{x}}(\mathbf{b}) - \pi_{\bar{\mathbf{x}}}(\mathbf{b})\| \leqq \|\mathbf{b}\| \|\mathbf{x} - \bar{\mathbf{x}}\| \text{ for all } \mathbf{b}, \mathbf{x}, \bar{\mathbf{x}} \text{ satisfying } \|\mathbf{x}\|, \|\bar{\mathbf{x}}\| \geqq 1. \tag{3.6}$$

This implies for all vectors $\mathbf{x}$, $\bar{\mathbf{x}}$

$$\|\mathbf{b}^*_{i,\mathbf{x}} - \mathbf{b}^*_{i,\bar{\mathbf{x}}}\| \leqq \|\mathbf{b}_i\| \|\mathbf{x} - \bar{\mathbf{x}}\| / \min\{\|\mathbf{x}\|, \|\bar{\mathbf{x}}\|\} \leqq 2^{-k}$$

by the assumptions on $\mathbf{x}$, $\bar{\mathbf{x}}$, and $\mathbf{b}_i$. Since on termination the Small Integer Relation Algorithm has $\|\mathbf{b}^*_{i,\mathbf{x}}\| \leqq 2^{-k}$ for $i = 1, \cdots, n$ the inequality above yields $\|\mathbf{b}^*_{i,\bar{\mathbf{x}}}\| \leqq 2^{-k+1}$ for $i = 1, \cdots, n$, which proves the theorem.

Proof of equation (3.6). We prove this in three cases.

Case $\|\mathbf{x}\| = \|\bar{\mathbf{x}}\| = 1$ and $\mathbf{b} \in \text{span}(\mathbf{x}, \bar{\mathbf{x}})$. Without loss of generality let $\mathbf{x}$, $\bar{\mathbf{x}}$, $\mathbf{b} \in \mathbb{R}^2$, and $\mathbf{x} = (x_1, x_2)$, $\bar{\mathbf{x}} = (1, 0)$, $\mathbf{b} = (b_1, b_2)$. Then we have

$$\begin{aligned}
\|\pi_{\mathbf{x}}(\mathbf{b}) - \pi_{\bar{\mathbf{x}}}(\mathbf{b})\|^2 &= \|\langle \mathbf{x}, \mathbf{b}\rangle \mathbf{x} - \langle \bar{\mathbf{x}}, \mathbf{b}\rangle \bar{\mathbf{x}}\|^2 \\
&= ((x_1 b_1 + x_2 b_2) x_1 - b_1)^2 + (x_1 b_1 + x_2 b_2)^2 x_2^2 \\
&= b_1^2 - 2 b_1 x_1 (x_1 b_1 + x_2 b_2) + (x_1 b_1 + x_2 b_2)^2 \quad (\text{since } x_1^2 + x_2^2 = 1) \\
&= b_1^2 + x_2^2 b_2^2 - x_1^2 b_1^2 = \|\mathbf{b}\|^2 x_2^2 \leqq \|\mathbf{b}\|^2 \|\mathbf{x} - \bar{\mathbf{x}}\|^2.
\end{aligned}$$

Case $\|\mathbf{x}\| = \|\bar{\mathbf{x}}\| \geqq 1$. Let $\bar{\mathbf{b}}$ be the component of $\mathbf{b}$ in span $(\mathbf{x}, \bar{\mathbf{x}})$. Then

$$\|\pi_{\mathbf{x}}(\mathbf{b}) - \pi_{\bar{\mathbf{x}}}(\mathbf{b})\| = \|\pi_{\mathbf{x}}(\bar{\mathbf{b}}) - \pi_{\bar{\mathbf{x}}}(\bar{\mathbf{b}})\| \leqq \|\bar{\mathbf{b}}\| \|\mathbf{x} - \bar{\mathbf{x}}\| \leqq \|\mathbf{b}\| \|\mathbf{x} - \bar{\mathbf{x}}\|$$

follows from the previous case.

General Case. We assume without loss of generality that $\|\bar{\mathbf{x}}\| \geqq \|\mathbf{x}\| \geqq 1$. Application of the previous case to $\tilde{\mathbf{x}} = \bar{\mathbf{x}}\|\mathbf{x}\| / \|\bar{\mathbf{x}}\|$ gives $\|\pi_{\mathbf{x}}(\mathbf{b}) - \pi_{\bar{\mathbf{x}}}(\mathbf{b})\| = \|\pi_{\mathbf{x}}(\mathbf{b}) - \pi_{\tilde{\mathbf{x}}}(\mathbf{b})\| \leqq \|\mathbf{b}\| \|\mathbf{x} - \tilde{\mathbf{x}}\| \leqq \|\mathbf{b}\| \|\mathbf{x} - \bar{\mathbf{x}}\|$. □

4. Finding linearly independent integer relations. There is a natural adaptation of the Small Integer Relation Algorithm to find several small linearly independent integer relations. We modify the exchange rule and the termination test accordingly. The following algorithm either finds r small linearly independent integer relations or proves they do not exist.

SEVERAL RELATIONS ALGORITHM. (On input $\mathbf{x} \in \mathbb{R}^n$ and r, $k \in \mathbb{N}$ this algorithm either finds r linearly independent integer relations $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ for $\mathbf{x}$ or it proves that $\lambda_r(\mathbf{x}) \geqq 2^k$.)

1. *Initiation.* For $\mathbf{b}_0 = \mathbf{x}$ and the standard basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of $\mathbb{Z}^n$ compute the Gram-Schmidt numbers $\mu_{i,j}$, $\|\mathbf{b}^*_i\|^2$ for $1 \leqq i, j \leqq n$. Find the unique $s \leqq n$ such that $\mathbf{b}^*_s = \mathbf{0}$.
2. *Termination test.* If $s \leqq n - r$ then r linearly independent integer relations are found; size-reduce $\mathbf{b}_1, \cdots, \mathbf{b}_n$, set $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top := [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$, output $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ and stop. If $\|\mathbf{b}^*_i\| \leqq 2^{-k}$ for $1 \leqq i \leqq n - r + 1$ then r small independent relations do not exist, output "$\lambda_r(\mathbf{x}) \geqq 2^k$" and stop.
3. *Exchange step.* Choose that i with $1 \leqq i < s$ that maximizes $\|\mathbf{b}^*_i\|^2 2^i$. Size-reduce $\mathbf{b}_{i+1}$ with respect to $\mathbf{b}_i$ by setting $\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$. Exchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$.

Update the Gram-Schmidt quantities $\|\mathbf{b}_\nu^*\|^2, \mu_{\nu,j}, \mu_{j,\nu}$ for $\nu = i, i+1$ and $1 \leqq j \leqq n$. If $\mathbf{b}_i^* = \mathbf{0}$ then set $s := i$. Go to 2.

There are two features that distinguish this algorithm from the Small Integer Relation Algorithm. First, the exchange rule only allows exchanges $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ such that i is smaller than that s for which $\mathbf{b}_s^* = \mathbf{0}$. This exchange rule strives to minimize $\max_{j=1,\cdots,s} \|\mathbf{b}_j^*\|^2 2^j$ where $s \geqq n-r$ holds throughout the algorithm. This rule is justified by fact (4.1) proved below that $1/\max_{j=1,\cdots,n-r+1} \|\mathbf{b}_j^*\|$ is a lower bound for $\lambda_r(\mathbf{x})$. The termination condition is modified accordingly so that the algorithm stops if $\max_{i=1,\cdots,n-r+1} \|\mathbf{b}_i^*\| \leqq 2^{-k}$. Second, in the termination step before computing the inverse matrix $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top = [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$, the basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ is size-reduced. This size-reduction makes the output relations $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ small as is proved in Theorem 4.2 below. The resulting basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ has the same $\mathbf{b}_i^*$ as before and $|\mu_{i,j}| \leqq \frac{1}{2}$ holds for all $1 \leqq j < i \leqq n$.

THEOREM 4.1. *When given for input a vector $\mathbf{x} \in \mathbb{R}^n$ and r, $k \in \mathbb{N}$ then the Several Relations Algorithm either finds r linearly independent integer relations $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ for $\mathbf{x}$ or proves $\lambda_r(\mathbf{x}) \geqq 2^k$. It halts after at most $O(n^3(k+n))$ arithmetic operations on real numbers.*

Proof. Correctness. The algorithm has two possibilities for termination. First, if $\mathbf{b}_s^* = \mathbf{0}$ occurs with $s \leqq n-r$ then $\mathbf{x} \in \text{span}(\mathbf{b}_1, \cdots, \mathbf{b}_{n-r})$. Since $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ are orthogonal to span $(\mathbf{b}_1, \cdots, \mathbf{b}_{n-r})$ they are r linearly independent integer relations for $\mathbf{x}$. Second, we must show that $\|\mathbf{b}_i^*\| \leqq 2^{-k}$ for $i = 1, \cdots, n-r+1$ implies $\lambda_r(\mathbf{x}) \geqq 2^k$. This follows from the claim that

$$\lambda_r(\mathbf{x}) \geqq 1/\max_{1 \leqq i \leqq n-r+1} \|\mathbf{b}_i^*\|. \tag{4.1}$$

To prove this inequality let $\mathbf{m}_1, \cdots, \mathbf{m}_r \in L_\mathbf{x}$ be linearly independent vectors such that $\|\mathbf{m}_i\| = \lambda_i(\mathbf{x})$ for $i = 1, \cdots, r$. Then there exists $i \leqq n-r+1$ and $j \leqq r$ such that $\langle \mathbf{m}_j, \mathbf{b}_i \rangle \neq 0$. For fixed j let i be minimal with $\langle \mathbf{m}_j, \mathbf{b}_i \rangle \neq 0$. This yields

$$|\langle \mathbf{m}_j, \mathbf{b}_i \rangle| = |\langle \mathbf{m}_j, \mathbf{b}_i^* \rangle| \geqq 1.$$

Hence $\|\mathbf{m}_j\| \geqq 1/\|\mathbf{b}_i^*\|$, which proves (4.1).

Running time bound. We use the quantity $D = \prod_{i=1}^n \max\{\|\mathbf{b}_i^*\|^2 2^n, 2^{-2k}\}^{n-i}$ used in the proof of Theorem 3.2. We claim that at each exchange step we have

$$D_{\text{new}} \leqq \tfrac{3}{4} D_{\text{old}}. \tag{4.2}$$

This follows by the argument of Lemma 3.3, with two modifications to reflect the new exchange rule. The new algorithm exchanges $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ where $2^i \|\mathbf{b}_i^*\|^2$ is maximized over $1 \leqq i \leqq s-1$, and $\mathbf{b}_s^* = \mathbf{0}$. Thus

$$\|\mathbf{b}_{i+1}^{\text{old}^*}\|^2 \leqq \tfrac{1}{2} \|\mathbf{b}_i^{\text{old}^*}\|^2$$

still holds. Since the algorithm did not previously halt there is some j with $1 \leqq j \leqq s-1$, $\|\mathbf{b}_j^{\text{old}^*}\| > 2^{-k}$, and the inequality

$$2^n \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2^i \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2^j \|\mathbf{b}_j^{\text{old}^*}\|^2 \geqq 2^{-2k+j} \geqq 2^{-2k+1} \tag{3.3}$$

still holds.

The $O(n^3(n+k))$ arithmetic operation bound now follows from (4.2) exactly as in Theorem 3.2. □

THEOREM 4.2. *If the Several Relations Algorithm finds r integer relations $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ for $\mathbf{x}$ then these satisfy $\|\mathbf{c}_{n-i+1}\|^2 \leqq 1.5^{2i-2} 2^{n-1} \lambda_i(\mathbf{x})^2$ for $i = 1, \cdots, r$.*

Proof. If the initiation step finds s with $s \leqq n-r$ then we have $\mathbf{c}_{n-i+1} = \mathbf{e}_{n-i+1}$ for $i = 1, \cdots, r$ and the claim holds. Now assume that initially we have $s > n-r$. Let $\mathbf{b}_1, \cdots, \mathbf{b}_n$ be the basis of lattice $\mathbb{Z}^n$ on termination of the Several Relations Algorithm. Since we have found r integer relations we have $\mathbf{b}_s^* = \mathbf{0}$ with $s = n-r$ and

$$[\mathbf{b}_1, \cdots, \mathbf{b}_n] = [\mathbf{b}_1^*, \cdots, \mathbf{b}_{s-1}^*, \mathbf{x}, \mathbf{b}_{s+1}^*, \cdots, \mathbf{b}_n^*] W^{\top},$$

holds for the matrix $W = [w_{i,j}]_{1 \leqq i,j \leqq n}$ with entries

$$w_{i,j} = \begin{cases} \langle \mathbf{b}_i, \mathbf{x} \rangle \|\mathbf{x}\|^{-2}, & j = s, \\ \mu_{i,j}, & j \neq s. \end{cases}$$

We have $w_{i,i} = 1$ for $i \neq s$ and, by the size-reduction in step 2, $|w_{i,j}| \leqq 1/2$ for $1 \leqq j < i \leqq n$ with $j \neq s$. The matrix W is lower triangular for all except the sth column, i.e., $w_{i,j} = 0$ for $i < j \neq s$. From the definition of $\mathbf{c}_1, \cdots, \mathbf{c}_n$ and since the vectors $\mathbf{x}, \mathbf{b}_1^*, \cdots, \mathbf{b}_n^*$ are orthogonal we conclude

$$\begin{aligned} [\mathbf{c}_1, \cdots, \mathbf{c}_n] &= ([\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1})^{\top} \\ &= [\mathbf{b}_1^*, \cdots, \mathbf{b}_{s-1}^*, \mathbf{x}, \mathbf{b}_{s+1}^*, \cdots, \mathbf{b}_n^*] \begin{bmatrix} \|\mathbf{b}_1^*\|^{-2} & & & & 0 \\ & \ddots & & & \\ & & \|\mathbf{x}\|^{-2} & & \\ & & & \ddots & \\ 0 & & & & \|\mathbf{b}_n^*\|^{-2} \end{bmatrix} W^{-1}. \end{aligned}$$

Let the matrix $V := W^{-1}$ have entries $v_{i,j}$. Since V is lower triangular for the last r columns we have

$$\|\mathbf{c}_{n-i+1}\|^2 = \sum_{j \geqq n-i+1} v_{j,n-i+1}^2 \|\mathbf{b}_j^*\|^{-2} \quad \text{for } i = 1, \cdots, r. \tag{4.3}$$

We prove below the inequalities

$$|v_{i,j}| \leqq 1.5^{i-j} \quad \text{for } s < j \leqq i \leqq n, \tag{4.4}$$

$$\|\mathbf{b}_{n-i+1}^*\|^{-2} \leqq \lambda_i(\mathbf{x})^2 2^{n-i-1} \quad \text{for } i = 1, \cdots, r. \tag{4.5}$$

The desired bounds follow from (4.3), (4.4), and (4.5):

$$\begin{aligned} \|\mathbf{c}_{n-i+1}\|^2 &\overset{(4.3),(4.4)}{\leqq} \sum_{j=1}^{i} 1.5^{2(i-j)} \|\mathbf{b}_{n-j+1}^*\|^{-2} \\ &\overset{(4.5)}{\leqq} \sum_{j=1}^{i} 1.5^{2(i-j)} 2^{n-j-1} \lambda_j(\mathbf{x})^2 \\ &\leqq \begin{cases} \lambda(\mathbf{x})^2 2^{n-2}, & i = 1, \\ \lambda_i(\mathbf{x})^2 1.5^{2i-2} 2^{n-1}, & i = 2, \cdots, r. \end{cases} \end{aligned}$$

It remains to prove the inequalities (4.4), (4.5). The inequality (4.4) follows from $|w_{i,j}| \leqq \frac{1}{2}$ for $s < j < i \leqq n$, $w_{i,i} = 1$ for $i > s$, together with the fact that the matrix W is lower triangular for the last r columns. We prove the inequality (4.4) by induction on $q = i - j$. If $q = 0$ we have $i = j$ and $v_{i,i} = 1$ holds for $i = s+1, \cdots, n$ since the matrix $V^{-1} = W$ is lower triangular, for the last $n-s$ columns, and has ones in the diagonal. For the induction step $q-1 \to q$ consider the equation

$$\sum_{t=j}^{i} \mu_{i,t} v_{t,j} = 0 \quad \text{for } s < j < i,$$

which follows from $V = W^{-1}$. We see that

$$v_{i,j} = -\sum_{t=j}^{j+q-1} \mu_{i,t} v_{t,j}$$

and the induction hypothesis implies

$$|v_{i,j}| \leqq \sum_{\nu=0}^{q-1} \frac{1}{2} 1.5^{\nu} \leqq 1.5^{q}.$$

To prove (4.5) consider the basis $\bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_n$ of $\mathbb{Z}^n$ before the last exchange $\mathbf{b}_{n-r} \leftrightarrow \mathbf{b}_{n-r+1}$. The maximum of $\|\bar{\mathbf{b}}_j^*\|^2 2^j$ for $j \leqq n-r$ is at $j = n-r$, and $\bar{\mathbf{b}}_{n-r+1}^* = \mathbf{0}$. Thus we have for $j = 1, \cdots, n-r+1$:

$$\|\bar{\mathbf{b}}_j^*\|^2 \leqq \|\bar{\mathbf{b}}_{n-r}^*\|^2 2^{n-r-j} = \|\mathbf{b}_{n-r+1}^*\|^2 2^{n-r-j}.$$

Hence

$$\lambda_r(\mathbf{x}) \overset{(4.1)}{\geqq} 1/\max_{\nu \leqq n-r+1} \|\bar{\mathbf{b}}_\nu^*\| \geqq \|\mathbf{b}_{n-r+1}^*\|^{-1} 2^{(-n+r+1)/2}.$$

This proves (4.5) for $i = r$. The inequality (4.5) for $i < r$ follows in the same way from the basis of $\mathbb{Z}^n$ before the last exchange $\mathbf{b}_{n-i} \leftrightarrow \mathbf{b}_{n-i+1}$. □

The Several Relations Algorithm can also be modified to find a basis of a lattice L of integer relations for $\mathbf{x}$ that contains all shortest vectors in $L_{\mathbf{x}}$.

The *Shortest Relations Algorithm*, which takes as input $\mathbf{x} \in \mathbb{R}^n$, is just the Several Integer Relations Algorithm with a new termination test.

New termination test. If $s < n$ and $\|\mathbf{b}_j^*\| < \|\mathbf{b}_n^*\|$ holds for $1 \leqq j \leqq s-1$ then size-reduce $\mathbf{b}_1, \cdots, \mathbf{b}_n$, compute $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^{\top} := [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$, output $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$, and stop.

The algorithm will not halt unless $\mathbf{x}$ has at least one integer relation. We show that if $L_{\mathbf{x}} \neq \{\mathbf{0}\}$ then the algorithm halts in time polynomial in n and $\log \lambda(\mathbf{x})$ and finds a lattice containing all shortest integer relations.

THEOREM 4.3. *When given for input a vector* $\mathbf{x}$ *with* $L_{\mathbf{x}} \neq \{\mathbf{0}\}$, *the Shortest Relations Algorithm finds integer relations* $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$ *for* $\mathbf{x}$ *such that*:

(1) *The lattice* $\langle \mathbf{c}_{s+1}, \cdots, \mathbf{c}_n \rangle$ *contains all shortest integer relations for* $\mathbf{x}$,

(2) $\|\mathbf{c}_i\| \leqq 3^n \lambda(\mathbf{x})$ *for* $i = s+1, \cdots, n$.

The Shortest Relations Algorithm halts after at most $O(n^4 + n^3 \log \lambda(\mathbf{x}))$ *arithmetic operations on real numbers.*

Proof. Let $\mathbf{b}_1, \cdots, \mathbf{b}_n$ be the basis of lattice $\mathbb{Z}^n$ on termination of the Several Relations Algorithm. On termination we have $\mathbf{b}_s^* = \mathbf{0}$ and thus $\mathbf{x} \in \operatorname{span}(\mathbf{b}_1, \cdots, \mathbf{b}_s)$. Therefore the vectors $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$ are integer relations for $\mathbf{x}$.

Next we show that the lattice

$$L = \langle \mathbf{c}_{s+1}, \cdots, \mathbf{c}_n \rangle$$

contains all $\mathbf{w} \in L_{\mathbf{x}}$ satisfying $\|\mathbf{w}\| \leqq \lambda(\mathbf{x})$. If $\mathbf{w}$ is not in the span of $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$ there exists an integer $i \leqq s$ such that $\langle \mathbf{w}, \mathbf{b}_i \rangle \neq 0$ and consequently the smallest such i satisfies

$$|\langle \mathbf{w}, \mathbf{b}_i^* \rangle| = |\langle \mathbf{w}, \mathbf{b}_i \rangle| \geqq 1.$$

Since the new termination condition holds we have

$$\|\mathbf{w}\| \geqq \|\mathbf{b}_i^*\|^{-1} > \|\mathbf{b}_n^*\|^{-1} = \|\mathbf{c}_n\| \geqq \lambda(\mathbf{x}).$$

If however $\mathbf{w}$ is in the span of $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$ then $\mathbf{w}$ must be in L since $\mathbf{c}_1, \cdots, \mathbf{c}_n$ is a basis of the lattice $\mathbb{Z}^n$.

We next establish property (2). Let $\mathbf{b}_1, \cdots, \mathbf{b}_n$ be the basis of lattice $\mathbb{Z}^n$ on termination of the Shortest Relations Algorithm. We have

$$\|\mathbf{c}_i\|^2 = \sum_{j=i}^{n} v_{j,i}^2 \|\mathbf{b}_j^*\|^{-2} \quad \text{for } i = s+1, \cdots, n$$

where $|v_{j,i}| \leqq 1.5^{j-i}$. Therefore the inequality (2) follows from

$$\|\mathbf{b}_j^*\|^{-2} \leqq \lambda(\mathbf{x})^2 2^{n+j-3} \quad \text{for } j = s+1, \cdots, n. \tag{4.6}$$

Proof of equation (4.6). The inequality holds for $\mathbf{b}_n^*$ since the output vector $\mathbf{c}_n = \pm \mathbf{b}_n^* \|\mathbf{b}_n^*\|^{-2}$ of the Shortest Relations Algorithms, by Proposition 3.1(3), satisfies $\|\mathbf{b}_n^*\|^{-2} = \|\mathbf{c}_n\|^2 \leqq 2^{n-2}\lambda(\mathbf{x})^2$. To prove the inequality for $\mathbf{b}_j^*$ with $s+1 \leqq j < n$, we consider the basis $\bar{\mathbf{b}}_1, \cdots, \bar{\mathbf{b}}_n$ before the last exchange step $\mathbf{b}_{j-1} \leftrightarrow \mathbf{b}_j$ of the Shortest Relations Algorithm. We have $\bar{\mathbf{b}}_n^* = \mathbf{b}_n^*$ and the exchange achieves $\mathbf{b}_{j-1}^{\text{new}^*} = \mathbf{0}$ and $\mathbf{b}_j^* = \mathbf{b}_j^{\text{new}^*} = \bar{\mathbf{b}}_{j-1}^*$. Before this exchange there exists $i < j$ satisfying $\|\bar{\mathbf{b}}_i^*\| > \|\bar{\mathbf{b}}_n^*\| = \|\mathbf{b}_n^*\|$, since the termination condition does not yet hold. From this and $\max_{i<j} \|\bar{\mathbf{b}}_i^*\|^2 2^i = \|\bar{\mathbf{b}}_{j-1}^*\|^2 2^{j-1}$ we conclude

$$\|\mathbf{b}_j^*\|^2 = \|\bar{\mathbf{b}}_{j-1}^*\|^2 \geqq \|\bar{\mathbf{b}}_i^*\|^2 2^{i-j+1} > \|\mathbf{b}_n^*\|^2 2^{i-j+1}.$$

This implies

$$\|\mathbf{b}_j^*\|^{-2} \leqq \|\mathbf{b}_n^*\|^{-2} 2^{j-i-1} \leqq \lambda(\mathbf{x})^2 2^{n-2} 2^{j-1},$$

which finishes the proof of (4.6).

Running time bound. We use the quantity

$$D = \prod_{i=1}^{n-1} \max\{\|\mathbf{b}_i^*\|^2 2^n, 2^{-n}\lambda(\mathbf{x})^{-2}\}^{n-i}.$$

It is the quantity of Lemma 3.3 except that 2^k is replaced by $2^{n/2}\lambda(\mathbf{x})$. We adjust the proof of Lemma 3.3 to show that each exchange step $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ achieves

$$D_{\text{new}} \leqq \tfrac{3}{4} D_{\text{old}}.$$

It is sufficient to show that the inequality (3.3) holds with 2^k replaced by $2^{n/2}\lambda(\mathbf{x})$, i.e., that $2^n \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2 \cdot 2^{-n}\lambda(\mathbf{x})^{-2}$ holds for each exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$.

We first show that there exists $j < s$ such that $\|\mathbf{b}_j^{\text{old}^*}\| \geqq 2^{-n/2}\lambda(\mathbf{x})^{-1}$. If $s < n$ this holds because $\mathbf{b}_n^{\text{old}}$ is the vector $\mathbf{b}_n$ on termination; since the algorithm did not previously halt there exists j with $1 \leqq j \leqq s-1$ such that $\|\mathbf{b}_j^{\text{old}^*}\| \geqq \|\mathbf{b}_n^{\text{old}^*}\| = \|\mathbf{c}_n\|^{-1} \geqq 2^{-n/2}\lambda(\mathbf{x})^{-1}$, where the latter inequality holds by Proposition 3.1(3). If $s = n$ we see from Proposition 3.1(2) that there exists $j \leqq n$ such that $\|\mathbf{b}_j^{\text{old}^*}\| \geqq \lambda(\mathbf{x})^{-1}$ and we have $j < n$ since $\|\mathbf{b}_n^{\text{old}^*}\| = 0$.

Since i with $1 \leqq i \leqq s-1$ was chosen such that $\|\mathbf{b}_i^{\text{old}^*}\|^2 2^i$ is maximal for $i < s$ we have for the above j that

$$2^n \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2^i \|\mathbf{b}_i^{\text{old}^*}\|^2 \geqq 2^j \|\mathbf{b}_j^{\text{old}^*}\|^2 \geqq 2 \cdot 2^{-n}\lambda(\mathbf{x})^{-2}.$$

From this we prove $D_{\text{new}} \leqq \frac{3}{4} D_{\text{old}}$ exactly as in the proof of Lemma 3.3 with 2^k replaced by $2^{n/2}\lambda(\mathbf{x})$.

Using the relation $D_{\text{new}} \leqq \frac{3}{4} D_{\text{old}}$ the claimed time bound is straightforward. Initially we have for the Shortest Relations Algorithm that $D \leqq 2^{n^3}$ and $D \geqq 2^{-2n^3}\lambda(\mathbf{x})^{-2n^2}$ holds on termination. Therefore the Shortest Relations Algorithm performs at most $O(n^3 + n^2 \log \lambda(\mathbf{x}))$ exchange steps. Since each exchange step uses at most $O(n)$ arithmetic operations the total number of arithmetic steps of the Shortest Relations Algorithm is at most $O(n^4 + n^3 \log \lambda(\mathbf{x}))$. □

Remarks. (i) If on termination of the Shortest Relations Algorithm we have $s = n-1$, then the output $\mathbf{c}_n$ is a shortest integer relation for $\mathbf{x}$. If $\dim(L_{\mathbf{x}}) = 1$, then the terminal value of s must be $n-1$, and thus the Shortest Relations Algorithm proves that the output $\mathbf{c}_n$ is a shortest vector in $L_{\mathbf{x}}$. In case $\dim(L_{\mathbf{x}}) = 1$, the Basic Integer Relation Algorithm also finds a shortest integer relation for $\mathbf{x}$ but it does not prove that $\mathbf{c}_n$ is shortest since $\dim(L_{\mathbf{x}})$ may be unknown.

(ii) The Shortest Relations Algorithm reduces the problem of finding a shortest (short, respectively) integer relation for $\mathbf{x}$ to the problem of finding a shortest (short, respectively) vector in the lattice $L = \langle \mathbf{c}_{s+1}, \cdots, \mathbf{c}_n \rangle$ where $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$ are the output vectors of the Shortest Relations Algorithm. We have shown that these vectors are already reasonably short. By applying Lovász basis reduction to the vectors $\mathbf{c}_{s+1}, \cdots, \mathbf{c}_n$ we obtain an integer relation $\mathbf{m}$ for $\mathbf{x}$ satisfying $\|\mathbf{m}\|^2 \leqq 2^{n-s-1}\lambda(\mathbf{x})^2$. A shortest integer relation for $\mathbf{x}$ can be found by Kannan's Algorithm *Shortest* that performs Korkine-Zolotareff reduction, see Kannan (1983). An improved version of Kannan's Algorithm and a hierarchy of polynomial time lattice basis reduction algorithms stretching from LLL-reduction toward Korkine-Zolotareff reduction has been given by Schnorr (1987).

5. Finding simultaneous integer relations. We describe an algorithm used to find simultaneous integer relations for real vectors $\mathbf{x}_1, \cdots, \mathbf{x}_l \in \mathbb{R}^n$, i.e., we search for linearly independent nonzero vectors $\mathbf{m} \in \mathbb{Z}^n$ such that $\langle \mathbf{x}_i, \mathbf{m} \rangle = 0$ for $i = 1, \cdots, l$. The *Simultaneous Relations Algorithm* resembles the Small Integer Relation Algorithm, but uses a more general exchange rule and a corresponding termination test. The Simultaneous Relations Algorithm includes the Small Integer Relation Algorithm and the Several Relations Algorithm as special cases.

Let the vectors $\mathbf{x}_1, \cdots, \mathbf{x}_l$ be fixed, and for a basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of lattice $\mathbb{Z}^n$ let $\mathbf{b}_j^*$ be the component of $\mathbf{b}_j$ that is orthogonal to $\mathbf{x}_1, \cdots, \mathbf{x}_l, \mathbf{b}_1, \cdots, \mathbf{b}_{j-1}$. The number of integers i, such that $\mathbf{b}_i^* = \mathbf{0}$, is the rank of the matrix $[\mathbf{x}_1, \cdots, \mathbf{x}_l]$. The general exchange rule is as follows.

GENERAL EXCHANGE RULE. Choose i with $1 \leqq i \leqq s$ that maximizes $2^{\tau_i}\|\mathbf{b}_i^*\|^2$ where $\tau_i = \#\{j: 1 \leqq j < i, \mathbf{b}_j^* \neq \mathbf{0}\}$ and $s = \max\{i \mid \mathbf{b}_i^* = \mathbf{0}\}$.

It includes the previous exchange rules, including the Bergman exchange rule, as special cases.

Let $L(\mathbf{x}_1, \cdots, \mathbf{x}_l)$ denote the lattice of simultaneous integer relations in $\mathbb{Z}^n$ for $\mathbf{x}_1, \cdots, \mathbf{x}_l$, i.e., $L(\mathbf{x}_1, \cdots, \mathbf{x}_l) = \{\mathbf{m} = (m_1, \cdots, m_n) \in \mathbb{Z}^n \mid \langle \mathbf{m}, \mathbf{x}_i \rangle = 0 \text{ for } 1 \leqq i \leqq l\}$. This lattice has rank $t = t(\mathbf{x}_1, \cdots, \mathbf{x}_l)$ where t may take any value in the range from 0 to $n - \operatorname{rank}[\mathbf{x}_1, \cdots, \mathbf{x}_l]$. Let $\lambda_i(\mathbf{x}_1, \cdots, \mathbf{x}_l)$ denote the ith successive minimum of the lattice $\langle \mathbf{x}_1, \cdots, \mathbf{x}_l \rangle$ for $1 \leqq i \leqq t$ and set $\lambda_i(\mathbf{x}_1, \cdots, \mathbf{x}_l) = \infty$ for $t+1 \leqq i \leqq n$.

SIMULTANEOUS RELATIONS ALGORITHM. (On input the real vectors $\mathbf{x}_1, \cdots, \mathbf{x}_l \in \mathbb{R}^n$ and $k, r \in \mathbb{N}$ this algorithm either finds r independent simultaneous integer relations $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ for $\mathbf{x}_1, \cdots, \mathbf{x}_l$ or it proves $\lambda_r(\mathbf{x}_1, \cdots, \mathbf{x}_l) \geqq 2^k$.)

1. *Initiation.* For the standard basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of $\mathbb{Z}^n$ compute the Gram-Schmidt quantities $\|\mathbf{b}_i^*\|^2$, $\mu_{i,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle \|\mathbf{b}_j^*\|^{-2}$ for $1 \leqq i, j \leqq n$, where $\mathbf{b}_i^*$ is the component of $\mathbf{b}_i$ that is orthogonal to $\mathbf{x}_1, \cdots, \mathbf{x}_l, \mathbf{b}_1, \cdots, \mathbf{b}_{i-1}$. Set $s := \max\{i: \mathbf{b}_i^* = \mathbf{0}\}$.
2. *Termination test.* If $s < n - r + 1$ then r simultaneous integer relations are found. Size-reduce the basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$. Compute $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^{\mathrm{T}} := [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$, output $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$, and stop. If $\|\mathbf{b}_i^*\| < 2^{-k}$ for $1 \leqq i \leqq n - r + 1$, then output "$\lambda_r(\mathbf{x}_1, \cdots, \mathbf{x}_l) \geqq 2^k$" and stop.
3. *Exchange step.* Choose i with $1 \leqq i \leqq s - 1$ that maximizes $2^{\tau_i}\|\mathbf{b}_i^*\|^2$ where $\tau_i = \#\{j \mid 1 \leqq j \leqq i, \mathbf{b}_j^* \neq \mathbf{0}\}$. Size-reduce $\mathbf{b}_{i+1}$ with respect to $\mathbf{b}_i$ by setting $\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,i} \rfloor \mathbf{b}_i$. Exchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$. Update the Gram-Schmidt quantities

$\|\mathbf{b}_\nu^*\|^2$, $\mu_{j,\nu}$, $\mu_{\nu,j}$ for $\nu = i$, $i+1$ and $j = 1, \cdots, n$. If $i = s-1$ and $\mathbf{b}_i^{\text{new}^*} = \mathbf{0}$, set $s := s-1$. Go to 2.

We prove the following result.

THEOREM 5.1. *When given vectors $\mathbf{x}_1, \cdots, \mathbf{x}_l \in \mathbb{R}^n$ and k, $r \in \mathbb{N}$ as input, the Simultaneous Relations Algorithm either finds r independent simultaneous integer relations $\mathbf{c}_{n+r+1}, \cdots, \mathbf{c}_n$ for $\mathbf{x}_1, \cdots, \mathbf{x}_l$, with*

$$\|\mathbf{c}_n\|^2 \leqq 2^{n-d+1}\lambda_1(\mathbf{x}_1, \cdots, \mathbf{x}_l)^2,$$

where $d = \operatorname{rank}[\mathbf{x}_1, \cdots, \mathbf{x}_l]$, or else it proves $\lambda_r(\mathbf{x}_1, \cdots, \mathbf{x}_l) \geqq 2^k$. It halts after at most $O(n^3(k+n))$ arithmetic operations on real numbers.

In addition to the upper bound on $\|\mathbf{c}_n\|^2$ we can extend the results of Theorem 4.2. Using the proof of Theorem 4.2 the reader can verify that the following holds.

If the Simultaneous Relations Algorithm finds integer relations $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ then we have $\|\mathbf{c}_{n-i+1}\|^2 \leqq 1.5^{2i-2}2^{n-d+1}\lambda_i(\mathbf{x}_1, \cdots, \mathbf{x}_l)$ for $i = 1, \cdots, r$.

Proof. Correctness. Suppose first that the algorithm gives output $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$. We prove that $\langle \mathbf{x}_i, \mathbf{c}_{n-r+j}\rangle = 0$ for $i = 1, \cdots, l$ and $j = 1, \cdots, r$. Since $\mathbf{b}_{n-r+j}^* \neq \mathbf{0}$ for $1 \leqq j \leqq r$ we have $\dim(\operatorname{span}(\mathbf{x}_1, \cdots, \mathbf{x}_l, \mathbf{b}_1^*, \cdots, \mathbf{b}_{n-r}^*)) = n-r$ and this implies that $\mathbf{x}_1, \cdots, \mathbf{x}_l \in \operatorname{span}(\mathbf{b}_1, \cdots, \mathbf{b}_{n-r})$. The row vectors $\mathbf{c}_{n-r+1}, \cdots, \mathbf{c}_n$ of the inverse matrix

$$[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top = [\mathbf{b}_1, \cdots, \mathbf{b}_n]^{-1}$$

satisfy $\langle \mathbf{b}_i, \mathbf{c}_{n-r+j}\rangle = 0$ for $1 \leqq i \leqq n-r$ and $j = 1, \cdots, r$, which yields the desired result that the vectors $\mathbf{c}_{n-r+j}$ are simultaneous integer relations.

To prove the claim that

$$\|\mathbf{c}_n\|^2 \leqq 2^{n-d+1}\lambda_1(\mathbf{x}_1, \cdots, \mathbf{x}_l)^2 \tag{5.1}$$

we use the special case $r = 1$ of the inequality

$$\lambda_r(\mathbf{x}_1, \cdots, \mathbf{x}_l) \geqq 1/\max_{1 \leqq i \leqq n-r+1} \|\mathbf{b}_i^*\| \quad \text{for } 1 \leqq r \leqq n-1, \tag{5.2}$$

which is proved by exactly the same argument that proves (4.1). Now the vector $\mathbf{c}_n$ is given by the inverse of the matrix $[\mathbf{b}_1, \cdots, \mathbf{b}_n]$ that occurs just after the last exchange $\mathbf{b}_{n-1} \leftrightarrow \mathbf{b}_n$. This exchange produces $\mathbf{b}_n^* \neq \mathbf{0}$, and the vectors $\mathbf{b}_{n-1}$ and $\mathbf{b}_n$ are never exchanged after that point and hence the last row $\mathbf{c}_n$ of the inverse matrix is never changed subsequently. Let $\mathbf{b}_1, \cdots, \mathbf{b}_n$ be the basis of $\mathbb{Z}^n$ before the last exchange $\mathbf{b}_n \leftrightarrow \mathbf{b}_{n-1}$. We have

$$\mathbf{c}_n = \pm\mathbf{b}_{n-1}^*\|\mathbf{b}_{n-1}^*\|^{-2}, \qquad \|\mathbf{c}_n\| = \|\mathbf{b}_{n-1}^*\|^{-1}. \tag{5.3}$$

The exchange rule says that the numbers $\|\mathbf{b}_i^*\|^2 2^{\tau_i}$ have a maximum at $i = n-1$, and we have $\tau_{n-1} - \tau_i \leqq n - \operatorname{rank}[\mathbf{x}_1, \cdots, \mathbf{x}_l] + 1 = n-d+1$ for $i = 1, \cdots, n-1$. This implies

$$\|\mathbf{b}_i^*\|^2 \leqq 2^{\tau_{n-1}-\tau_i}\|\mathbf{b}_{n-1}^*\|^2 \leqq 2^{n-d+1}\|\mathbf{b}_{n-1}^*\|^2 \quad \text{for } i = 1, \cdots, n-1$$

and thus

$$\|\mathbf{b}_{n-1}^*\|^{-2} \leqq 2^{n-d+1}\|\mathbf{b}_i^*\|^{-2}.$$

Therefore the claim (5.1) follows from

$$\lambda_1(\mathbf{x}_1, \cdots, \mathbf{x}_l)^2 \overset{(5.2)}{\geqq} 1/\max_{1 \leqq i \leqq n-1} \|\mathbf{b}_i^*\|^2 \geqq 2^{-n+d-1}\|\mathbf{b}_{n-1}^*\|^{-2} \overset{(5.3)}{=} 2^{-n+d-1}\|\mathbf{c}_n\|^2.$$

The correctness of the part of the termination test asserting "$\lambda_r(\mathbf{x}_1, \cdots, \mathbf{x}_l) \geqq 2^k$" follows immediately from (5.2).

Running time bound. We measure the progress of the algorithm using the quantity

$$D = \prod_{i=1}^{n} \max\{\|\mathbf{b}_i^*\|^2 2^n, 2^{-2k}\}^{n-i}.$$

We claim that at any exchange step we have

$$D_{\text{new}} \leqq \tfrac{3}{4} D_{\text{old}}. \tag{5.4}$$

This has been proved in the case $\tau_i = i$ for all $i \leqq n-1$, by Lemma 3.3. We prove (5.4) analogously to Lemma 3.3 with the change $\alpha(\mathbf{b}_i^*) = \max\{\|\mathbf{b}_i^*\|^2 2^n, 2^{-2k}\}$. If the Simultaneous Relation Algorithm exchanges $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$, then $\|\mathbf{b}_i^*\| \neq 0$, $\tau_{i+1}^{\text{old}} = \tau_i^{\text{old}} + 1$ and $\tau_j^{\text{new}} \leqq \tau_j^{\text{old}}$ for all j. Since i has been chosen to maximize the number $\|\mathbf{b}_j^*\|^2 2^{\tau_j}$ for $1 \leqq j < s$ we have $\|\mathbf{b}_{i+1}^{\text{old}*}\|^2 \leqq \frac{1}{2}\|\mathbf{b}_i^{\text{old}*}\|^2$, and thus the inequality

$$\|\mathbf{b}_i^{\text{new}*}\|^2 \leqq \tfrac{3}{4}\|\mathbf{b}_i^{\text{old}*}\|^2 \tag{3.2}$$

still holds. Since the algorithm did not previously terminate in step 2 we have $\|\mathbf{b}_j^{\text{old}*}\| \geqq 2^{-k}$ for some $j < s$ so that the inequality

$$2^n\|\mathbf{b}_i^{\text{old}*}\|^2 \geqq 2^{\tau_i+1}\|\mathbf{b}_i^{\text{old}*}\|^2 \geqq 2^{\tau_j+1}\|\mathbf{b}_j^{\text{old}*}\|^2 \geqq 2^{-2k+\tau_j+1} \geqq 2^{-2k+1} \tag{3.3}$$

still holds. Now the claim (5.4) follows by imitating the rest of the proof of Lemma 3.3.

Since we have $D \leqq 2^{n^3}$ at the start of the algorithm, while $D \geqq 2^{-kn^2}$ always holds, it follows that the number of exchange steps is at most $O(n^2(k+n))$, hence $O(n^3(k+n))$ arithmetic operations are used. □

Remarks. (i) A particular instance of the simultaneous relation problem is the problem of finding the minimal polynomial of an algebraic number. Let $\alpha = (\text{Re}(\alpha), \text{Im}(\alpha)) \in \mathbb{C}$ be an algebraic number with degree at most n. Then every simultaneous relation $m = (m_0, \cdots, m_n)$ for $x_1 = (\text{Re}(\alpha^0), \cdots, \text{Re}(\alpha^n))$, $x_2 = (\text{Im}(\alpha^0), \cdots, \text{Im}(\alpha^n))$ yields a multiple $p(x) = \sum_{i=0}^{n} m_i x^i$ of the minimal polynomial for α. The minimal polynomial can be found by repeating this process with smaller values of n. The degree of the minimial polynomial is the smallest n for which a simultaneous relation exists. Algorithms which determine the minimial polynomial of an algebraic number have been proposed by Schönhage (1984) and Kannan, Lenstra, and Lovász (1984, 1988).

(ii) Another instance of the simultaneous relation problem is to find an *integer dependency* $\mathbf{m} = (m_1, \cdots, m_s) \in \mathbb{Z}^s$ for the real vectors $\mathbf{y}_1, \cdots, \mathbf{y}_s \in \mathbb{R}^n$, i.e., a nonzero integer vector $\mathbf{m}$ satisfying $\sum_{i=1}^{s} m_i \mathbf{y}_i = 0$. An integer dependency for $\mathbf{y}_1, \cdots, \mathbf{y}_s$ is a simultaneous relation for the row vectors $\mathbf{x}_1, \cdots, \mathbf{x}_n$ of the matrix with column vectors $\mathbf{y}_1, \cdots, \mathbf{y}_s$, i.e., $[\mathbf{x}_1, \cdots, \mathbf{x}_n]^\top = [\mathbf{y}_1, \cdots, \mathbf{y}_s]$.

6. Finding short integer relations among integers. The problem to find short integer relations for an integer vector $\mathbf{x} \in \mathbb{Z}^n$ is particularly interesting since it is closely related to the *Knapsack problem*, which is known to be NP-complete. The *Knapsack problem* is to decide for given integers $a_1, \cdots, a_n, b$ whether the equation $\sum_{i=1}^{n} a_i m_i = b$ has a $\{0, 1\}$-solution $(m_1, \cdots, m_n) \in \{0, 1\}^n$. This means to decide whether there is a $\{0, 1\}$-relation $\mathbf{m} \in \{0, 1\}^{n+1}$ for the integer vector $\mathbf{x} = (a_1, \cdots, a_n, -b) \in \mathbb{Z}^{n+1}$ satisfying $m_{n+1} = 1$. Polynomial time algorithms for integer programming with a fixed number of variables have been given by Lenstra (1983) and Kannan (1983), (1987).

While we cannot expect to find a $\{0, 1\}$-relation in polynomial time, it is shown that we can find a basis of the lattice $L_{\mathbf{x}}$ of integer relations in polynomial time. The

following algorithm when given $\mathbf{x}$ as input finds a basis of short vectors of $L_{\mathbf{x}}$ within $O(n^3 \log \|\mathbf{x}\|)$ arithmetic operations in $O(n+\log \|\mathbf{x}\|)$-bit integers. It essentially applies the original Lovász algorithm to the set of linearly dependent vectors $\mathbf{b}_0, \mathbf{b}_1, \cdots, \mathbf{b}_n$ where $\mathbf{b}_0$ is the primitive vector $\mathbf{x}/\gcd(x_1, \cdots, x_n)$ and $\mathbf{b}_1, \cdots, \mathbf{b}_n$ is the standard basis of $\mathbb{Z}^n$. It uses the Lovász exchange rule, which exchanges $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$ for the smallest i such that $\|\mathbf{b}_i^*\|^2 > 2\|\mathbf{b}_{i+1}^*\|^2$. Here the vector $\mathbf{b}_i^*$ is the component of $\mathbf{b}_i$ that is orthogonal to $\mathbf{b}_0, \cdots, \mathbf{b}_{i-1}$. Now the Lovász exchange rule is efficient since the numbers $\|\mathbf{b}_i^*\|^2$ cannot become arbitrarily small and nonzero. We analyze this algorithm in the bit complexity model. This analysis would not work when using Bergman's exchange rule. Because of its great similarity to the original Lovász algorithm we call this algorithm the *Lovász Integer Relation Algorithm.*

LOVÁSZ INTEGER RELATION ALGORITHM. (On input $\mathbf{x} \in \mathbb{Z}^n$ this algorithm finds a basis $\mathbf{c}_2, \cdots, \mathbf{c}_n$ of lattice $L_{\mathbf{x}}$ such that $\|\mathbf{c}_{n-i+1}\|^2 \leqq 1.5^{2i-2}2^{n-1}\lambda_i(\mathbf{x})^2$ for $i = 1, \cdots, n-1$.)

1. *Initiation.* $\mathbf{b}_0 := \mathbf{x}/\gcd(x_1, \cdots, x_n)$.
   For the standard basis $\mathbf{b}_1, \cdots, \mathbf{b}_n$ of $\mathbb{Z}^n$ compute the Gram-Schmidt numbers $\|\mathbf{b}_i^*\|^2$, $\mu_{i,j}$ for $0 \leqq n$, $i := 1$ (i is the *stage*).
2. *Termination test.* If $i = n$ then
   $([\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top := [\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n]^{-1}$, output $(\mathbf{c}_2, \cdots, \mathbf{c}_n)$ and stop).
3. *Reduction in size.* For $j = i$ down to 0 do
   $(\mathbf{b}_{i+1} := \mathbf{b}_{i+1} - \lceil \mu_{i+1,j} \rfloor \mathbf{b}_j$, for $\nu = 0, \cdots, j$ do $\mu_{i+1,\nu} := \mu_{i+1,\nu} - \lceil \mu_{i+1,j} \rfloor \mu_{j,\nu})$.
   If $\|\mathbf{b}_i^*\|^2 \leqq 2\|\mathbf{b}_{i+1}^*\|^2$ then ($i := i+1$, goto 2).
4. *Exchange step.* Exchange $\mathbf{b}_i$ and $\mathbf{b}_{i+1}$. Update $\|\mathbf{b}_\nu^*\|^2$ and $\mu_{\nu,j}$ for $\nu = i, i+1$ and $j = 0, \cdots, i$.
   If $i > 1$ then $i := i-1$.
   Goto 3.

Note that this algorithm leaves the vector $\mathbf{b}_0$ fixed throughout and that it performs a complete size-reduction after each exchange step, which seems necessary to obtain good running time bounds in the bit complexity model.

THEOREM 6.1. *When given an integer vector $\mathbf{x} \in \mathbb{Z}^n$, the Lovász Integer Relations Algorithm finds a basis $\mathbf{c}_2, \cdots, \mathbf{c}_n$ of the lattice $L_{\mathbf{x}}$ satisfying $\|\mathbf{c}_{n-i+1}\|^2 \leqq 1.5^{2i-2}2^{n-1}\lambda_i(\mathbf{x})^2$ for $i = 1, \cdots, n-1$. Here $\lambda_1(\mathbf{x}), \cdots, \lambda_{n-1}(\mathbf{x})$ are the successive minima of the lattice $L_{\mathbf{x}}$ of integer relations for $\mathbf{x}$. It terminates after at most $O(n^3 \log \|\mathbf{x}\|)$ arithmetic operations on $O(n+\log \|\mathbf{x}\|)$ bit integers.*

Proof. We show that the output vectors $\mathbf{c}_2, \cdots, \mathbf{c}_n$ form a basis of lattice $L_{\mathbf{x}}$. Throughout the computation the vectors $\mathbf{b}_0, \mathbf{b}_1, \cdots, \mathbf{b}_n$ generate the lattice $\mathbb{Z}^n$. The vector $\mathbf{b}_0$ is never exchanged. On termination we have $\|\mathbf{b}_i^*\|^2 \leqq 2\|\mathbf{b}_{i+1}^*\|^2$ for $i = 1, \cdots, n-1$. This implies $\mathbf{b}_1^* = \mathbf{0}$ and thus $\mathbf{b}_1 \in \operatorname{span}(\mathbf{b}_0)$. Since $\mathbf{b}_0$ is a primitive vector we see that the vectors $\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n$, on termination, form a basis of lattice $\mathbb{Z}^n$. Therefore the output vectors $\mathbf{c}_2, \cdots, \mathbf{c}_n$ that are the row vectors of the inverse matrix $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top = [\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n]^{-1}$ form a basis of the lattice $L_{\mathbf{x}}$.

The inequalities $\|\mathbf{c}_{n-i+1}\|^2 \leqq 1.5^{2i-2}2^{n-1}\lambda_i(\mathbf{x})^2$ for $i = 1, \cdots, n-1$ have been shown in Theorem 4.2. This proof is still valid since the equalities (4.3) and the inequalities (4.4), (4.5) hold for $r = n-1$, $s = 1$ and for the output vectors $\mathbf{b}_2, \cdots, \mathbf{b}_n$ of the Lovász Integer Relation Algorithm. While this is obvious for (4.3) and (4.4) we remark that the inequalities

$$\|\mathbf{b}_{n-i+1}^*\|^{-2} \leqq \lambda_i(\mathbf{x})^2 2^{n-i-1} \quad \text{for } i = 1, \cdots, n-1 \tag{4.5}$$

follow from

$$\lambda_i(\mathbf{x}) \overset{(4.1)}{\geqq} 1/\max_{j=1,\cdots,n-i+1} \|\mathbf{b}_j^*\|$$

and

$$\|\mathbf{b}_j^*\|^2 \leqq 2\|\mathbf{b}_{j+1}^*\|^2 \quad \text{for } j = 1, \cdots, n-1.$$

Running time bound. We measure the progress of the algorithm using the quantity

$$D := \prod_{i=1, \mathbf{b}_i^* \neq \mathbf{0}}^{n} \|\mathbf{b}_i^*\|^{2(n-\tau(i))} \quad \text{where } \tau(i) = \#\{j \mid 1 \leqq j \leqq i, \mathbf{b}_j^* \neq \mathbf{0}\}.$$

We first show that every exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ achieves

$$D_{\text{new}} \leqq \tfrac{3}{4} D_{\text{old}} \quad \text{if } \mathbf{b}_i^{\text{new}^*} \neq \mathbf{0}, \tag{6.1}$$

$$D_{\text{new}} = D_{\text{old}} \quad \text{if } \mathbf{b}_i^{\text{new}^*} = \mathbf{0}. \tag{6.2}$$

If $\mathbf{b}_i^{\text{new}^*} \neq \mathbf{0}$, then (6.1) follows from (3.2) $\|\mathbf{b}_i^{\text{new}^*}\|^2 \leqq \frac{3}{4}\|\mathbf{b}_i^{\text{old}^*}\|^2$. If $\mathbf{b}_i^{\text{new}^*} = \mathbf{0}$, then $\mathbf{b}_{i+1}^{\text{new}^*} = \mathbf{b}_i^{\text{old}^*}$ and we have $D_{\text{new}} = D_{\text{old}}$. Obviously there is for each i at most one exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ such that $\mathbf{b}_i^{\text{new}^*} = \mathbf{0}$.

Upon entry of the algorithm the number D is at most 1, and now we bound from below the number D on termination. We know that $\mathbf{b}_1^* = \mathbf{0}$ holds on termination and thus

$$D = \prod_{i=2}^{n} \|\mathbf{b}_i^*\|^{2(n-i+1)}.$$

Since the vectors $\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n$ generate the lattice $\mathbb{Z}^n$ we have

$$1 = \|\mathbf{b}_0\| \prod_{i=2}^{n} \|\mathbf{b}_i^*\|.$$

It can easily be seen that the inequalities $\|\mathbf{b}_i^*\| \leqq 1$ for $i = 2, \cdots, n$ hold throughout the computation. This implies

$$\prod_{i=2}^{j} \|\mathbf{b}_i^*\|^2 = \|\mathbf{b}_0\|^{-2} \prod_{i=j+1}^{n} \|\mathbf{b}_i^*\|^{-2} \geqq \|\mathbf{b}_0\|^{-2} \geqq \|\mathbf{x}\|^{-2}.$$

Thus, on termination we have

$$D = \prod_{j=2}^{n} \prod_{i=2}^{j} \|\mathbf{b}_i^*\|^2 \geqq \|\mathbf{b}_0\|^{-2n+2} \geqq \|\mathbf{x}\|^{-2n+2}.$$

These bounds and the inequalities (6.1), (6.2) show that the number of exchange steps is at most

$$n + (2n-2)\log_{4/3} \|\mathbf{x}\| = O(n \log \|\mathbf{x}\|).$$

Each exchange step costs $O(n^2)$ arithmetic operations, the reduction in size of $\mathbf{b}_{i+1}$ included. From this we see that the total number of arithmetic steps is at most $O(n^3 \log \|\mathbf{x}\|)$, which also covers the steps for the initial computation of the Gram-Schmidt numbers $\mu_{i,j}$, $\|\mathbf{b}_i^*\|^2$.

The size of the integers involved. For $j = 1, \cdots, n$ let $d_j = \prod_{i=0}^{j} \|\mathbf{b}_i^*\|^2 = \det[\langle \mathbf{b}_i, \mathbf{b}_l \rangle]_{0 \leqq i,l \leqq j}$ where i and l only range over integers satisfying $\mathbf{b}_i^*, \mathbf{b}_l^* \neq \mathbf{0}$. The integer d_j is the square of the determinant of the lattice $\langle \mathbf{b}_0, \cdots, \mathbf{b}_j \rangle$. It is known from Lenstra, Lenstra Jr., and Lovász (1982, formulae (1.28), (1.29)) that the vector $\mathbf{b}_j^* d_{j-1}$ and the numbers $\mu_{i,j} d_j$ are integral. Therefore all numbers involved with the Lovász

Integer Relations Algorithm are rational with denominator not larger than $\max_j d_j$. We have $\max_j d_j \leqq \|\mathbf{x}\|^2$ since $\|\mathbf{b}_0^*\| \leqq \|\mathbf{x}\|$ and $\|\mathbf{b}_i^*\| \leqq 1$ for $i = 1, \cdots, n$.

Next we show that the following inequalities always hold upon entry of stage i:

$$\|\mathbf{b}_0\| \leqq \|\mathbf{x}\|, \|\mathbf{b}_k^*\| \leqq 1 \quad \text{for } k = 1, \cdots, n, \tag{6.3}$$

$$\|\mathbf{b}_k\|^2 \leqq \|\mathbf{x}\|^2 + n \quad \text{for } k = 1, \cdots, n, \tag{6.4}$$

$$|\mu_{k,j}| \leqq \|\mathbf{x}\|^2 + n \quad \text{for } 0 \leqq j \leqq k \text{ and } i \leqq k \leqq n. \tag{6.5}$$

Proof of equation (6.3). Throughout the computation we have $\mathbf{b}_0 = \mathbf{x}/\gcd(x_1, \cdots, x_n)$ and $\max_k \|\mathbf{b}_k^*\| \leqq 1$. This is because an exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ does not increase the maximum of $\|\mathbf{b}_1^*\|, \cdots, \|\mathbf{b}_n^*\|$.

Proof of equation (6.4). Initially $\|\mathbf{b}_k\| \leqq 1$ holds for $k = 1, \cdots, n$. During the computation we only permute the vectors $\mathbf{b}_1, \cdots, \mathbf{b}_n$, and on stage i we reduce the vector $\mathbf{b}_{i+1}$ in size. After the reduction of $\mathbf{b}_{i+1}$ we have

$$\|\mathbf{b}_{i+1}\|^2 \leqq \sum_{j=0}^{i+1} \mu_{i+1,j}^2 \|\mathbf{b}_j^*\|^2 \overset{(6.3)}{\leqq} \frac{(\|\mathbf{x}\|^2 + i + 1)}{4}.$$

Proof of equation (6.5). We have

$$\begin{aligned} |\mu_{k,j}| &\leqq \|\mathbf{b}_k\| \, \|\mathbf{b}_j^*\|^{-1} \\ &\overset{(6.4)}{\leqq} (\|\mathbf{x}\|^2 + n)^{1/2} (d_{j-1}/d_j)^{1/2} \\ &\leqq (\|\mathbf{x}\|^2 + n)^{1/2} d_{j-1}^{1/2} \leqq \|\mathbf{x}\|^2 + n. \end{aligned}$$

We next prove bounds for the other integers occurring with the algorithm. During the reduction of the vector $\mathbf{b}_{i+1}$ at stage i the numbers $\mu_{i+1,l}$ are transformed as

$$\mu_{i+1,l} := \mu_{i+1,l} - \lceil \mu_{i+1,j} \rfloor \mu_{j,l} \quad \text{for } l = 0, \cdots, j-1$$

for $j = i$ down to 0. Since $|\mu_{j,l}| \leqq \frac{1}{2}$ these steps at most double the number $M := \max_l |\mu_{i+1,l}|$ for each j. From (6.5) we know that $M \leqq \|\mathbf{x}\|^2 + n$, when the reduction of $\mathbf{b}_{i+1}$ starts, and thus $M \leqq 2^n(\|\mathbf{x}\|^2 + n)$ holds throughout the reduction of $\mathbf{b}_{i+1}$. So far we see that the numerators and denominators of the rational numbers d_j, $\mu_{i,j}$, $\|\mathbf{b}_j^*\|^2$ are at most $2^n(\|\mathbf{x}\|^2 + n)\|\mathbf{x}\|^2$ in absolute value.

The cost of the final matrix inversion. We consider the computation of the output vectors $\mathbf{c}_2, \cdots, \mathbf{c}_n$ along the matrix inversion $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top := [\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n]^{-1}$. We know from (4.3) that

$$\|\mathbf{c}_{n-i+1}\|^2 = \sum_{j \geqq n-i+1} v_{j,n-i+1}^2 \|\mathbf{b}_j^*\|^{-2} \quad \text{for } i = 1, \cdots, n-1$$

holds with $|v_{j,n-i+1}| \leqq 1.5^n$ and $\|\mathbf{b}_j^*\|^{-2} = d_{j-1}/d_j \leqq d_{j-1} \leqq \|\mathbf{x}\|^2$. This implies $\|\mathbf{c}_{n-i+1}\| \leqq \sqrt{n} 1.5^n \|\mathbf{x}\|$ for $i = 1, \cdots, n-1$.

This shows that the coordinates of the output vectors $\mathbf{c}_2, \cdots, \mathbf{c}_n$ have at most $\lceil \log_2(\sqrt{n} 1.5^n \|\mathbf{x}\|) \rceil$ bits. Therefore it is sufficient to compute the inverse matrix $[\mathbf{c}_1, \cdots, \mathbf{c}_n]^\top = [\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n]^{-1}$ modulo 2^e with $e = 1 + \lceil \log_2(\sqrt{n} 1.5^n \|\mathbf{x}\|) \rceil$. The inverse matrix $[\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n]^{-1}$ exists modulo 2^e since the vectors $\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n$ generate the lattice $\mathbb{Z}^n$, and thus $\det[\mathbf{b}_0, \mathbf{b}_2, \cdots, \mathbf{b}_n] = 1$. This matrix inversion can be done using $O(n^3)$ arithmetic steps (i.e., additions, multiplications, divisions) modulo 2^e. The matrix inversion requires at most n divisions modulo 2^e and each division can be done via the extended Euclidean algorithm using $O(e)$ arithmetic steps with $O(e)$-bit integers. From this we see that the matrix inversion costs at most $O(n^3 + n \log \|\mathbf{x}\|)$ arithmetical steps with $O(n + \log \|\mathbf{x}\|)$-bit integers. Thus the cost of the matrix inversion is covered by the time bound of Theorem 6.1. □

It is an interesting idea to replace in the above algorithm the Lovász exchange rule by Bergman's exchange rule. Will the resulting algorithm be equally efficient? The inequalities (6.1), (6.2) still hold for every exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$, and thus the number of arithmetic operations is still $O(n^3 \log \|\mathbf{x}\|)$. Moreover, the inequalities (6.3), (6.4), and (6.5) still hold after each exchange $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$ using Bergman's exchange rule. However, during size-reduction of the vector $\mathbf{b}_{i+1}$ where the numbers $\mu_{i+1,l}$ are transformed by

$$\mu_{i+1,l} := \mu_{i+1,l} - \lceil \mu_{i+1,j} \rfloor \mu_{j,l} \quad \text{for } l = 0, \cdots, j-1$$

from $j = i$ down to 0, the vector $\mathbf{b}_j$ need not to be size-reduced. So we only know $|\mu_{j,l}| \leqq \|\mathbf{x}\|^2 + n$ instead of $|\mu_{j,l}| \leqq \frac{1}{2}$ in case of the Lovász exchange rule. Thus in case of Bergman's exchange rule we only see that

$$\max_l |\mu_{i+1,l}| \leqq (\|\mathbf{x}\|^2 + n)^{n+1} \|\mathbf{x}\|^2$$

holds throughout size-reduction of the vector $\mathbf{b}_{i+1}$.

We can obtain upper bounds for the successive minima $\lambda_1(\mathbf{x}), \cdots, \lambda_{n-1}(\mathbf{x})$ of the lattice $L_{\mathbf{x}}$ in terms of $\mathbf{x}$ and Hermite's constant γ_{n-1}. For each n Hermite's constant γ_n is the maximal value of $\lambda_1(L)^2 \det(L)^{-2/n}$ where l ranges over all lattices of rank n, $\lambda_1(L)$ is the first successive minimum, and $\det(L)$ the determinant of the lattice L. It is known that $\gamma_n \leqq (4/\pi)\Gamma(1 + n/2)^{2/n}$ (see Cassels (1971, Chap. IX.7)), and this implies $\gamma_n \leqq \frac{2}{3}n$ for $n \geqq 2$.

THEOREM 6.2. *Let $\mathbf{x} \in \mathbb{Z}^n$ be a nonzero integer vector, $L_{\mathbf{x}}$ the lattice of integer relations for $\mathbf{x}$ and $\lambda_1(\mathbf{x}), \cdots, \lambda_{n-1}(\mathbf{x})$ the successive minima of lattice $L_{\mathbf{x}}$. We have*

(1) $\det L_{\mathbf{x}} = \|\mathbf{x}\| / \gcd(x_1, \cdots, x_n)$;

(2) $\lambda_1(\mathbf{x}) \leqq \sqrt{\gamma_{n-1}} (\|\mathbf{x}\| / \gcd(x_1, \cdots, x_n))^{1/(n-1)}$;

(3) $\prod_{i=1}^{n-1} \lambda_i(\mathbf{x}) \leqq \gamma_{n-1}^{(n-1)/2} \|\mathbf{x}\| / \gcd(x_1, \cdots, x_n) \leqq (n-1)^{(n-1)/2} \|\mathbf{x}\| / \gcd(x_1, \cdots, x_n)$.

Proof. (1) The rank of lattice $L_{\mathbf{x}}$ is $n-1$. Let $\mathbf{c}_1, \cdots, \mathbf{c}_n$ be a basis of the lattice $\mathbb{Z}^n$ such that $\mathbf{c}_2, \cdots, \mathbf{c}_n$ is a basis of the lattice $L_{\mathbf{x}}$. (Every basis $\mathbf{c}_2, \cdots, \mathbf{c}_n$ of $L_{\mathbf{x}}$ can be extended to a basis of $\mathbb{Z}^n$ since $\operatorname{span}(L_{\mathbf{x}}) \cap \mathbb{Z}^n = L_{\mathbf{x}}$.) The component of $\mathbf{c}_1$ that is orthogonal to $\mathbf{c}_2, \cdots, \mathbf{c}_n$ is $\langle \mathbf{c}_1, \mathbf{x} \rangle \|\mathbf{x}\|^{-2} \mathbf{x}$ and has length $\langle \mathbf{c}_1, \mathbf{x} \rangle \|\mathbf{x}\|^{-1}$. This implies

$$1 = \det \mathbb{Z}^n = (\det L_{\mathbf{x}}) \quad |\langle \mathbf{c}_1, \mathbf{x} \rangle| \, \|\mathbf{x}\|^{-1}.$$

On the other hand, $\langle \mathbf{c}_1, \mathbf{x} \rangle$ is the minimal positive integer in $\{\langle \mathbf{m}, \mathbf{x} \rangle \mid \mathbf{m} \in \mathbb{Z}^n\}$, and thus $\langle \mathbf{c}_1, \mathbf{x} \rangle = \gcd(x_1, \cdots, x_n)$. From this we see that $\det L_{\mathbf{x}} = \|\mathbf{x}\| / \gcd(x_1, \cdots, x_n)$. The inequalities (2) and (3) are direct consequences of (1) by the Minkowski inequality

$$\prod_{i=1}^{n-1} \lambda_i(L) \leqq \gamma_{n-1}^{(n-1)/2} \det L$$

where L is a lattice with dimension $n-1$ and with successive minima $\lambda_1(L), \cdots, \lambda_{n-1}(L)$ (see Cassels (1971), Chap. VIII.2)). □

The upper bound $\lambda(\mathbf{x}) \leqq \sqrt{\gamma_{n-1}} \|\mathbf{x}\|^{1/(n-1)}$ in Theorem 6.2 is optimal for rank $n = 2$. The shortest integer relation for $\mathbf{x} = (x_1, x_2) \in \mathbb{Z}^2$ is $(-x_2, x_1)/\gcd(x_1, x_2)$ that has length $\|\mathbf{x}\| / \gcd(x_1, x_2) = \sqrt{\gamma_1} \|\mathbf{x}\| / \gcd(x_1, x_2)$.

REFERENCES

L. BABAI, B. JUST, AND F. MEYER AUF DER HEIDE (1988), *On the limits of computations with the floor function*, Inform. Comput., 78, pp. 99-107.

G. BERGMAN (1980), *Notes on Ferguson and Forcade's Generalized Euclidean Algorithm*, unpublished paper, University of California, Berkeley, CA.

L. BERNSTEIN (1971), *The Jacobi-Perron Algorithm. Its Theory and Applications*, Lecture Notes in Mathematics 207, Springer-Verlag, Berlin, New York.

A. J. BRENTJES (1981), *Multi-dimensional Continued Fraction Algorithms*, Mathem. Centre Tracts No. 145, Mathem. Centrum, Amsterdam, the Netherlands.

J. W. S. CASSELS (1971), *An Introduction to the Geometry of Numbers*, Springer-Verlag, Berlin, New York.

P. VAN EMDE BOAS (1981), *Another* NP-*complete partition problem and the complexity of computing short vectors in a lattice*, Mathematics Department Report 81-04, University of Amsterdam, Amsterdam, the Netherlands.

H. R. P. FERGUSON (1986), *A short proof of the existence of vector Euclidean algorithms*, Proc. Amer. Math. Soc., 97, pp. 8-11.

———, (1987), *A non-inductive* GL $(n, \mathbb{Z})$ *algorithm that constructs integral linear relations for n* $\mathbb{Z}$*-linearly dependent real numbers*, J. Algorithms, 8, pp. 131-145.

H. R. P. FERGUSON AND R. W. FORCADE (1979), *Generalization of the Euclidean algorithm for real numbers to all dimensions higher than two*, Bull. Amer. Math. Soc., 1, pp. 912-914.

———, (1982), *Multidimensional Euclidean Algorithms*, J. Reine Angew. Math., 334, pp. 171-181.

A. FRANK AND E. TARDÖS (1985), *An application of simultaneous approximation in combinatorial optimization*, in Proc. 26th Annual IEEE Conference on Foundations of Computer Science, IEEE Press, pp. 459-463.

J. HASTAD, B. JUST, J. C. LAGARIAS, AND C. P. SCHNORR (1986), *Polynomial Time Algorithms for Finding Integer Relations Among Real Numbers*, in Proc. 3rd STACS, Paris, 1986, Lecture Notes in Computer Science 210, Springer-Verlag, Berlin, New York, pp. 105-118.

C. G. J. JACOBI (1868), *Allgemeine Theorie der Kettenbruchähnlichen Algorithmen*, J. Reine Angew. Math., 69, pp. 29-64.

B. JUST (1987), *Effiziente Kettenbruchalgorithmen in beliebigen Dimensionen*, Dissertation, Universität Frankfurt, Frankfurt, FRG.

R. KANNAN (1983), *Improved algorithms for integer programming and related lattice problems*, in Proc. 15th Annual ACM Symposium on Theory Computing, ACM Press, pp. 193-206.

———, (1987), *Minkowski's convex body theorem and integer programming*, Math. Oper. Res., 12, pp. 415-440.

R. KANNAN, A. K. LENSTRA, AND L. LOVÁSZ (1984, 1988), *Polynomial factorization and nonrandomness of bits of algebraic and some transcendental numbers*, in Proc. 16th Annual ACM Symposium on Theory of Computing, pp. 191-200; Math. Comp., 50, (1988), pp. 235-250.

A. K. LENSTRA, H. W. LENSTRA, JR., AND L. LOVÁSZ (1982), *Factoring polynomials with rational coefficients*, Math. Ann., 21, pp. 515-534.

H. W. LENSTRA, JR. (1983), *Integer programming with a fixed number of variables*, Math. Oper. Res., 8, pp. 538-548.

O. PERRON (1907), *Grundlagen für eine Theorie des Jacobischen Kettenbruchalgorithmus*, Math. Ann., 64, pp. 1-76.

C. P. SCHNORR (1987), *A hierarchy of polynomial time lattice basis reduction algorithms*, Theoret. Comput. Sci., 53, pp. 201-224.

———, (1986, 1988), *A more efficient lattice basis reduction algorithm*, extended abstract in Proc. ICALP, Rennes, France, 1986, Lecture Notes in Computer Science 226, Springer-Verlag, Berlin, New York, 1986, pp. 359-369; J. Algorithms, 9 (1988), pp. 47-62.

A. SCHÖNHAGE (1984), *Factorization of univariate integer polynomials by diophantine approximation and an improved basis reduction algorithm*, in Proc. 11th ICALP, Antwerpen, 1984, Lecture Notes in Computer Science 127, Springer-Verlag, Berlin, New York, 1984, pp. 436-447.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 882-892, October 1989

002

RANKING THE BEST BINARY TREES*

S. ANILY† AND R. HASSIN‡

Abstract. The problem of ranking the K-best binary trees with respect to their weighted average leaves' levels is considered. Both the alphabetic case, where the order of the weights in the sequence $w_1, \cdots, w_n$ must be preserved in the leaves of the tree, and the nonalphabetic case, where no such restriction is imposed, are studied.

For the alphabetic case a simple algorithm is provided for ranking the K-best trees based on a recursive formula of complexity $O(Kn^3)$. For nonalphabetic trees two different ranking problems are considered, and for each of them it is shown that the next best tree can be solved by a dynamic programming formula of low complexity order.

Key words. binary trees, alphabetic and nonalphabetic trees, ranking of solutions

AMS(MOS) subject classifications. 94B45, 68E99

1. Introduction. Let $w_1, \cdots, w_n$ be given "weights." This paper deals with the problem of computing the best, second best, $\cdots$, K-best binary trees with respect to these weights. The problem arises when we want to construct the best tree satisfying certain constraints, and no efficient algorithm is known to find this tree. We may then rank the best trees ignoring these additional constraints starting from the best to the next best until the best tree obeying the constraints is reached.

We consider both the alphabetic case, where the order the weights are given must be preserved in the leaves of the tree, and the nonalphabetic case where no such constraints are imposed. The techniques we present can be used however in other problems of ranking trees. For example, ranking binary search trees is done almost in the same way as for the alphabetic case.

Ranking alphabetic trees is relatively a straightforward task. The problem is solvable by dynamic programming, and thus partitioning of the solution set can be obtained by introducing constraints on the decisions made while executing the computations. In this regard the problem is similar to the well-solved problem of ranking the shortest paths between a pair of nodes in a network. In § 2 we show how this can be done efficiently, and the K-best trees can be computed in $O(Kn^3)$-time.

Nonalphabetic trees are useful in the context of binary encoding of a set of words where each word v_i has a given frequency w_i in which it appears in the language. In a given code each word is written as a string of zeros and ones, and the length of a word is defined as the length of the string. The main objective is to find a binary encoding of minimum average length. Here we distinguish between two different problems:

(a) The language is viewed as a collection of n objects (words); we say that two codes are different if there exists a word v_i, $1 \leqq i \leqq n$ that is associated with strings of different lengths in these codes (see § 5).

(b) Here we do not distinguish between words of identical weights, i.e., given a code for a language containing two words v_i and v_j for which $w_i = w_j$, then exchanging

* Received by the editors January 27, 1988; accepted for publication (in revised form) December 7, 1988.

† Faculty of Commerce and Business Administration, University of British Columbia, Vancouver, British Columbia, Canada V6T148. Present address, Faculty of Management, Tel-Aviv University, Ramat-Aviv, Tel-Aviv 69978, Israel. The research of this author was supported in part by National Science and Engineering Research Council of Canada grant A4082.

‡ Department of Statistics, Tel-Aviv University, Ramat-Aviv, Tel-Aviv 69978, Israel.

between the strings associated with v_i and v_j does **not** induce a new code even if their lengths (=levels) are different. In other words, the set of words $\{v_1, v_2, \cdots, v_n\}$ is partitioned into disjoint subsets according to their weights. A code is identified by the corresponding subsets of the strings' lengths where the order in which the levels of a particular subset are assigned to the words in that subset, is unimportant (see § 3).

We note that if all the weights are different from each other, then the two problems coincide; otherwise, the number of different codes is larger in the problem defined in (a). In both cases care must be taken to avoid repetition of solutions (where "repetition" is defined differently in the two cases), as a solution is uniquely identified by the length of the words and thus may be represented in many ways by different topological trees with different orderings of the weights.

Ranking the nonalphabetic trees is not as straightforward as ranking the alphabetic trees since no order is defined on the problem's elements. We note that the set of all alphabetic trees corresponding to a given order of leaves is only a subset (of a **much smaller** size) of the set of all nonalphabetic trees with the same number of leaves. (For example, in all alphabetic trees with three leaves v_1, v_2, and v_3 the second leaf of the trees is of level two while in the set of nonalphabetic trees using the same leaves, v_2 may also be of level one.) Therefore, the task of ranking the nonalphabetic trees cannot be achieved by applying the corresponding algorithm for alphabetic trees on any specific order of the leaves. Moreover, since the leaves can be ordered in $n!$ **different ways**, a direct application of the ranking procedure for alphabetic trees to the nonalphabetic case may result in an unefficient algorithm and an enormous number of repetitions of solutions. The main objective of this paper is in developing efficient ranking algorithms for nonalphabetic trees.

We show that the best nonalphabetic tree (i.e., the "Huffman tree") can be computed by any algorithm for **alphabetic** trees. We then extend this property to rank nonalphabetic trees using ranking schemes for alphabetic trees: in § 3 we introduce another algorithm for alphabetic trees (with a higher complexity order-$O(kn^4)$) that is modified in § 4 to rank the solutions for the nonalphabetic problem (b) defined above. In § 5, we present an $O(kn^3)$ algorithm that ranks the solutions for the nonalphabetic problem (a) by combining a procedure for ranking solutions for the assignment problem.

We assume that the reader is familiar with the basic concepts involved with binary trees as described, for example, in [K].

2. Alphabetic trees: Algorithm A. An alphabetic tree with n leaves $v_1, \cdots, v_n$ is represented by the sequence of levels of its leaves, ordered from left to right. We denote this sequence by $(l_1, \cdots, l_n)$. For a given sequence of **weights** we define the cost of the tree $T=(l_1, \cdots, l_n)$ as $C(T)=\sum_{i=1}^{n} w_i l_i$. The optimal tree, i.e., the one of minimum cost, can be found in $O(n \log n)$-time by the algorithm of Hu and Tucker [HT]; however, we do not know of any method that will use this algorithm to rank the K-best trees. In this section and in the next we describe, instead, two methods for ranking the best trees that are based on the recursive algorithm suggested by Gilbert and Moore [GM]. The first computes the K-best trees in $O(Kn^3)$-time by modifying the above algorithm in a way similar to that used by Dreyfus [Dre] and Lawler [L2] to rank the K shortest s-t paths in a network. The second requires $O(Kn^4)$-time and will serve later to rank the best (nonalphabetic) binary trees.

Let T_{ij}^k and C_{ij}^k denote the k-best tree and its cost for a problem consisting of the weights $w_i, w_{i+1}, \cdots, w_j$ and define $W_{ij}=\sum_{r=i}^{j} w_r$. Then $C_{ii}^1=0$ $i=1, \cdots, n$, and

$$C_{ij}^1=\min_{i\leq r\leq j-1}\{C_{ir}^1+C_{r+1,j}^1\}+W_{ij}, \qquad i<j. \tag{1}$$

For $k>1$, C_{ij}^k is given by $C_{ir}^u + C_{r+1,j}^v + W_{ij}$ for some $i \leqq r < j$ and $u, v \leqq k$. Thus C_{ij}^k is fully characterized by the triple (r, u, v) and we denote $T_{ij}^k = (r_{ij}^k, u_{ij}^k, v_{ij}^k)$.

Let

$$U(i, j, r, K) = \max \{u \mid r_{ij}^k = r \text{ and } u_{ij}^k = u, \text{ for some } k = 1, \cdots, K-1\},$$

$$\text{LAST}(i, j, r, K, u) = \max \{v \mid v_{ij}^k = v, u_{ij}^k = u, r_{ij}^k = r \text{ for some } k = 1, \cdots, K-1\}.$$

Both U and LAST are set to zero when the maximization is over an empty set. U and LAST focus on the subset of the $(K-1)$st-best solutions for $v_i, v_{i+1}, \cdots, v_j$ in which the left subtree consists of the leaves $v_i, v_{i+1}, \cdots, v_r$ and the right subtree consists of the leaves $v_{r+1}, \cdots, v_j$, i.e., the set $\{T_{ij}^k \mid T_{ij}^k = (r, u_{ij}^k, v_{ij}^k), k = 1, \cdots, K-1\}$. The operator U provides us with the maximum u_{ij}^k in the set that represents the rank value of the worse left subtree used among these solutions. The operator LAST, on the other hand, has an additional parameter u and is applied on a subset of the above set, namely, $\{T_{ij}^k \mid T_{ij}^k = (r, u, v_{ij}^k), k = 1, \cdots, K-1\}$ consisting only of those solutions using the uth best tree for $v_i, v_{i+1}, \cdots, v_r$ as their left subtree. LAST is assigned the maximum v_{ij}^k in this set, i.e., the rank value of the worse right subtree consisting of the leaves $v_{r+1}, \cdots, v_j$ used together with T_{ir}^u among the $(K-1)$st-best solutions for $v_i, \cdots, v_j$. The operators U and LAST are used in the design of Algorithm A.

Let

$$M_n = \frac{1}{n}\binom{2n-2}{n-1}$$

be the number of distinct alphabetic binary trees with n leaves [RH]. For a given sequence of weights $w_1, \cdots, w_n$ and $K \leqq M_n$, we propose an algorithm, based on a dynamic programming formulation that we explain in the sequel, for computing the K-best trees:

ALGORITHM A.
Compute C_{ij}^1 for all $1 \leqq i \leqq j \leqq n$ using recursion (1).
For $m = 1, \cdots, n-1$ do begin
For $i = 1, \cdots, n-m$ do begin
For $k = 2, \cdots, \min(K, M_{m+1})$ do

$$C_{i,i+m}^k = W_{i,i+m} + \min_{i \leqq r < i+m} \left\{ \min_{1 \leqq u \leqq U(i,i+m,r,k)+1} \{C_{ir}^u + C_{r+1,i+m}^{\text{LAST}(i,i+m,r,k,u)+1}\} \right\} \tag{2}$$

end
end

In (2) it is assumed that $C_{ij}^k = \infty$ for $k > M_{j-i+1}$. It is also assumed that some tie-breaking rule is applied whenever $C_{ij}^{k+1} = C_{ij}^k$. For example, we may require in such a case that either $r_{ij}^k < r_{ij}^{k+1}$, or $r_{ij}^k = r_{ij}^{k+1}$ and $u_{ij}^k < u_{ij}^{k+1}$, or $r_{ij}^k = r_{ij}^{k+1}$, $u_{ij}^k = u_{ij}^{k+1}$ and $v_{ij}^k < v_{ij}^{k+1}$.

Formula (2) can be explained as follows. $T_{i,i+m}^K$ can be viewed as combined of two subtrees emanating from its root; the left one consisting of the leaves $v_i, v_{i+1}, \cdots, v_r$ and the right one consisting of the leaves $v_{r+1}, \cdots, v_{i+m}$ for some $i \leqq r < i+m$. Among the $(K-1)$st-best trees for the leaves $v_i, \cdots, v_{i+m}$ consider only those combined of two subtrees in which v_r is the highest indexed leaf in the left subtree. The best such solution is, of course, consisting of the subtrees T_{ir}^1 and $T_{r+1,i+m}^1$. The second best such solution may either consist of the subtrees T_{ir}^2 and $T_{r+1,i+m}^1$ or the subtrees T_{ir}^1

and $T^2_{r+1,i+m}$, etc. By the same reason with respect to $T^K_{i,i+m}$ we distinguish between the following cases:

(a) $T^K_{i,i+m}$ consists of a subtree T^u_{ir} that already has been used in one of the trees $T^k_{i,i+m}$, $1 \leqq k \leqq K-1$, i.e., $u \leqq U(i, i+m, r, K)$. In that case the right subtree must be the **best** tree for $v_{r+1}, \cdots, v_{i+m}$ that has not been used previously together with T^u_{ir} in one of the solutions $T^k_{i,i+m}$, $1 \leqq k \leqq K-1$, thus the right subtree is given by $T^{\mathrm{LAST}(i,i+m,r,K,u)+1}_{r+1,i+m}$.

(b) If the left subtree T^u_{ir} has not been used in one of the solutions $T^k_{i,i+m}$, $1 \leqq k \leqq K-1$, then it is easily verified that u must be equal to $U(i, i+m, r, K)+1$ and the right subtree must be the optimal one, i.e., $T^1_{r+1,i+m}$. We observe that for $u > U$ the operator LAST is equal to zero by definition. In addition, since the costs of T^u_{ir} and $T^v_{r+1,i+m}$, i.e., C^u_{ir} and $C^v_{r+1,i+m}$ are computed relative to **their** roots, we must adjust the leaves' levels that results in adding the sum of the weights $W_{i,i+m}$ to $C^u_{ir}+C^v_{r+1,i+m}$.

Formula (2) follows immediately from the above observations.

Maintaining an appropriate data structure of $\{C^u_{ir}+C^{\mathrm{LAST}(i,i+m,r,K,u)+1}_{r+1,i+m} \mid r = i, \cdots, i+m-1, u = 1, \cdots, K\}$ for each (i, m) pair, the $O(K)$ minimizations in the inner loop require $O(n+K \log (n+K))$-time. Since $\log K \leqq \log M_n = O(n)$, the overall complexity of the algorithm is $O(Kn^3)$.

3. Alphabetic trees: Algorithm B. We do not know of any efficient modification of Algorithm A to solve for the best binary (nonalphabetic) trees. Next we describe an $O(Kn^4)$ algorithm for alphabetic trees, for which such a modification is possible as will be described in the next section. This algorithm that we call Algorithm B uses a partitioning procedure of the solution set, similar to that of Murty [M] and Lawler [L1]. (See also [KIM2], [KIM3] and the extensive bibliographies on ranking the K-best shortest paths.)

Let A_n be the set of alphabetic trees with n leaves.

Let $T^k = (l^k_1, \cdots, l^k_n)$ be the K-best tree in A_n. Suppose that T_1 is known. For $i = 1, \cdots, n-1$ let $S_i(S'_i)$ be the subset of A_n satisfying $l_1 = l^1_1, \cdots, l_{i-1} = l^1_{i-1}$, $l_i < l^1_i$ $(l_i > l^1_i)$. The union of all these sets is $A_n - \{T^1\}$, so that if we can solve for the best tree in each set then we can obtain T^2 by comparing these trees and selecting the one with minimum cost.

Suppose, without loss of generality, that $T^2 \in S_i$. To compute T^3 we first partition $S_i - \{T^2\}$ to subsets $R_j(R'_j)$ $j = i, \cdots, n-1$ satisfying new constraints in addition to those defining S_i. Specifically,

$$R_i = \{T \in A_n \mid l_1 = l^1_1, \cdots, l_{i-1} = l^1_{i-1}, l_i < l^2_i\},$$

$$R'_i = \{T \in A_n \mid l_1 = l^1_1, \cdots, l_{i-1} = l^1_{i-1}, l^2_i < l_i < l^1_i\}, \text{ and for } j = i+1, \cdots, n-1,$$

$$R_j = \{T \in A_n \mid l_1 = l^1_1, \cdots, l_{i-1} = l^1_{i-1}, l_i = l^2_i, \cdots, l_{j-1} = l^2_{j-1}, l_j < l^2_j\},$$

$$R'_j = \{T \in A_n \mid l_1 = l^1_1, \cdots, l_{i-1} = l^1_{i-1}, l_i = l^2_i, \cdots, l_{j-1} = l^2_{j-1}, l_j > l^2_j\}.$$

If we could solve for the best tree in each of these sets, then we could compute T^3 as the tree of minimum cost among these trees and the best trees of S'_i, S_j, and S'_j $j \neq i$. By repeating this procedure $K-1$ times we could compute the K-best trees in A_n.

To apply such a procedure we need an algorithm that computes the best tree satisfying constraints of the following type: $l_1 = \bar{l}_1, \cdots, l_{i-1} = \bar{l}_{i-1}$, $D_1 \leqq l_i \leqq D_2$. We

denote the set of trees satisfying these constraints by F_i. All these trees share a common left part consisting of the paths from the root to the $(i-1)$st most left leaves $v_1, \cdots, v_{i-1}$.

DEFINITION. The **front** of the trees in F_i consists of the subgraph of a tree $T \in F_i$ induced by the paths from the root to the i leftmost leaves, where the path to v_i is extended from the leaf so that its total length is D_2. This extended path is called the **stem** of the front.

We note that the front is uniquely defined and is independent of the choice of the tree T. The front is also independent of D_1, so that it is characterized by the level sequence $(\bar{l}_1, \cdots, \bar{l}_{i-1}, D_2)$.

The nodes on the front, except those on the stem, are either leaves or parents to two sons. The nodes on the stem fall into three categories:

(i) Parents to two sons. We call them **closed** nodes.

(ii) Parents to a single son. We call them **open** nodes.

(iii) The leaf. We classify it also as an open node.

We name the open nodes in order from the leaf to the root by $O_1, \cdots, O_m$ as in Fig. 1, and denote their levels by $l(O_1), \cdots, l(O_m)$, respectively.

Let $q = \max\{j \mid l(O_j) \geqq D_1$ and there are no closed nodes below $O_j\}$, i.e., O_q is the open node closest to the root whose level is at least D_1 such that all the closed nodes on the stem have lower levels. In other words, the ith leaf of the trees in F_i must be one of the nodes $O_q, O_{q-1}, \cdots, O_1$.

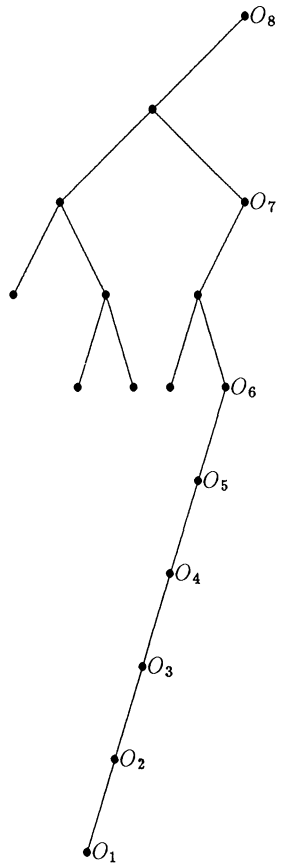


FIG. 1. *The front* (3, 4, 4, 4, 9).

Any tree in F_i is constructed from the front of F_i by assigning v_i to an open node O_p, $p \leqq q$, deleting $O_1, \cdots, O_{p-1}$, and joining $v_{i+1}, \cdots, v_n$ to $O_{p+1}, \cdots, O_m$ through **nonempty** alphabetic trees whose leaves are consecutive subsequences of $v_{i+1}, \cdots, v_n$, and whose roots are connected to the open nodes. For example, suppose $F_i = \{T \in A_n \mid l_1 = 3,\ l_2 = 4,\ l_3 = 4,\ l_4 = 4,\ 5 \leqq l_5 \leqq 9\}$. Then the front of F_i is as in Fig. 1, and $O_q = O_5$. Figure 2 illustrates the construction of a member of F_i from the front of F_i.

We now show how to solve the problem of computing the best tree in F_i by dynamic programming.

Let V_{jl}, $j \geqq 1$, $l \geqq 1$ denote the minimum cost involved with attaching leaves $v_i, \cdots, v_l$ to subtrees connected to $O_1, \cdots, O_j$. Let C_{rl} denote the cost of an optimal alphabetic tree with leaves $v_r, v_{r+1}, \cdots, v_l$. Note that if we attach this tree to the front through an edge from O_j to its root, then the actual cost is $C_{rl} + W_{rl}(l(O_j)+1)$. Therefore,

$$V_{jl} = \min_{i \leqq r \leqq l-1} \{V_{j-1,r} + C_{r+1,l} + W_{r+1,l}(l(O_j)+1)\}, \qquad l > i, \quad j > 2,$$

$$V_{1l} = \infty, \qquad l > i,$$

$$V_{ji} = \begin{cases} w_i l(Oj), & 1 \leqq j \leqq q, \\ \infty, & j > q. \end{cases} \tag{3}$$

The costs C_{rl}, $1 \leqq r < l \leqq n$ can be computed in $O(n^3)$-time by (1). Then (3) can be computed for a given front and q (determined by a lower bound D_1) and for all relevant j and l in $O(n^3)$.

To apply (3) we must first determine the levels of the open nodes on the stem. For this purpose we make the following definitions:

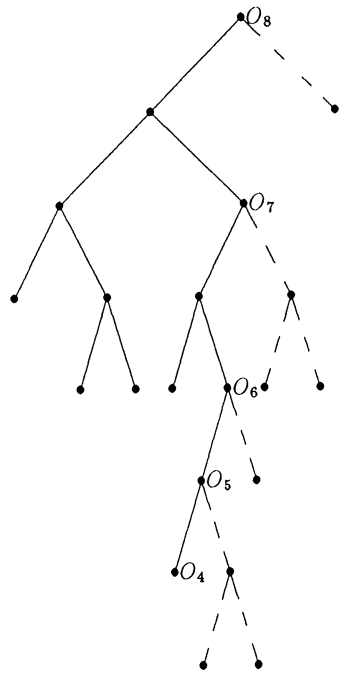


FIG. 2. *A possible completion for the tree in Fig.* 1.

For a sequence $a_1, \cdots, a_n$ let $k = \min\{i \mid a_{i-1} = a_i\}$. Then the sequence $a_1, \cdots, a_{i-2}, a_i - 1, a_{i+1}, \cdots, a_n$ is the **reduction from the left** of the original sequence. The **left-reduced sequence** for $a_1, \cdots, a_n$ is obtained by repeating the process of reduction from the left until $a_{i-1} \neq a_i$, $i = 2, \cdots, n$. For example, the sequence (3, 4, 4, 4) generates (3, 3, 4) and then (2, 4). No further reduction is possible and thus (2, 4) is the left reduced sequence.

LEMMA 1 [HT]. *A sequence $l_1, \cdots, l_n$ defines a tree in A_n if and only if its left reduced sequence is* (0).

In a similar way we can prove the following theorem.

THEOREM 2. *Let $(l_1^*, \cdots, l_p^*)$ be the left reduced sequence of $(\bar{l}_1, \cdots, \bar{l}_{i-1})$. Then*

(a) *F_i is nonempty if and only if $D_2 \geqq l_p^*$ and $\max\{D_1, l_p^*\} - p \leqq n - i + 1$.*

(b) *If F_i is nonempty then the stem of the front of the trees in F_i, defined by $(\bar{l}_1, \cdots, \bar{l}_{i-1}, D_2)$, contains exactly p closed nodes, at levels $l_1^* - 1, \cdots, l_p^* - 1$.*

The costs $C_{rl} \leqq r < l \leqq n$ used in (3) can be computed in $O(n^3)$-time by (1). For a given front the open nodes can be computed with the aid of Theorem 2, and q is determined by the front and the lower bound D_1. Then (3) can be computed for all relevant j and l in $O(n^3)$-time. For each value of k, $1 \leqq k \leqq K$ we apply (3) at most $2n$ times using the above partitioning procedure, and then select the best of the solutions obtained for the $O(Kn)$ subsets of the partition. Therefore the time required to compute the next best tree is $O(n^4 + n \log Kn) = O(n^4)$, and the overall complexity of computing the K-best trees is $O(Kn^4)$.

4. Nonalphabetic trees: Algorithm C. In this section and the subsequent one, we describe two algorithms for computing the K-best binary trees for the leaves $v_1, v_2, \cdots, v_n$. In contrast to the previous section, the order of the leaves is not prespecified. The optimal tree can be computed in $O(n \log n)$-time by the algorithm due to Huffman [Huf]. Alternatively, the leaves can be numbered in ascending order of their weights $w_1, \cdots, w_n$ and then the best alphabetic tree for $v_1, v_2, \cdots, v_n$ can be computed by the Hu-Tucker Algorithm [HT] that also requires $O(n \log n)$ time. It is known that the resulting tree is indeed optimal.

As mentioned in § 1 we consider two different ranking problems on the set of nonalphabetic trees. In this section we provide a ranking algorithm of complexity $O(Kn^4)$ according to (b) defined in § 1. In this problem a solution $(l_1, l_2, \cdots, l_n)$ is characterized by the set $\{(l_i, w_i) \mid 1 \leqq i \leqq n\}$, i.e., interchanging the levels of two words with identical weights will **not** create a new solution.

We first modify the partitioning algorithm of § 3 to rank nonalphabetic trees. To reduce the time complexity we modify the partitioning scheme so that when computing the next best solution each subset is replaced by just two new ones, rather than $O(n)$ new ones. As in [G], [Der], [KIM1] and [KIM3] this requires knowledge of both the best and the second-best solution in each subset of the partition.

Without loss of generality we assume that $w_1 \leqq w_2 \leqq \cdots \leqq w_n$ and let l_i denote the level of v_i. A **tree** is uniquely defined by the sequence $(l_1, \cdots, l_n)$ for which there exists a permutation $(l_1', \cdots, l_n')$ defining an alphabetic tree. It is well known that a necessary and sufficient condition for integers $(l_1, \cdots, l_n)$ to define a tree is that

$$\sum_{i=1}^{n} 2^{-l_i} = 1. \tag{4}$$

The cost of the tree $(l_1, \cdots, l_n)$ is $\sum_{i=1}^{n} w_i l_i$. Trees $(l_1, \cdots, l_n)$ and $(l_1', \cdots, l_n')$ are said to be isomorphic if one sequence is a permutation of the other. Trees $T = (l_1, \cdots, l_n)$

and $T' = (l'_1, \cdots, l'_n)$ are distinct if the ordered sequences $(l_1, \cdots, l_n)$ and $(l'_1, \cdots, l'_n)$ are different. As before, we denote by $T^k = (l_1^k, \cdots, l_n^k)$ the k-best tree.

Clearly there exists an optimal binary tree $(l_1, l_2, \cdots, l_n)$ satisfying $l_1 \geqq l_2 \geqq \cdots \geqq l_n$ and without loss of generality we call it T^1. The next lemma shows that T^1 is also the best alphabetic tree with respect to the nondecreasing weight sequence $w_1, \cdots, w_n$, and thus can be computed by the Hu-Tucker Algorithm or by (1).

LEMMA 3 [Has]. *Any nonincreasing sequence* $(l_1, \cdots, l_n)$ *satisfying* (4) *represents an alphabetic tree.*

The following theorem uses this property and will be used to compute the second-best tree in every subset of the partition.

THEOREM 4. *Either* T^2 *is the second-best alphabetic tree with respect to* $w_1, \cdots, w_n$, *or* T^2 *is isomorphic to* T^1 *and* $(l_1^2, \cdots, l_n^2) = (l_1^1, \cdots, l_{r-2}^1, l_r^1, l_{r-1}^1, l_{r+1}^1, \cdots, l_n^1)$ *for some* $r \geqq 2$ *such that* $l_{r-1}^1 > l_r^1$ *and* $w_r \neq w_{r-1}$.

Proof. Clearly if T^1 and T^2 are not isomorphic then $l_1^2 \geqq l_2^2 \geqq \cdots \geqq l_n^2$. By Lemma 3, T^2 is an alphabetic tree, thus it is the second-best alphabetic tree. Suppose now that T^1 and T^2 are isomorphic. Since $l_1^1 \geqq l_2^1 \geqq \cdots \geqq l_n^1$, there must exist indices $j < r$ such that $l_j^1 > l_r^1$ and $l_r^2 = l_j^1$. Moreover, we can assume $l_{r-1}^1 > l_r^1$ and $w_{r-1} < w_r$. Then T^2 is the best tree satisfying $l_r = l_j^1$, which means that except for l_r the levels are nondecreasing: $l_1^2 \geqq l_2^2 \geqq \cdots \geqq l_{r-1}^2 \geqq l_{r+1}^2 \geqq \cdots \geqq l_n^2$. Consequently, $(l_1^2, \cdots, l_n^2) = (l_1^1, \cdots, l_{j-1}^1, l_{j+1}^1, \cdots, l_r^1, l_j^1, l_{r+1}^1, \cdots, l_n^1)$, so that T^2 is obtained from T^1 by a cyclic permutation of a subsequence $(l_j, \cdots, l_r)$. The difference in the costs of T^2 and T^1 is

$$\sum_{i=1}^{n} w_i(l_i^2 - l_i^1) = -\sum_{i=j}^{r-1} w_i(l_i^1 - l_{i+1}^1) + w_r(l_j^1 - l_r^1)$$

$$\geqq -w_{r-1} \sum_{i=j}^{r-1} (l_i^1 - l_{i+1}^1) + w_r(l_j^1 - l_r^1) = (w_r - w_{r-1})(l_j^1 - l_r^1)$$

$$\geqq (w_r - w_{r-1})(l_{r-1}^1 - l_r^1).$$

The last term is the change in costs obtained with respect to the tree $(l_1^1, \cdots, l_{r-2}^1, l_r^1, l_{r-1}^1, l_{r+1}^1, \cdots, l_n^1)$. Hence this tree is the second-best tree as claimed. □

Let B_n denote the set of nonalphabetic trees with n leaves. We now consider the problem of computing the best tree in B_n satisfying $l_i = \bar{l}_i$, $i \in Q$. Let $i_1, \cdots, i_{|Q|}$ be a permutation of $i \in Q$ such that $\bar{l}_{i_1} \leqq \bar{l}_{i_2} \leqq \cdots \leqq \bar{l}_{i_{|Q|}}$. Let $i_{|Q|+1}, \cdots, i_n$ be a permutation of $i \in \{1, \cdots, n\} \backslash Q$ such that $w_{i_{|Q|+1}} \leqq w_{i_{|Q|+2}} \leqq \cdots \leqq w_{i_n}$. Clearly there exists an optimal nonalphabetic tree $(\tilde{l}_1, \cdots, \tilde{l}_n)$ with respect to the constraints $l_i = \bar{l}_i$, $i \in Q$ satisfying $\tilde{l}_{i_{|Q|+1}} \geqq \tilde{l}_{i_{|Q|+2}} \geqq \cdots \geqq \tilde{l}_{i_n}$. Theorem 5 below shows that the sequence $\tilde{l}_{i_1}, \cdots, \tilde{l}_{i_n}$ defines an **alphabetic** tree, and obviously this implies that $(\tilde{l}_{i_1}, \cdots, \tilde{l}_{i_n})$ is the optimal alphabetic tree with respect to $w_{i_1}, \cdots, w_{i_n}$. Therefore the best **nonalphabetic** tree under the constraints $l_i = \bar{l}_i$, $i \in Q$ can be computed by applying (3) to $w_{i_1}, \cdots, w_{i_n}$.

THEOREM 5. *A sequence of integers* $(l_1, \cdots, l_n)$ *satisfying conditions* (a) *and* (b) *represents an alphabetic binary tree*:

(a) $\sum_{i=1}^{n} 2^{-l_i} = 1$

(b) *For some* m, $1 \leqq m \leqq n$, $l_1 \leqq l_2 \leqq \cdots \leqq l_{m-1} < l_m$, *and* $l_m \geqq l_{m+1} \geqq \cdots \geqq l_n$.

Proof. The proof is by induction on n. For $n = 2$ the only sequence (l_1, l_2) satisfying (a) and (b) is $(1, 1)$, which also represents an alphabetic tree. Suppose the conclusion holds for a sequence of k elements with $k \leqq n-1$ and assume the sequence $(l_1, \cdots, l_n)$

satisfies (a) and (b). Clearly $l_m = \max_{1 \leqq j \leqq n} l_j$ and as $\sum_{j=1}^{n} 2^{-l_j} = 1$ there must exist an even number of indices for which $l_j = l_m$. Without loss of generality assume $l_m = l_{m+1} = \cdots = l_{m+k_1}$ for some odd number $k_1 \geqq 1$. Let j^* and j^*+1 be the most left pair of indices for which $l_{j^*} = l_{j^*+1}$. Clearly, $j^* \leqq m$. The first step of reduction from the left of the sequence l will generate the sequence $(l_1, \cdots, l_{j^*-1}, l_{j^*} - 1, l_{j^*+2}, \cdots, l_n)$. It is easily verified that the last sequence of $n-1$ elements satisfies both (a) and (b), thus by the assumption it represents an alphabetic tree T with $n-1$ leaves. The alphabetic tree for the sequence l is obtained by adding two sons to the j^*th leaf of T. □

We now describe the partitioning scheme. Suppose $T^1 = (l_1^1, \cdots, l_n^1)$ has been computed. T^2 is then either the second-best tree in $F_1 = \{T \mid l_1 = l_1^1\}$ or the optimal tree in $F_2 = \{T \mid l_1 \neq l_1^1\}$.

To compute the optimal tree in F_2 we could imitate the partitioning procedure described in § 2 also here. However, we do not know how to solve a problem with constraints of the type $l_i \leqq D_2$ except for by solving $D_2 = O(n)$ problems with $l_i = l$, $l = q, \cdots, D_2$. The scheme requires solving $O(Kn)$ such problems, and each requires $O(n^3)$-time, so the overall complexity is $O(Kn^5)$. We now describe a modified partitioning scheme, relying on our ability to compute the two best solutions to constrained problems, that results in $O(Kn^4)$ time complexity.

The optimal tree in F_2 is obtained by solving $O(n)$ problems, each with a constraint of the form $l_1 = \hat{l}$, for $\hat{l} \in \{1, \cdots, n\} \backslash \{l_1^1\}$. Altogether the two best solutions are computed in $O(n^4)$ time complexity.

At the beginning of the kth iteration of the algorithm, we have a partition $\tilde{F}$ of $B_n \backslash \{T^1, \cdots, T^{k-1}\}$ into subsets. In each subset $F \in \tilde{F}$ we are given the best and second-best trees $T^1(F)$ and $T^2(F)$. We define $T(F)$ to be $T^1(F)$ if $T^1(F) \notin \{T^1, \cdots, T^{k-1}\}$. Otherwise we define $T(F) = T^2(F)$, and in this case $T^2(F) \notin \{T^1, \cdots, T^{k-1}\}$. The next best tree T^k, is therefore the best of all trees $\{T(F) \mid F \in \tilde{F}\}$.

Suppose $T^k = T(F^*)$. If $T^k = T^1(F^*)$ then we do not change the partition and set $T(F^*) = T^2(F^*)$. If $T^k = T^2(F^*)$ then there exist $j < k$ such that $T^j = T^1(F^*)$. Suppose $F^* = \{T \in B_n \mid l_i = l_i^j, i \in Q\}$. Since $T^k \neq T^j$ there exist $m \notin Q$ such that $l_m^k \neq l_m^j$. We replace F^* by new subsets, $F_r = \{T \in B_n \mid l_i = l_i^j, i \in Q, l_m = r\}$ for all $r = 1, \cdots, n-1$. For each of these new subsets we compute both the best and the second-best solutions. This requires $O(n^3)$-time for each subset.

We note that $\bigcup_{r=1}^{n-1} F_r = F^*$ and these sets are disjoint. For $r = l_m^j$, $T^1(F_r) = T^j$ and for $r = l_m^k$ $T^1(F_r) = T^k$. Thus for these values of r we set $T(F_r)$ to $T^2(F_r)$. For the other sets F_r we set $T(F_r)$ to $T^1(F_r)$. This requires $O(n^3)$-time for each set and $O(n^4)$ in total. Thus the overall complexity per iteration is $O(n^4)$ and for ranking K-best trees it amounts to $O(Kn^4)$.

5. Nonalphabetic trees: Algorithm D. In this section we propose an $O(Kn^3)$ algorithm for ranking the K-best nonalphabetic trees for problem (a) defined in § 1. Here a solution is defined by the sequence $(l_1, \cdots, l_n)$, i.e., interchanging the lengths of two words with identical weights **will** create a new solution of the same cost. The algorithm uses a two-stage partitioning scheme. First B_n—the set of nonalphabetic trees—is partitioned into subsets characterised by the (unordered) level set $\{l_1, \cdots, l_n\}$. Then these sets are further partitioned by an algorithm for ranking a special type of transportation problems adopted from that of Murty [M] and Weintraub [W].

Let $w_1, \cdots, w_n$ be given in ascending order $w_1 \leqq w_2 \leqq \cdots \leqq w_n$, and $T_a^1, \cdots, T_a^K$ be the K-best alphabetic trees for $w_1, \cdots, w_n$. For $k = 1, \cdots, K$, let S_{kl} be the set of

leaves of T_a^k whose level is l. Let $T_A^1, \cdots, T_A^R$ be the subsequence of $T_a^1, \cdots, T_a^K$ obtained by deleting all trees T_a^k for which there exists $j<k$ such that $|S_{jl}|=|S_{kl}|$, $l=1, \cdots, n-1$.

For $r=1, \cdots, R$ consider the following transportation problem P_r that assigns the weights $w_1, \cdots, w_n$ to the level sets S_{rl} of T_A^r:

$$
\begin{aligned}
(P_r) \qquad \text{minimize} \quad & \sum_{i,l} l w_i X_{il} \\
\text{subject to} \quad & \sum_{l} X_{il} = 1 \qquad \forall i, \\
& \sum_{i} X_{il} = |S_{rl}| \qquad \forall l, \\
& X_{il} \geqq 0 \qquad \forall i, l.
\end{aligned}
$$

Let X^{kr} be the k-best solution to (P_r) and let Y^k be the k-best solution among $\{X^{jr} | j=1, \cdots, K, r=1, \cdots, R\}$. Let T^k be the tree defined by Y^k.

THEOREM 6. *T^k is the k-best nonalphabetic tree.*

Proof. X^{rk} defines the k-best solution when the tree is restricted to have the level sets defined by S_{rl}, $l=1, \cdots, n-1$. Therefore, Y^k defines the k-best nonalphabetic tree, under the restriction that it has $|S_{rl}|$ leaves of level l, $l=1, \cdots, n-1$, for some $r \in \{1, \cdots, R\}$. We now show that this restriction is legitimate. Assume otherwise that the list $T^1, \cdots, T^k$ contains a tree that does not obey the above restriction. Let k^* be the smallest index of such a tree. Clearly in T^{k^*} the weights are assigned to leaves in a nonincreasing order, i.e., $l(w_i) \geqq l(w_j)$ for $i<j$. By Lemma 3 there exists an alphabetic tree with the same level set and where the levels of the leaves are nonincreasing. Therefore T^{k^*} is an alphabetic tree with respect to $w_1, \cdots, w_n$ and $T^{k^*} = T_A^r$ for some r, $1 \leqq r \leqq R$, in contradiction to the assumption. □

Each problem (P_r) is a transportation problem that can be reformulated as an assignment problem (P_r'):

$$
\begin{aligned}
(P_r') \qquad \text{minimize} \quad & \sum_{i} \sum_{l} \sum_{e \in S_{rl}} l w_i X_{ie} \\
\text{subject to} \quad & \sum_{e} X_{ie} = 1 \qquad \forall i \\
& \sum_{i} X_{ie} = 1 \qquad \forall e.
\end{aligned}
$$

The next best solution of (P_r) can therefore be obtained by solving $O(n)$ problems of this type, as described by Murty [M] and Weintraub [W].

The complexity of producing $T_a^1, \cdots, T_a^K$, and deleting trees to obtain $T_A^1, \cdots, T_A^R$ is of order $O(Kn^3)$. Weintraub [W] in his interesting paper presents an $O(Kn^3)$ algorithm for ranking the K-best assignments. By using his procedure to rank the solutions of (P_r') our algorithm for calculating $Y^1, \cdots, Y^K$ requires an overall complexity of $O(Kn^3)$.

It is worth pointing out that for the special case where $w_1 < w_2 < \cdots < w_n$ the two last algorithms solve the same problem. In view of the differences in their order complexities, we should prefer to use Algorithm D.

REFERENCES

[Der] U. DERIGS, *Some basic exchange properties in combinatorial optimization and their application to constructing the K-best solution*, Discrete Appl. Math., 11 (1985), pp. 129–141.

[Dre] S. F. DREYFUS, *An appraisal of some shortest path algorithms*, Oper. Res., 17 (1969), pp. 395–412.

[G] H. N. GABOW, *Two algorithms for generating weighted spanning trees in order*, SIAM J. Comput., 6 (1977), pp. 139–151.

[GM] E. N. GILBERT AND E. F. MOORE, *Variable length binary encodings*, Bell Syst. Tech. J., 38 (1959), pp. 933–968.

[HT] T. C. HU AND A. C. TUCKER, *Optimal computer search trees and variable-length alphabetic codes*, SIAM J. Appl. Math., 21 (1971), pp. 514–532.

[Has] R. HASSIN, *A dichotomous search for a geometric random variable*, Oper. Res., 32 (1984), pp. 423–439.

[Huf] P. A. HUFFMAN, *A method for the construction of minimum redundancy codes*, Proc. I.R.E., 40 (1952), pp. 1098–1101.

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

[KIM1] N. KATOH, T. IBARAKI, AND H. MINE, *An algorithm for finding K minimum spanning trees*, SIAM J. Comput., 10 (1981), pp. 247–255.

[KIM2] ———, *An algorithm for the K best solution of the resource allocation problem*, J. Assoc. Comput. Mach., 28 (1981), pp. 752–764.

[KIM3] ———, *An efficient algorithm for K shortest simple paths*, Networks, 12 (1982), pp. 411–427.

[L1] E. L. LAWLER, *A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem*, Management Sci., 18 (1972), pp. 401–405.

[L2] ———, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[M] K. G. MURTY, *An algorithm for ranking all the assignments in order of increasing cost*, Oper. Res., 16 (1968), pp. 682–687.

[RH] F. RUSKEY AND T. C. HU, *Generating binary trees lexicographically*, SIAM J. Comput., 6 (1977), pp. 745–758.

[W] A. WEINTRAUB, *The shortest and the K-shortest routes as assignments problems*, Networks, 3 (1973), pp. 61–73.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 893-905, October 1989

003

AN ANALYSIS OF THE RATIONAL EXPONENTIAL INTEGRAL*

G. W. CHERRY†

Abstract. In this paper an algorithm is presented for integrating expressions of the form $\int ge^f dx$, where f and g are rational functions of x, in terms of a class of special functions called the special incomplete Γ functions. This class of special functions includes the exponential integral, the error function, the sine and cosine integrals, and the Fresnel integrals. The algorithm presented here is an improvement over those published previously for integrating with special functions in the following ways: (i) This algorithm combines all the above special functions into one algorithm, whereas previously they were treated separately. (ii) Previous algorithms require that the underlying field of constants be algebraically closed. This algorithm, however, works over any field of characteristic zero in which the basic field operations can be carried out. (iii) This algorithm does not rely on Risch's solution of the differential equation $y'+fy=g$. Instead, a more direct method of undetermined coefficients is used.

Key words. Liouville's theorem, integration in finite terms, special functions, logarithmic integrals, exponential integrals, error functions

AMS(MOS) subject classifications. 68Q40, 12H05

1. Background. It is well known that if C is the field of complex numbers and g is a rational function in $C(x)$, then $\int g\,dx$ can be written in terms of elementary functions. Specifically, the integral can be written as the sum of a rational function and a linear combination of logarithms. There are, however, algorithmic difficulties encountered if we try to factor the denominator of g (or any part of this denominator) into linear factors to determine the logarithms. Only recently (cf. [Roth76], [Trag76]) have the algorithmic details been worked out.

In this paper we present a similar analysis of the *rational exponential integrals*—integrals of the form $\int ge^f dx$ where f and g are rational functions of x. Our algorithm will determine whether a given rational exponential integral can be written in terms of a class of functions called the *special incomplete* Γ *functions* [Bate53]. This class includes, in addition to the elementary functions, a number of well-known special functions such as the exponential integral

$$\operatorname{ei}(u) = \int \frac{u'}{u} e^u\,dx$$

and the error function[1]

$$\operatorname{erf}(u) = \int u'e^{u^2}\,dx.$$

Examples of integrations performed by our algorithm are

$$\int e^{1/x}\,dx = xe^{1/x} - \operatorname{ei}\left(\frac{1}{x}\right)$$

and

$$\int \left(\frac{1}{x}+\frac{1}{x^2}\right) e^{1/x^2}\,dx = -\frac{1}{2}\operatorname{ei}\left(\frac{1}{x^2}\right) - \operatorname{erf}\left(\frac{1}{x}\right).$$

* Received by the editors June 22, 1987; accepted for publication (in revised form) December 14, 1988.
† Tektronix Labs, Computer Research Laboratories, Beaverton, Oregon, 97077.

[1] The usual error function, $\operatorname{Erf}(x) = \int_0^x e^{-t^2}\,dt$ [Bate53], differs from our definition, which is denoted as Erfi in [Bate53], as follows: $\operatorname{Erf}(x) = 1/i \operatorname{Erfi}(ix)$. Also see the Appendix.

As these examples indicate, there are many integrals that cannot be written in terms of elementary functions but can be written in terms of special incomplete Γ functions. Another, more general example of this is the integral $\int ge^x\,dx$. It is elementary only if g is a polynomial, but it can always be integrated in terms of elementary functions and exponential integrals.

Algorithms for integrating in terms of special functions have been reported in [Ssc85], [Cher83], [Cher85], and [Cher86] where separate procedures have been given for integrating transcendental elementary functions in terms of exponential integrals and error functions. This work was later extended in [Know86a] and [Know86b]. The algorithm presented in this paper can be viewed as a synthesis and refinement of the so-called "base cases" of these procedures. In addition to the synthesis we have made two improvements to the original algorithms. The first concerns algebraic constants. If we are interested in expressing rational exponential integrals only in terms of elementary functions, then it is not necessary to introduce new algebraic constants (e.g., if the integrand is composed of rational functions f and g that lie in $Q(x)$, then an elementary integral, if it exists, will be of the form he^f where h is in $Q(x)$). However, when integrating in terms of special incomplete Γ functions new algebraic constants may arise. For example,

$$\int \frac{e^x}{x^2-2}\,dx = e^{\sqrt{2}}\,\mathrm{ei}\,(x+\sqrt{2}) + e^{\sqrt{2}}\,\mathrm{ei}\,(x+\sqrt{2}).$$

In [Cher83], and again in [Know86a], [Know86b], it is assumed that the field of constants over which the integrand is defined is algebraically closed. Under this assumption it is not necessary to introduce "new" algebraic constants to express the integral. It is necessary, however, to perform all operations such as partial fraction decompositions over an algebraically closed field such as the algebraic closure of the rationals. Although theoretically sound (cf. [DaTr81]) there are difficulties with this approach. The algorithm presented here works over any constant field of characteristic zero, provided the basic field operations can be performed. Also, any new algebraic constants that are necessary for expressing the integral will be expressed in a field of lowest possible degree. In fact, it will be shown that any algebraic constants occurring in the error functions are quadratic over the constant field.

The second improvement presented here concerns the overall structure of the algorithm. In previous algorithms a two step strategy has been employed: first we generate a finite set of special functions

$$\left\{\int h_1 e^f, \cdots, \int h_n e^f\right\}$$

such that if the integral can be resolved, then the special functions that appear in the result are from this set. For the class of special functions we are considering here, this implies that there exist constants c_i and a rational function y such that

$$\int ge^f + c_1 \int h_1 e^f + \cdots + c_n \int h_n e^f = ye^f.$$

The second step is to apply an algorithm of Risch's [Risch69, Main Theorem, part (b)] that reduces the calculation of the elements $y, c_1, \cdots, c_n$ to the solution of a linear system of equations. We observe, however, that the actual construction of this linear system is often not necessary. In fact, it has recently been shown [Dav86] that in the base case of Risch's algorithm (the only part needed in this paper) it is never necessary to explicitly solve a linear system of equations. Instead, by examining the partial

fraction decompositions of f and g, we can directly solve for y. The algorithm presented here uses similar techniques to those in [Dav86] to directly solve for the elementary part y and then for the exponential integrals and error functions.

The paper is organized as follows. In §2 we give a formal definition of the special incomplete Γ functions and prove two results concerning the structure of integrals that can be expressed in terms of these functions. In §3 we examine the problem of completing squares of rational functions, which is a basic problem encountered when integrating with error functions. The algorithm is described in §4 and examples are given in §5.

2. The special incomplete Γ functions. Let F be a differential field[2] of characteristic zero with derivation $'$. A differential extension E of F is called a *special incomplete* Γ *extension of* F if there exists a tower of fields $F = F_0 \subset F_1 \subset \cdots \subset F_n = E$ such that for each i, $1 \leqq i \leqq n$, $F_i = F_{i-1}(\theta_i)$ and one of the following holds:

(i) θ_i is algebraic over F_{i-1}.

(ii) θ_i is logarithmic over F_{i-1} (i.e., $\theta_i' = u'/u$ for some nonzero u in F_{i-1}, and we write $\theta_i = \log(u)$).

(iii) θ_i is an exponential over F_{i-1} (i.e., $\theta_i' = u'\theta_i$ for some u in F_{i-1}, and we write $\theta_i = e^u$).

(iv) θ_i is an error function over F_{i-1} (i.e., $\theta_i' = u'v$ for some u and v in F_{i-1} such that $v' = (u^2)'v$, and we write $\theta_i = \operatorname{erf}(u)$).

(v) θ_i is an exponential integral over F_{i-1} (i.e., $\theta_i' = (u'/u)v$ for some nonzero u and v in F_{i-1} such that $v' = u'v$, and we write $\theta_i = \operatorname{ei}(u)$).

We note that (v) could be replaced by the equivalent statement (v′):

(v′) θ_i is a logarithmic integral over F_{i-1} (i.e., $\theta_i' = u'/v$ for some nonzero u and v in F_{i-1} such that $v' = u'/u$, and we write $\theta_i = \operatorname{li}(u)$).

The following theorem is a synthesis of two results that have appeared in [Ssc85] and [Cher83]. The proof relies on the following lemma (that also can be found in [Cher83]).

LEMMA 2.1. *Let k be a field containing the nth roots of unity and let K be an algebraic extension of k. If v is an element of K such that v^n is in k, then either v is in k or the trace of v in K with respect to k is zero.*

Proof. First note that v satisfies a pure equation and therefore has a cyclic Galois group: $\{\sigma, \sigma^2, \cdots, \sigma^r\}$ [Vdw50]. Next, let $\sigma(v) = \xi^k v$ where ξ is a primitive nth root of unity and write the conjugates of v as

$$\begin{aligned} \sigma(v) &= \xi^k v \\ \sigma^2(v) &= \xi^{2k} v \\ &\vdots \\ \sigma^r(v) &= \xi^{rk} v. \end{aligned}$$

Now since $\sigma = \sigma^{r+1}$, ξ^k is an rth root of unity and so either $r = 1$ or

$$\operatorname{Tr}(v) = (\xi^k + \cdots + \xi^{rk})v = 0 \cdot v = 0. \qquad \square$$

THEOREM 2.2. *Let E be a Liouvillian extension of $C(x)$ where C is a field of constants, x is transcendental over C, and $x' = 1$. Assume C is algebraically closed and has characteristic zero, and let γ be an element of E that has an integral in some special*

[2] We shall assume the reader is familiar with the basic terminology of differential algebra (cf. [Kap57]).

incomplete Γ*-extension of* E. *Then there exist constants* b_i, c_i, *and* d_i *in* C, *elements* w_i, u_i, *and* v_i *in* E *and* $\tilde{u}_i$ *and* $\tilde{v}_i$ *algebraic over* E *such that*

$$\gamma = w_0' + \sum b_i \frac{w_i'}{w_i} + \sum c_i \frac{u_i'}{u_i} v_i + \sum d_i \tilde{u}_i' \tilde{v}_i \tag{2.1}$$

where $v_i' = u_i' v_i$, $\tilde{v}_i' = (\tilde{u}_i^2)' \tilde{v}_i$, *and* $\tilde{u}_i^2$, $\tilde{v}_i^2$, *and* $\tilde{u}_i' \tilde{v}_i$ *are in* E.

Proof. A direct application of the main theorem from [Ssc85][3] implies that there exist constants b_i, c_i, and d_i and elements w_i, u_i, v_i, $\tilde{u}_i$, and $\tilde{v}_i$ algebraic over E satisfying (2.1). Now for each i, since $v_i' = u_i' v_i$, we have by the lemma on p. 338 of [RoSi77] that u_i is in E and some positive power of v_i is in E. Therefore some positive power of $(u_i'/u_i)v_i$ is in E. Similarly, for each i, we have $\tilde{u}_i^2$ and some positive power of $\tilde{v}_i$ in E. Moreover, $(\tilde{u}_i')^2 = \frac{1}{4}((\tilde{u}_i^2)')^2/\tilde{u}_i^2$, and so $(\tilde{u}_i')^2$ is in E and some positive power of $\tilde{u}_i' \tilde{v}_i$ is in E. Next let $\bar{E}$ be a normal extension of E containing the w_i, u_i, v_i, $\tilde{u}_i$, and $\tilde{v}_i$ and take the trace in $\bar{E}$ on both sides of (2.1) over the field E. This yields

$$m\gamma = (\operatorname{Tr}(w_0))' + \sum c_i \frac{(N(w_i))'}{N(w_i)} + \sum c_i \operatorname{Tr}\left(\frac{u_i'}{u_i} v_i\right) + \sum d_i \operatorname{Tr}(\tilde{u}_i' \tilde{v}_i)$$

where m is a positive integer and $N(w_i)$ is the norm of w_i. Now Lemma 2.1 implies that, for each i, $\operatorname{Tr}((u_i'/u_i)v_i) = 0$ unless $(u_i'/u_i)v_i$ is in E and $\operatorname{Tr}(\tilde{u}_i' \tilde{v}_i) = 0$ unless $\tilde{u}_i' \tilde{v}_i$ is in E. In the first case u_i is in E and so v_i is in E. In the second case $(\tilde{u}_i')^2$ is in E and so $\tilde{v}_i^2 = (\tilde{u}_i' \tilde{v}_i)^2/(\tilde{u}_i')^2$ is in E. This completes the proof. □

We now examine the case where F is a field of constants, f and g are in $F(x)$, and E is the differential field $\bar{F}(x, e^f)$.

THEOREM 2.3. *Let* F *be a field of constants and let* x *be transcendental over* x *with* $x' = 1$. *Let* f *and* g *be elements of* $F(x)$ *and suppose that* ge^f *has an integral in some special incomplete* Γ*-extension of* $F(x, e^f)$. *Then there exist constants* c_i *and* d_i *in* $\bar{F}$, *an element* y *in* $F(x)$, *and elements* u_i *and* $\tilde{u}_i$ *in* $\bar{F}(x)$ *such that*

$$g = y' + f'y + \sum c_i \frac{u_i'}{u_i} + \sum d_i \tilde{u}_i' \tag{2.2}$$

where

(i) *For each* u_i *there exists an* α_i *algebraic over* F *such that* $u_i = f + \alpha_i$; *and*

(ii) *For each* $\tilde{u}_i$ *there exists a* β_i *algebraic over* F *such that* $\tilde{u}_i^2 = f + \beta_i$.

Proof. Let θ denote the exponential e^f and apply the above theorem (where E is the field $\bar{F}(x, \theta)$). This yields constants b_i, c_i, and d_i algebraic over F, w_i, u_i, and v_i in $\bar{F}(x, \theta)$, and $\tilde{u}_i$ and $\tilde{v}_i$ algebraic over $\bar{F}(x, \theta)$ such that

$$g\theta = w_0' + \sum b_i \frac{w_i'}{w_i} + \sum c_i \frac{u_i'}{u_i} v_i + \sum d_i \tilde{u}_i' \tilde{v}_i \tag{2.3}$$

where $v_i' = u_i' v_i$, $\tilde{v}_i' = (\tilde{u}_i^2)' \tilde{v}_i$ and $\tilde{u}_i^2$, $\tilde{v}_i^2$ and $\tilde{u}_i' \tilde{v}_i$ are in $\bar{F}(x, \theta)$. Since, for each i, v_i is an exponential of u_i, Theorem 3.1 of [RoCa79] implies that there exist rational numbers δ_i and constants α_i in $\bar{F}$ so that $u_i = \delta_i f - \alpha_i$. Thus θ^{δ_i} and v_i are both exponentials of u_i and, since two exponentials of a fixed element of a differential field must differ by a multiplicative constant, we have $v_i = \theta^{\delta_i} \eta_i$ where η_i is in $\bar{F}$. Note that since v_i is in $\bar{F}(x, \theta)$, δ_i is an integer. Similarly, there exist rational numbers $\tilde{\delta}_i$ and constants β_i and $\tilde{\eta}_i$ in $\bar{F}$ so that $\tilde{u}_i^2 = \tilde{\delta}_i f + \beta_i$ and $\tilde{v}_i = \theta^{\tilde{\delta}_i} \tilde{\eta}_i$. Here $\tilde{\delta}_i$ must be an integer since $\tilde{u}_i' \tilde{v}_i$ is in

[3] Where the alternative definition (v') is used for the special incomplete Γ-extensions.

$\bar{F}(x, \theta)$ and $\tilde{u}_i'$ does not involve θ. This now implies that $\tilde{u}_i'$ is in $\bar{F}(x)$ and hence, $\tilde{u}_i = (\tilde{u}_i^2)'/2\tilde{u}_i'$ is in $\bar{F}(x)$. Now, absorbing η_i into c_i and $\tilde{\eta}_i$ into d_i, write (2.3) as

$$g\theta = w_0' + \sum b_i \frac{w_i'}{w_i} + \sum c_i \frac{u_i'}{u_i}\theta^{\delta_i} + \sum d_i \tilde{u}_i' \theta^{\tilde{\delta}_i} \tag{2.4}$$

and compare the partial fraction decompositions in the variable θ over the field $\bar{F}(x)$ on both sides of (2.4). This yields $\delta_i = \tilde{\delta}_i = 1$, $w_i = 0$ for $i \geqq 1$, and $w_0 = y\theta$ for some y in $\bar{F}(x)$. Equation (2.4) becomes

$$g\theta = (y\theta)' + \sum c_i \frac{u_i'}{u_i}\theta + \sum d_i \tilde{u}_i' \theta$$

and dividing by θ yields

$$g = y' + f'y + \sum c_i \frac{u_i'}{u_i} + \sum d_i \tilde{u}_i'. \tag{2.5}$$

Now let E be a normal extension of $F(x)$ containing y and the constants α_i, c_i, β_i, and d_i, and let σ be an automorphism of E that fixes the field $F(x)$. Then from (2.5) we have

$$g = (\sigma y)' + f'(\sigma y) + \sum \sigma(c_i) \frac{\sigma(u_i')}{\sigma(u_i)} + \sum \sigma(d_i)\sigma(\tilde{u}_i)'$$

where $\sigma(u_i) = f + \sigma(\alpha_i)$ and $\sigma(\tilde{u}_i)^2 = f + \sigma(\beta_i)$. Summing over all such automorphisms, we have

$$mg = \left(\sum_\sigma y\right)' + f'\left(\sum_\sigma y\right) + \sum_\sigma \sum \sigma(c_i) \frac{\sigma(u_i')}{\sigma(u_i)} + \sum_\sigma \sum \sigma(d_i)\sigma(\tilde{u}_i)'$$

where m is a positive integer. Since $\sum_\sigma y$ is in $F(x)$, dividing this equation by m yields an equation in the same form as (2.2) and the proof is complete. □

The theorem implies that

$$\int g e^f \, dx = y e^f + \sum c_i e^{-\alpha_i} \operatorname{ei}(u_i) + \sum d_i e^{-\beta_i} \operatorname{erf}(\tilde{u}_i).$$

3. Completing squares of rational functions. In Theorem 2.3 we have shown that if error functions appear in the integral, then there exist algebraic constants β_i such that

$$f + \beta_i = \tilde{u}_i^2. \tag{3.1}$$

In this section we show that there can be at most two such values for β_i and that these values are either in F or are quadratic conjugates over F.

In the case where f is in $F[x]$ we have the familiar problem of completing squares of polynomials. Here the basic result is well known. If f is a nonconstant polynomial, then there can be at most one constant β such that $f + \beta$ is a perfect square. We state and prove here an equivalent statement that shall be useful in the next section.

THEOREM 3.1. *Let F be a constant field of characteristic zero and let f and g be two nonconstant polynomials in $F[x]$ such that $f^2 - g^2 = c$ where c is a constant. Then $c = 0$.*

Proof. Assume there are nonconstant polynomials f and g and a nonzero constant c such that $f^2 - g^2 = c$. Then f and g must be of the same degree and relatively prime. Now differentiate both sides of the equation $f^2 - g^2 = c$. This yields the equality $f'f = g'g$. Since f and g have no common factors, this implies $f = \delta g'$ where δ is in F. This, however, is impossible since $\partial f = \partial g$ (where ∂f denotes the degree of f). □

The more general problem of completing squares of rational functions and its connection to error functions was first treated in [Ssc85]. Since then the problem has been pursued in [CoTr82], [Cher83], and [Zwil84]. The results below were originally stated in the unpublished manuscript [CoTr82], although the proofs presented here are new.

THEOREM 3.2. *Let F be a constant field of characteristic zero and let f be a nonconstant element of $F(x)$. Then there are at most two constants β_i in $\bar{F}$ such that $f+\beta_i = g_i^2$ for some g_i in $\bar{F}[x]$.*

Proof. Let $f = p/q$ where p and q are relatively prime and q is monic, and for each i let $g_i = r_i/s_i$ where r_i and s_i are relatively prime and s_i is monic. Now note that for all β_i, $p+\beta_i q$ and q are relatively prime. Therefore $q = s_i^2$ and $p+\beta_i q = r_i^2$. This determines s_i uniquely and therefore we drop the subscript on s_i. We first claim that any two solutions r_i and r_j, corresponding to distinct β_i and β_j are relatively prime. To show this let β_i and β_j be distinct elements of $\bar{F}$ and r_i and r_j be elements of $\bar{F}[x]$ so that $p+\beta_i q = r_i^2$ and $p+\beta_j q = r_j^2$. Subtraction yields the equality $(\beta_i-\beta_j)q = r_i^2 - r_j^2$. This implies that if a common factor of r_i and r_j exists, then it must divide q. But from $p+\beta_i q = r_i^2$, this common factor of q and r_i also divides p. This contradicts the assumption that p and q are relatively prime and proves our claim. Now let $\max\{\partial p, \partial q\} = m$ for some integer m. Clearly, if $\partial q \geqq \partial p$, then $m = 2\partial s$ is even. On the other hand, if $\partial p > \partial q$ then $m = \partial p$ must be even if there is to be a solution r to $p+\beta q = r^2$. Therefore let $m = 2n$. Now consider the possibility that $\partial r_i < n$ for some i. This can only happen if (i) $\partial p = \partial q$ and $\beta_i = -lc(p)$ (where $lc(p)$ denotes the leading coefficient of p) or (ii) $\partial p < \partial q$ and $\beta_i = 0$. Since these situations are mutually exclusive we know that there can be at most one r_i such that $\partial r_i < n$. For the remainder of the proof we will label such an r_i as r_1. Thus, $\partial r_1 \leqq n$ and $\partial r_j = n$ for $j > 1$.

Now differentiating both sides of the equation $p/q + \beta_i = (r_i/s)^2$, we obtain $sp' - 2ps' = 2r_i(sr_i' - r_i s')$. Thus for distinct i and j we have $r_i(sr_i' - r_i s') = r_j(sr_j' - r_j s')$, and, in particular, $r_1(sr_1' - r_1 s') = r_j(sr_j' - r_j s')$ for $j > 1$. This implies, since r_1 and r_j are relatively prime, that $r_j | (sr_1' - r_1 s')$ for all $j > 1$. But $\partial(sr_1' - r_1 s') \leqq 2n-1$, implying that there can be at most one solution r_j where $j > 1$ and a maximum of two solutions. □

COROLLARY 3.3. *Let F, f, g, β_1, and β_2 be as in Theorem 3.2. Then, if solutions β_1 and β_2 are not in F, they are quadratic conjugates over F.*

Proof. This follows directly from the observation that if β is a solution to (3.1), i.e., $f+\beta = g^2$ for some g in $\bar{F}[x]$, and if $\sigma(\beta)$ is an automorphic image over F of β, then $\sigma(\beta)$ is also a solution to (3.1). □

Solutions to (3.1) can be computed by calculating the resultant, resultant $(p+\beta q, p'+\beta q')$, with respect to x. This will be a polynomial, say $h(\beta)$, in the variable β of degree $3n-1$ where n is as above. Observe that $\hat{\beta}$ is a root of h if and only if $p+\hat{\beta}q$ has multiple factors. We can therefore find all the linear and quadratic roots of $h(\beta)$ and test for perfect squares. However, in the next section we will see that it is not necessary to perform this computation to integrate with error functions. Instead it will be possible to find constants U and V such that the solutions to (3.1) are roots of the quadratic polynomial $\beta^2 - U\beta + V$.

4. The algorithm. In § 2 it has been shown that computing the indefinite integral $\int ge^f$ is equivalent to finding suitable y, c_i, u_i, d_i, and $\tilde{u}_i$ such that

$$g = y' + f'y + \sum c_i \frac{u_i'}{u_i} + \sum d_i \tilde{u}_i'. \tag{4.1}$$

The algorithm presented in this section solves first for the elementary part y, then for the exponential integrals, and finally for the error functions.

First we introduce some notation. We will assume that $f = p/q$ where p and q are relatively prime polynomials and q is monic. Then for each i, since $f + \alpha_i = u_i$, we can write $u_i = p_i/q$ where $p_i = p + \alpha_i q$ is relatively prime to q. Also for each $i, f + \beta_i = \tilde{u}_i^2$ implies that there exist polynomials r_i in $\bar{F}[x]$ such that $\tilde{u}_i = r_i/s$, where s is the unique monic square root of q, $r_i^2 = p + \beta_i q$, and r_i and s are relatively prime. We note here that no error functions can appear in the integral if q is not a perfect square.

We will also make use of the following definitions. For an arbitrary element f in $F(x)$ we define the *degree* of f, denoted ∂f, to be $\partial(\text{num}(f)) - \partial(\text{den}(f))$. Now write $f = f_p + f_n/f_d$ where f_p, f_n, and f_d are in $F[x]$, $\partial f_n < \partial f_d, f_n$, and f_d are relatively prime, and f_d is monic. We define the *leading coefficient* of f, denoted $lc(f)$, to be the leading coefficient of f_p. Finally we borrow an idea from [Dav86]. For f not in $F[x]$ write f_n in the form $c_{r-1}x^{r-1} + \cdots + c_0$ where r is the degree of f_d. We then refer to the constant c_{r-1} as the *proper fraction coefficient* of f and denote it f_∞. We also define f_∞ to be zero for all f in $F[x]$. Note that $f_\infty = 0$ if $\partial f_n < r - 1$. It is easy to show that (i) $(f')_\infty = 0$ for all f in $F(x)$, and (ii) $(f+g)_\infty = f_\infty + g_\infty$ for all f and g in $F(x)$.

4.1. The elementary part. To compute y we can use the same strategy that is used in [Dav86, Thm. 1]. We repeat the steps here to show that the partial fraction arguments presented in [Dav86] are not hampered by the special function terms in (4.1). First write $g = \bar{G}/g_1 g_2$ where $\bar{G}$, g_1, and g_2 are in $F[x]$, g_1 and g_2 are monic, g_2 is the product of all the irreducible factors of the denominator of g that divide q, and g_1 is the product of all the irreducible factors of the denominator of g that do not. Next perform a partial fraction decomposition of g

$$g = G + \frac{\tilde{g}_1}{g_1} + \frac{\tilde{g}_2}{g_2} \tag{4.2}$$

where $G, \tilde{g}_1$, and $\tilde{g}_2$ are in $F[x], \partial(\tilde{g}_1) < \partial(g_1)$, and $\partial(\tilde{g}_2) < \partial(g_2)$. We now divide the computation of y into three steps. Steps E1 and E2 will solve for the "fractional part" of y, and Step E3 will solve for the "polynomial part" of y.

Step E1. Perform a squarefree decomposition of g_1, $g_1 = h_r^r \cdots h_1$, and a partial fraction decomposition

$$\frac{\tilde{g}_1}{g_1} = \sum_{i=1}^{r} \sum_{j=1}^{i} \frac{G_{ij}}{h_i^j} \tag{4.3}$$

where $\partial(G_{ij}) < \partial(h_i)$. Combining (4.2) and (4.3), we have the following decomposition for g:

$$g = G + \sum_{i=1}^{r} \sum_{j=1}^{i} \frac{G_{ij}}{h_i^j} + \frac{\tilde{g}_2}{g_2}. \tag{4.4}$$

Now suppose that y is written in the same format

$$y = Y + \sum_{i=1}^{n} \sum_{j=1}^{i} \frac{Y_{ij}}{z_i^j} + \frac{\tilde{y}_2}{y_2} \tag{4.5}$$

and substitute (4.4) and (4.5) into (4.1). This yields

$$G + \sum_{i=1}^{r} \sum_{j=1}^{i} \frac{G_{ij}}{h_i^j} + \frac{\tilde{g}_2}{g_2} = \left(Y + \sum_{i=1}^{n} \sum_{j=1}^{i} \frac{Y_{ij}}{z_i^j} + \frac{\tilde{y}_2}{y_2} \right)' + \left(\frac{p}{q} \right)' \left(Y + \sum_{i=1}^{n} \sum_{j=1}^{i} \frac{Y_{ij}}{z_i^j} + \frac{\tilde{y}_2}{y_2} \right) + \sum c_i \left(\frac{p_i'}{p_i} - \frac{q'}{q} \right) + \sum d_i \left(\frac{r_i}{s} \right)'. \tag{4.6}$$

Performing the differentiations and comparing terms on both sides of this equation, we see that $r=n+1$, $z_n=h_r$, and $G_{rr}\equiv -nY_{nn}h_r'(\text{modulo } h_r)$. Since h_r and h_r' are relatively prime, this congruence can be solved uniquely for Y_{nn}. With Y_{nn} known we can make the substitution $y=\bar{y}-Y_{nn}/h_r^r$ in (4.1). This will result in an equation in the same form of (4.1) (with $\bar{y}$ replacing y) where the multiplicity of h_r in g_1 is reduced. After a sequence of such substitutions g_1 becomes squarefree, implying that each irreducible factor of the denominator of y divides q. Note that we have reduced n to zero—the variable n will be reused below in Step E2.

Step E2. We now make the following observations concerning any irreducible factors of y_2. Let ϕ be such an irreducible factor and suppose $\phi^n\|\text{den}(y)$ and $\phi^m\|q$.[4] By assumption, $n, m>0$. Then $\phi^{n+1}\|\text{den}(y')$, $\phi^{m+1}\|\text{den}(f')$ and $\phi^{n+m+1}\|\text{den}(f'y)$. For the special function terms we have, for each i, $\phi\|\text{den}(u_i'/u_i)$, and if $s\neq 1$ then $\phi^{(m/2)+1}\|\text{den}(\tilde{u}_i')$. Now examine the partial fraction decompositions on both sides of (4.1). Since $m>0$, the $f'y$ term dominates with respect to ϕ and $\phi^{n+m+1}\|g_2$. Our strategy, therefore, is to apply substitutions of the form $y=\bar{y}-\delta$ that reduce the multiplicities in g_2 until the above degree argument implies that $y_2=1$. This will happen when the highest multiplicity in g_2 is $m+1$ or less.

Following the strategy outlined in [Dav86, Lemma 4.3], we first find a squarefree polynomial h such that $h^r\|g_2$, $h^m\|q$, and such that g_2/h^r and q/h^m are relatively prime to h. h can be generated easily from the squarefree factors of g_2 and q. If $r\leqq m+1$ then the above observations imply that h is relatively prime to y_2. Therefore assume that $r>m+1$. Then the above observations applied to the irreducible factors of h imply that $h^{r-(m+1)}\|y_2$. Now write the partial fraction decompositions of $\tilde{g}_2/g_2$, $\tilde{y}_2/y_2$ and f' as

$$\frac{\tilde{g}_2}{g_2}=\frac{\hat{g}}{h^r}+\cdots,$$

$$\frac{\tilde{y}_2}{y_2}=\frac{\hat{y}}{h^{r-(m+1)}}+\cdots,$$

$$f'=\frac{\hat{f}}{h^{m+1}}+\cdots$$

where $\hat{g}$, $\hat{y}$, and $\hat{f}$ are relatively prime to h with degrees less than h. We then have $\hat{g}\equiv\hat{f}\hat{y}(\text{modulo } h)$. This congruence can be uniquely solved for $\hat{y}$ and the substitution $y=\bar{y}-\hat{y}/h^{r-(m+1)}$ will reduce the multiplicity of h in g_2. After a finite number of such substitutions we can assume that y is a polynomial.

Step E3. When y is written as a polynomial, $y=y_nx^n+\cdots+y_0$, (4.1) becomes

$$g=(y_nx^n+\cdots+y_0)'+f'(y_nx^n+\cdots y_0)+\sum c_i\frac{u_i'}{u_i}+\sum d_i\tilde{u}_i'. \tag{4.7}$$

Now we consider two cases.

Case (i). $\partial p>\partial q$. Here $lc(f')=0$ and we compare leading coefficients on both sides of (4.7). First note that, for each i, $\partial(u_i'/u_i)=-1$ and, since $\tilde{u}_i^2=f+\beta_i$, $\partial(\tilde{u}_i')=\frac{1}{2}\partial f-1=\frac{1}{2}(\partial f'-1)$. Therefore

$$\partial(y'+f'y)=\partial(f'y)>\partial\left(\sum c_i\frac{u_i'}{u_i}+\sum d_i\tilde{u}_i'\right),$$

[4] The notation $u^r\|v$, for polynomials u and v and a nonnegative integer r, means that u^r divides v but u^{r+1} does not.

which implies $lc(g) = lc(f')y_n$. Solving for y_n and making the substitution $y = \bar{y} - y_n x^n$ will reduce the degree of g. Repeating this procedure will eventually yield all of y.

Case (ii). $\partial p \leqq \partial q$. Here $\partial f' \leqq -2$ and, for all i, $\partial(u_i'/u_i) = -1$ and $\partial(\tilde{u}_i') \leqq -2$. In the subcase where $n > 0$, we have

$$\partial(y' + f'y) = \partial(y') > \partial\left(\sum c_i \frac{u_i'}{u_i} + \sum d_i \tilde{u}_i'\right),$$

which implies $lc(g) = ny_n$ and we can solve for y_n and reduce the degree of g. Next consider the subcase where $n = 0$. Choose any squarefree factor of q, say h, such that $h^m \| q$ and q/h^m is relatively prime to h. Again we have $h^{m+1} \| \text{den}(f')$, and for each i, $h \| \text{den}(u_i'/u_i)$ and $h^{m/2+1} \| \text{den}(\tilde{u}_i')$. Next we form the partial fraction expansions of g and f'

$$g = \frac{\hat{g}}{h^r} + \cdots, \qquad f' = \frac{\hat{f}}{h^{m+1}} + \cdots,$$

and compare partial fraction expansions on both sides of (4.1). This yields $r = m+1$ and $y_0 = \hat{g}/\hat{f}$.

4.2. The exponential integrals. When we assume that y has been calculated, (4.1) becomes

$$g = \sum c_i \left(\frac{p_i'}{p_i} - \frac{q'}{q}\right) + \sum d_i \left(\frac{r_i}{s}\right)'.$$

Comparing this equation with the decomposition of g in (4.2), we obtain

$$\frac{\tilde{g}_1}{g_1} = \sum_{p_i \notin \bar{F}} c_i \frac{p_i'}{p_i} \tag{4.8}$$

and

$$G + \frac{\tilde{g}_2}{g_2} = -\frac{q'}{q} \sum_{p_i \notin \bar{F}} c_i - \sum_{p_i \in \bar{F}} c_i \frac{q'}{q} + \sum d_i \left(\frac{r_i}{s}\right)'. \tag{4.9}$$

We will consider each of these two equations separately.

Step P1. To resolve (4.8) compute the resultant of g_1 and $p + \alpha q$ with respect to x. This will be a polynomial in the variable α, say $h(\alpha)$. The roots of $h(\alpha)$ will each yield a factor $p_i = p + \alpha_i q$ of g_1. We now have a factorization of g_1, $g_1 = \Pi p_i$, and can perform a partial fraction decomposition of $\tilde{g}_1/g_1$:

$$\frac{\tilde{g}_1}{g_1} = \sum \frac{\tilde{p}_i}{p_i}. \tag{4.10}$$

Comparison of (4.8) and (4.10) determines whether constants c_i exist satisfying equation (4.8). If no such constants exist, then the integral does not exist in a special incomplete Γ-extension and the algorithm terminates.

Step P2. Now consider (4.9) and note that there are only two mutually exclusive cases where $p_i = p + \alpha_i q$ is in $\bar{F}$ for some i. These are (i) $\partial p = \partial q$ and $\alpha_i = lc(p)$, and (ii) p is in F and $\alpha_i = 0$. In either case there can only be one term in the sum $\sum_{p_i \in \bar{F}} c_i(q'/q)$. We therefore write this sum as $c(q'/q)$ and, transposing the known quantity $-q'/q \sum_{p_i \notin \bar{F}} c_i$ to the left-hand side, we rewrite (4.9) as

$$\gamma = -c\frac{q'}{q} + \sum d_i \left(\frac{r_i}{s}\right)'$$

where γ is a known element in $F(x)$. Now if $q \neq 1$ we have, by comparing proper fraction coefficients on both sides of this equation, that $c = -\gamma_\infty/\partial q$. This concludes the calculation of the exponential integrals.

4.3. The error functions. We have reduced the original problem to the calculation of constants d_i and polynomials r_i such that

$$g = \sum d_i \left(\frac{r_i}{s}\right)' \tag{4.11}$$

where, for each i, there exists a constant β_i so that $p/q + \beta_i = (r_i/s)^2$. It is easy to determine whether q is a perfect square and, if so, to compute s such that $q = s^2$. Clearly if no such s exists, then there can be no error functions and the algorithm halts.

Now note, for each i, that

$$\left(\frac{p}{q} + \beta_i\right)' = \left(\frac{p}{q}\right)' = 2\left(\frac{r_i}{s}\right)\left(\frac{r_i}{s}\right)',$$

and therefore

$$\left(\frac{r_i}{s}\right)' = \frac{1}{2}\frac{s}{r_i}\left(\frac{p}{q}\right)'.$$

Thus (4.11) can be written

$$\frac{2g}{s(p/q)'} = \sum \frac{d_i}{r_i}. \tag{4.12}$$

We have, since r_i and r_j are relatively prime for distinct i and j, that the monic denominator of the left side of this equation is the unique monic associate of the product Πr_i. Denoting this denominator by h we have

$$ch = \Pi r_i \tag{4.13}$$

where c is an unknown constant in $\bar{F}$.

Now Theorem 3.2 implies that there can be only one or two terms in the above sum. Thus our strategy is to first attempt to satisfy (4.11) with one error function. If this fails, then two error functions will be tried. Again we label the two steps as follows.

Step R1. If we assume there is one term in (4.11), then (4.13) implies that $ch = r_1$ and, therefore, $p = c^2h^2 - \beta_1 q$. Since h and q are relatively prime (a common divisor would also divide p, an impossibility), we can determine if there exist constants R and S such that $p = Rh^2 + Sq$. If such constants exist, then they must be unique and we have $\beta_1 = -S$. This yields r_1. The constant d_1 can then be calculated from (4.12). Of course, if R and S do not exist or (4.12) does not yield a constant for d_1, then the remaining integral is not a single error function.

Step R2. Next we consider the case where there are two error functions. Here (4.13) yields $ch = r_1 r_2$ and we have

$$c^2h^2 = (p + \beta_1 q)(p + \beta_2 q) \tag{4.14}$$

or

$$p^2 = c^2h^2 - [(\beta_1 + \beta_2)p + (\beta_1\beta_2)q]q.$$

We now claim that $\partial(h^2) > n$ where $n = \max\{\partial(p), \partial(q)\}$. This follows from (4.14) as follows. First note that $\partial(p + \beta q) < n$ only if (i) $\partial p = \partial q$ and $\beta = -lc(p)$, or (ii) $\partial p < \partial q$ and $\beta = 0$. In the first case (4.14) implies that $\partial(h^2) > n$ unless $p + \beta_i q$ is a constant for $i = 1$ or 2. Without loss of generality assume $p + \beta_1 q = \delta$ where δ is a constant and subtract $p + \beta_1 q = \delta$ from $p + \beta_2 q = r_2^2$. This yields $(\beta_2 - \beta_1)q = r_2^2 - \delta$, contradicting Theorem 3.1, since $q = s^2$. In the second case, $\partial(h^2) > n$ unless p is a constant. But if p is a constant, then the equation $p + \beta_2 q = r_2^2$ again contradicts Theorem 3.1.

Now since $\partial[(\beta_1+\beta_2)p+(\beta_1\beta_2)q] \leqq n$, we have $\partial[(\beta_1+\beta_2)p+(\beta_1\beta_2)q] < \partial(h^2)$. We therefore calculate unique polynomials R and S such that $p^2 = Rh^2 + Sq$ where $\partial(R) < \partial(q)$ and $\partial(S) < \partial(h^2)$. This implies that

$$-S = (\beta_1+\beta_2)p+(\beta_1\beta_2)q.$$

We then determine if there exist constants U and V such that $-S = Up + Vq$. If these constants exist, then β_1 and β_2 are roots of the quadratic equation $\beta^2 - U\beta + V$. From here we can calculate r_1 and r_2. Finally, comparing the numerators of (4.12), we obtain an equation of the form $k = d_1r_2 + d_2r_1$ where k is known. From this equation we can determine if constants d_1 and d_2 exist satisfying (4.11).

5. Examples.

Example 5.1. Consider the integral

$$\int e^{1/x}\,dx.$$

Since there are no error functions here (q is not square), (4.1) becomes

$$1 = y + \frac{-1}{x^2}\,y + \sum c_i\left(\frac{p_i'}{p_i} - \frac{q'}{q}\right). \tag{5.1}$$

Since $g = 1$, the elementary part of the integral, if it exists, must be of the form $y = y_1x + y_0$. Substituting this into (5.1), we obtain $y_1 = 1$ and so $y = x + y_0$, which we substitute again into (5.1). This yields an equation of the form

$$\frac{1}{x} = y_0' + \frac{-1}{x^2}\,y_0 + \sum c_i\left(\frac{p_i'}{p_i} - \frac{q'}{q}\right).$$

Comparing partial fraction decompositions, we obtain $y_0 = 0$, thus completing the determination of y. We now have

$$\frac{1}{x} = \sum c_i\left(\frac{p_i'}{p_i} - \frac{q'}{q}\right). \tag{5.2}$$

Since the denominator of $1/x$ does not contain a factor relatively prime to q, we go directly to Step P2. Here $\gamma = 1/x$, $c = -\gamma_\infty/\partial q = -1$, and so

$$\int e^{1/x}\,dx = xe^{1/x} - \mathrm{ei}\left(\frac{1}{x}\right). \qquad \square$$

Example 5.2. Consider the integral

$$\int \left(\frac{1}{x} + \frac{1}{x^2}\right) e^{1/x^2}\,dx.$$

In this case Steps E1, E2, and E3 reveal that there is no elementary part, i.e., $y = 0$, and we go to Step P2 where we have

$$c = -\frac{(1/x + 1/x^2)_\infty}{\partial q} = \frac{-1}{2}.$$

Now consider the error function problem. Equation (4.12) becomes

$$-1 = \sum \frac{d_i}{r_i}.$$

Applying Step R1, we find R and S such that $1 = R + Sq$. This yields $S = 0$, and so $\beta_1 = 0$ and $r_1^2 = 1$. Thus $r_1 = 1$, $d_1 = -1$, and

$$\int \left(\frac{1}{x} + \frac{1}{x^2}\right) e^{1/x^2}\,dx = -\frac{1}{2}\,\mathrm{ei}\left(\frac{1}{x^2}\right) - \mathrm{erf}\left(\frac{1}{x}\right). \qquad \square$$

Example 5.3. Consider the integral

$$\int \frac{x^4-1}{x^5+x}\, e^{(x^2+1)/x}\, dx.$$

Here (4.8) becomes

$$\frac{x^4-1}{x^5+x} = \frac{2x^3}{x^4+1} - \frac{1}{x},$$

and so Step P1 applies. From the resultant $h(\alpha) = \alpha^4 - 4\alpha^2 + 4$ we get $\alpha_1 = \sqrt{2}$ and $\alpha_2 = -\sqrt{2}$. Thus $p_1 = x^2 + \sqrt{2}x + 1$ and $p_2 = x^2 - \sqrt{2}x + 1$, leading to the partial fraction decomposition

$$\frac{2x^3}{x^4+1} = \frac{x - \sqrt{2}/2}{x^2 - \sqrt{2}x + 1} + \frac{x + \sqrt{2}/2}{x^2 + \sqrt{2}x + 1}.$$

Comparing this to (4.8), we obtain $c_1 = c_2 = \frac{1}{2}$. Subtracting

$$\frac{1}{2}\frac{p_1'}{p_1} + \frac{1}{2}\frac{p_2'}{p_2} - \frac{1}{x}\left(\frac{1}{2} + \frac{1}{2}\right)$$

from g, we obtain zero and so

$$\int \frac{x^4-1}{x^5+x}\, e^{(x^2+1)/x}\, dx = \frac{e^{\sqrt{2}}}{2}\,\mathrm{ei}\left(\frac{x^2 - \sqrt{2}x + 1}{x}\right) + \frac{e^{-\sqrt{2}}}{2}\,\mathrm{ei}\left(\frac{x^2 + \sqrt{2}x + 1}{x}\right). \qquad \square$$

Example 5.4. Consider the integral

$$\int e^{(x^4+a)/x^2}\, dx.$$

Here the constant field F contains the parameter a (i.e., $F = Q(a)$). The left side of (4.12) becomes $x^2/(x^4 - a)$, and so $h = x^4 - a$. Solving $p^2 = Rh^2 + Sq$, we obtain $R = 1$ and $S = 4ax^2$. Solving $-4ax^2 = Up + Vq$, we obtain $U = 0$ and $V = -4a$. The roots of $\beta^2 - 4a$ are $2\sqrt{a}$ and $-2\sqrt{a}$, yielding $r_1 = x^2 + \sqrt{a}$ and $r_2 = x^2 - \sqrt{a}$. Calculating d_1 and d_2, we obtain

$$\int e^{(x^4+a)/x^2}\, dx = \frac{1}{2}\, e^{-2\sqrt{a}}\,\mathrm{erf}\left(\frac{x^2 + \sqrt{a}}{x}\right) + \frac{1}{2}\, e^{2\sqrt{a}}\,\mathrm{erf}\left(\frac{x^2 - \sqrt{a}}{x}\right). \qquad \square$$

Appendix. The incomplete Γ functions [Bate53, Chap. 9] are functions that can be expressed in the form

$$\gamma(a, x) = \int_0^x e^{-t} t^{a-1}\, dt, \qquad \mathrm{Re}\, a > 0$$

or

$$\Gamma(a, x) = \int_x^\infty e^{-t} t^{a-1}\, dt = \Gamma(a) - \gamma(a, x).$$

Bateman refers to the extensively studied members of this class as the *special* incomplete Γ functions. Of these, the exponential integral and error function correspond to the parameter values $a = 0$ and $a = \frac{1}{2}$, respectively. Although our algorithm only explicitly handles exponential functions and error functions, each of the other special functions discussed in [Bate53] can be expressed in terms of these two as follows:

(i) The logarithmic integral:

$$\mathrm{li}\,(u) = \int \frac{u'}{\log u}\, dx = \mathrm{ei}\,(\log\,(u)).$$

(ii) The sine and cosine integrals:

$$\operatorname{si}(u) = \int \frac{u' \sin(u)}{u}\, dx = \frac{1}{2i}[\operatorname{ei}(iu) - \operatorname{ei}(-iu)],$$

$$\operatorname{ci}(u) = \int \frac{u' \cos(u)}{u}\, dx = \frac{1}{2}[\operatorname{ei}(iu) + \operatorname{ei}(-iu)].$$

(iii) The Fresnel integrals:

$$S(u) = \int u' \sin\left(\frac{\pi}{2} u^2\right) dx = \frac{-1}{\sqrt{2\pi i}}[\operatorname{erf}(\sqrt{-\pi i/2}\, u) + \operatorname{erf}(\sqrt{\pi i/2}\, u)],$$

$$C(u) = \int u' \cos\left(\frac{\pi}{2} u^2\right) dx = \frac{1}{\sqrt{2\pi i}}[\operatorname{erf}(\sqrt{\pi i/2}\, u) - \operatorname{erf}(\sqrt{-\pi i/2}\, u)].$$

Acknowledgments. I thank D. Coppersmith and B. Trager for the insight I received from their unpublished note on completing squares [CoTr82]. This paper is dedicated to the memory of Dr. Hatem Khalil.

REFERENCES

[Bate53] H. BATEMAN, *Higher Transcendental Functions*, McGraw-Hill, New York, 1953.

[Cher83] G. W. CHERRY, *Algorithms for integrating elementary functions in terms of error functions and logarithmic integrals*, Ph.D. thesis, University of Delaware, Newark, DE, 1983.

[Cher85] ———, *Integration in finite terms with special functions: The error function*, J. Symb. Comput., 1 (1985), pp. 283-302.

[Cher86] ———, *Integration in finite terms with special functions: The logarithmic integral*, SIAM J. Comput., 15 (1986), pp. 1-21.

[CoTr82] D. COPPERSMITH AND B. TRAGER, *Completing the square, revisited*, unpublished manuscript.

[Dav86] J. H. DAVENPORT, *The Risch differential equation problem*, SIAM J. Comput., 15 (1986), pp. 903-918.

[DaTr81] J. H. DAVENPORT AND B. M. TRAGER, *Factorization over finitely generated fields*, in Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation, P. S. Wang, ed., Association for Computing Machinery, New York, 1981.

[Kap57] I. KAPLANSKY, *An Introduction to Differential Algebra*, Hermann, Paris, 1957.

[Know86a] P. H. KNOWLES, *Symbolic integration in terms of logarithmic integrals and error functions*, Ph.D. thesis, North Carolina State University, Raleigh, NC, 1986.

[Know86b] ———, *Integration of Liouvillian functions with special functions*, in Proc. 1986 ACM Symposium on Symbolic and Algebraic Computation, B. W. Char, ed., Association for Computing Machinery, New York, 1986.

[Risch69] R. H. RISCH, *The problem of integration in finite terms*, Trans. Amer. Math. Soc., 139 (1969), pp. 167-189.

[RoCa79] M. ROTHSTEIN AND B. F. CAVINESS, *A structure theorem for exponential and primitive functions*, SIAM J. Comput., 8 (1979), pp. 357-367.

[RoSi77] M. ROSENLICHT AND M. F. SINGER, *On elementary, generalized elementary and Liouvillian extension fields*, in Contributions to Algebra, Bass, Cassidy, and Kovacic, eds., Academic Press, New York, pp. 329-342.

[Roth76] M. ROTHSTEIN, *Aspects of symbolic integration and simplification of exponential and primitive functions*, Ph.D. thesis, University of Wisconsin, Madison, WI, 1976.

[Ssc85] M. F. SINGER, B. D. SAUNDERS, AND B. F. CAVINESS, *An extension of Liouville's theorem on integration in finite terms*, SIAM J. Comput., 14 (1985), pp. 966-990.

[Trag76] B. TRAGER, *Algebraic factoring and rational function integration*, in Proc. 1976 ACM Symposium on Symbolic and Algebraic Computation, R. D. Jenks, ed., Association for Computing Machinery, New York, 1976.

[Vdw50] B. L. VAN DER WAERDEN, *Modern Algebra*, Second edition, Fredrick Ungar, New York, 1950.

[Zwil84] D. ZWILLINGER, *Completing the Lth power in $Z[x]$*, Sigsam Bulletin, 71 (1984), pp. 20-22.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 906-908, October 1989

004

AN ANALYSIS OF A GOOD ALGORITHM FOR THE SUBTREE PROBLEM, CORRECTED*

RAKESH M. VERMA† AND STEVEN W. REYNER‡

Abstract. It is shown that the proof of the main result in Reyner's paper, similarly titled, is incorrect. Interestingly, by combining a simple modification of the algorithm with tighter analysis, one can obtain the original result with a minor improvement.

Key words. algorithm, algorithm analysis, bipartite graphs, computational complexity, matching, subtree

AMS(MOS) subject classification. 68Q25

1. Introduction. The rooted subtree isomorphism[1] problem has several applications in computer science, besides being one of the known subcases of subgraph isomorphism in the complexity class *P*. The main contribution of [2] was the analysis of a very simple algorithm for subtree isomorphism in terms of the complexity of bipartite matching. The result of this analysis had the following important implication: Improvements in the upper bound for bipartite matching would yield corresponding improvements in the upper bound for subtree isomorphism. Unfortunately, however, we show that the proof of Theorem 1 in [2] is incorrect (for the benefit of the reader we include the statement of Theorem 1). More importantly, by modifying the algorithm given by Reyner we are able to prove this theorem. At the same time, our modification preserves the simplicity of the algorithm. The proof of Theorem 2 in [2] is not affected, except that in the last line of the proof for Theorem 2, $0+m-1$ should be $0+n-1$. The statement of Theorem 2 can be obtained from the statement of Theorem 1 by replacing rs^u by rs, omitting $u>1$, and replacing nm^u by $nm \ln n$.

2. Main results.

THEOREM 1 (Reyner [2]). *Given an algorithm for bipartite matching that requires at most $O(rs^u)$ operations where $r \leqq s$ and $u>1$, the subtree algorithm will require at most $O(nm^u)$ operations.*

The original proof is erroneous. The proof starts off by choosing a b large enough so that the matching algorithm requires at most brs^u operations, and so that the subtree algorithm requires at most bnm^u operations whenever n and m satisfy $(n-1)\cdot((m-1)^u+m-1)>nm^u$. It was claimed in [2] that *such a b exists since only finitely many m and n satisfy the above inequality*, and the marriage algorithm is $O(rs^u)$. We show that for all real values of u, $1<u<2$ there are *infinitely* many positive integers n and m such that the inequality holds. Note that since n and m denote the number of vertices of the trees S and T, respectively, the set of positive integers is the only domain of interest for n, m. For convenience, we choose $n=m$, which corresponds to the tree isomorphism problem.

THEOREM 2. *For all $1<u<2$ there is an $m_0(u)$ such that the inequality $(m-1)^{u+1}+(m-1)^2-m^{u+1}>0$ holds for all $m>m_0(u)$ or almost everywhere (a.e.).*

Proof. Expanding $(m-1)^{u+1}$ by the binomial theorem, we need to show that

$$(m-1)^2-m^{u+1}+\sum_{k=0}^{\infty}\binom{u+1}{k}(-1)^k m^{u+1-k}>0 \text{ a.e.}$$

* Received by the editors April 11, 1988; accepted for publication (in revised form) December 21, 1988.
† Department of Computer Science, State University of New York, Stony Brook, New York 11794.
‡ Department of Mathematics, State University of New York, Oswego, New York 13126.
[1] The word "rooted" will be implicit hereafter.

or that

$$m^2-2m+1+\binom{u+1}{2}m^{u-1}-\binom{u+1}{1}m^u+\sum_{k=3}^{\infty}\binom{u+1}{k}(-1)^k m^{u+1-k}>0 \text{ a.e.}$$

Observe that since $1<u<2$, the terms in the last summation are all negative and since $|\binom{u+1}{k}|\leqq 1$ for $k\geqq 3$, the last summation is less in absolute value than $\sum_{k=3}^{\infty} m^{u+1-k}=m^{u-2}(\sum_{k=0}^{\infty} m^{-k})=m^{u-1}/(m-1)<1+\binom{u+1}{2}m^{u-1}$. Therefore it suffices to show that $m^2-2m-(u+1)m^u>0$ almost everywhere. As $2m<2m^u$, we need only prove that $m^2>(u+3)m^u$ almost everywhere. Since $u<2$, the last inequality is true almost everywhere and we are done. □

Remark. The above theorem can also be proved by showing that the more general inequality $1+x^{\varepsilon-1}>(1+1/x)^{k-\varepsilon}$ holds for all real ε $0<\varepsilon<1$, positive integers k, and sufficiently large x [3].

What is more interesting and useful is that by making some modifications to the algorithm originally described by Matula [1], we can prove a slightly stronger theorem than Reyner's main result. If G is any graph, let $v(G)$ denote the number of vertices of G. The basic idea is that the subtree isomorphism problem is trivial when $v(S)$ is 1 or 2. In that case, S is a subtree of any rooted tree T, provided $v(S)\leqq v(T)$. Therefore, we determine for each vertex v in S, T the number of its children $c(v)$, the number of its descendants $d(v)$, and the number of its subtrees with at least two vertices $o(v)$. It should be clear that this takes only $3(n+m)\leqq 6m$ operations. A high-level description of the modified algorithm is as follows. It is assumed that the algorithm is given the roots (root of a tree G is denoted r_g) of the trees to be checked and invoked for trees having at least three vertices.

Algorithm: Rooted Subtree Isomorphism (r_s, r_t)

(1) If $d(r_s)\leqq d(r_t)$ and $c(r_s)\leqq c(r_t)$ and $o(r_s)\leqq o(r_t)$, then delete r_s and r_t to obtain rooted subtrees $S_1,\cdots,S_p$ and $T_1,\cdots,T_q$, else return 0. Here $p=c(r_s)$ and $q=c(r_t)$.

(2) (Recursively) For each S_i such that $v(S_i)>2$ (decide whether S_i is a rooted subtree of T_j with $v(T_j)>2$ and form a $k\times l$ matrix M with $m_{ij}=1(0)$ if S_i is (not) a rooted subtree of T_j). k is the number of such S_i's and l is the number of such T_j's. Relabel the S_i's and T_j's if necessary.

(3) Apply an algorithm for obtaining a maximal matching in the bipartite graph corresponding to M. If there is no such matching, then return 0 else return 1.

We can now prove a slightly stronger theorem than the one given by Reyner.

Proof. Choose c large enough so that the matching algorithm requires at most crs^u for each r and s and $c\geqq 1$. Let d and f be the number of S_i's and T_j's of size at least 3. Then the total number of operations of algorithm Rooted Subtree Isomorphism is given by five comparisons, operations in step 3, and the operations contributed by the recursive calls. Hence the total number of operations is bounded by

$$5+cdf^u+\sum_{i=1}^{d}\sum_{j=1}^{f} cn_i m_j^u\leqq 5+cdf^u+c(n-1)\sum_{j=1}^{f} m_j^u.$$

Since $1\leqq d\leqq \lfloor (n-1)/3\rfloor$ and $m_j\geqq 3$ for $1\leqq j\leqq f$,

$$5+c(n-1)\left(f^u/3+\sum_{j=1}^{f} m_j^u\right)\leqq 5+c(n-1)(f^u/3+(f-1)3^u+(m+2-3f)^u)=g(f).$$

On showing that $g(f)\leqq cnm^u$ we will be done. Also since $1\leqq f\leqq \lfloor (m-1)/3\rfloor$ and the second derivative of $g(f)$ is positive ($u>1$), the maximum occurs at an endpoint.

Therefore an upper bound is the larger of $g(1)$ or $g((m-1)/3)$. Now $g(1) = 5 + c(n-1)(1/3 + (m-1)^u) \leqq cnm^u$ for $m > 2 (n \leqq m, u > 1)$. Also $g((m-1)/3) = 5 + c(n-1)((1/3)((m-1)/3)^u + (m-1)3^{u-1})$ is less than cnm^u if either $u \geqq 1.03$ or $u < 1.03$ and m large. To see this consider $h(m) = 5 + c(n-1)((m-1)^u 3^{-u-1} + (m-1)3^{u-1}) - cnm^u$. After careful analysis it can be shown that the second derivative of $h(m)$ is negative, and its first derivative is negative if $u \geqq 1.03$ or $u < 1.03$ and m is large. □

The entire algorithm (including processing) therefore takes $6m + cnm^u$ operations or about $(c+2)nm^u$ (no preprocessing required if $n < 3$) operations overall (observe that the preprocessing operations are done only once; hence, it is still correct to use $cn_i m_j^u$ for the recursive calls). This is an improvement of the result in Reyner's original article. In the event that a bipartite matching algorithm is obtained with $1 < u < 1.03$, further analysis along these lines might be appropriate.

3. Conclusion. Thus we have been able to show that improving the exponent u in the upper bound for bipartite matching will improve the exponent for subtree isomorphism. The advantage of the algorithm is its obvious simplicity, and when there is a hierarchical structure involved (e.g., trees representing terms), this may be the only algorithm of interest.

REFERENCES

[1] D. W. MATULA, *An algorithm for subtree identification*, SIAM Rev., 10 (1968), pp. 273–274 (abstract).

[2] S. W. REYNER, *An analysis of a good algorithm for the subtree problem*, SIAM J. Comput., 6 (1977), pp. 730–732.

[3] R. M. VERMA, *An error in Reyner's "An analysis of a good algorithm for the subtree problem,"* Tech. Report No. 88/03, Department of Computer Science, State University of New York, Stony Brook, NY, April, 1988.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 909-918, October 1989

005

THE COMPLETE CONVERGENCE OF BEST FIT DECREASING*

WANSOO T. RHEE† AND MICHEL TALAGRAND‡

Abstract. Consider a probability measure μ on $[0, 1]$ and an independent sequence of random variables $X_1, \cdots, X_n, \cdots$ distributed according to μ. No regularity assumptions are made on μ. Denote by $B(X_1, \cdots, X_n)$ the number of unit-size bins that are used by Best Fit Decreasing to pack items of size $X_1, \cdots, X_n$. The existence of a constant $b(\mu)$ is proven such that for each $\varepsilon > 0$,

$$\sum_{n \geq 1} P(|n^{-1}B(X_1, \cdots, X_n) - b(\mu)| \geq \varepsilon) < \infty.$$

Key words. probabilistic analysis, approximation algorithm, bin packing

AMS(MOS) subject classifications. 90B99, 60K30

1. Introduction. The bin-packing problem requires finding the minimum number of unit-size bins needed to pack a given collection of items with sizes in $[0, 1]$. The problem has many applications and has been shown to be NP-complete. There is a growing literature devoted to the analysis of stochastic models for bin packing (see [1]). Most authors have analyzed approximation algorithms under a model of items drawn independently from a special distribution, e.g., uniform on some subinterval of $[0, 1]$. In that case, most approximation algorithms will pack few items in a typical bin and will never produce complicated patterns. Thus, the choice of a restrictive stochastic model may ignore most of the inherent complexity of the algorithm. For example, it is known that many approximation algorithms exhibit anomalous behavior [3]. This behavior does not seem to occur for typical lists drawn at random from the uniform distribution. One way to deepen our understanding could be to consider more general models, where the items are drawn independently according to a distribution μ, on which no restriction is made. (This approach has met some success in [4], [6], [7].) In that case, the typical list can produce very complicated packing patterns. In this paper, we are interested in the algorithm Best Fit Decreasing (BFD). In BFD, we first order the items according to decreasing size. Each item is then packed in the bin in which it fits the best, i.e., in which the remaining space is minimal after adding the item, using a new bin whenever necessary. We refer the reader to [2] for the deterministic analysis of BFD.

We denote by $B(x_1, \cdots, x_n)$ the number of bins used by BFD to pack items of size $x_1, \cdots, x_n$. Our main result is the following convergence property.

THEOREM 1. *For each probability measure μ on $[0, 1]$, there exists a constant $b(\mu)$, such that if $(X_i)_{i \geq n}$ are independent and identically distributed like μ, for each $\varepsilon > 0$, we have*

$$\sum_{n \geq 1} P\left(\left|\frac{1}{n} B(X_1, \cdots, X_n) - b(\mu)\right| \geq \varepsilon\right) < \infty. \tag{1}$$

It might be worth mentioning that while many results in the literature reach a conclusion stronger than (1) they either deal with specific distributions or with much better-behaved algorithms [5], [9].

*Received by the editors March 21, 1988; accepted for publication October 3, 1988.

† Academic Faculty of Management Sciences, Ohio State University, Columbus, Ohio 43210-1399. The research of this author was supported in part by National Science Foundation grant CCR-8801517.

‡ Equipe d'Analyse-Tour 46, Université de Paris VI, 4 Place Jussieu, 75230, Paris, France, and Department of Mathematics, Ohio State University, Columbus, Ohio 43210. The research of this author was supported in part by National Science Foundation grant DMS-8801180.

The problem of complete convergence, i.e., of proving a result corresponding to Theorem 1, seems of interest for all approximation algorithms. In a companion paper [8], we investigate the case of First Fit Decreasing (FFD), which turns out to be considerably deeper. Interestingly enough, there is a considerable difference between these two algorithms. Using the methods of [8], one can show that the pattern BFD produces when packing a typical random list exhibits some kind of convergence. However, this is not needed to prove Theorem 1, which is a simple consequence of the following deterministic result.

PROPOSITION 1. *There exists a function* $\varphi(\delta, n)$ *such that* $\lim_{n\to\infty,\delta\to 0}\varphi(\delta, n)=0$ *with the following property. Consider two lists of items,* $x_1, \cdots, x_n$, *and* $y_1, \cdots, y_{n'}$. *Let*

$$\delta = \sup_{t\geqq 0}\left|\frac{1}{n}\,card\{i\leqq n;\ x_i\geqq t\}-\frac{1}{n'}\,card\{i\leqq n';\ y_i\geqq t\}\right|. \tag{2}$$

Then

$$\left|\frac{1}{n}B(x_1, \cdots, x_n)-\frac{1}{n'}B(y_1, \cdots, y_{n'})\right|\leqq\varphi(\delta, \min(n, n')). \tag{3}$$

It would be of interest to know what is the best possible order of decrease of the function φ. Our method shows that for $n\geqq\delta^{-1}$, one can take $\varphi(\delta, n)=K(\log(1/\delta))^{-1}$ for some universal constant K, and the proof of Theorem 1 given below shows that this implies

$$\left|\frac{1}{n}B(X_1, \cdots, X_n)-b(\mu)\right| = O((\log n)^{-1})\text{a.s.}$$

We see no reason why these estimates would be sharp.

It might be worthwhile to point out that the two-sided control in (2) is essential. Examples in [3] show that we can have $n=n'$, and

$$\forall t\geqq 0,\quad card\{i\leqq n;\ x_i\geqq t\}\geqq card\{i\leqq n;\ y_i\geqq t\},$$

but $B(y_1, \cdots, y_n)=(43/42)B(x_1, \cdots, x_n)$.

Proposition 1 can be interpreted as a kind of *uniform* continuity of BFD. The great difference in nature between BFD and FFD is that such a result does not seem to hold for FFD. (Thus one has to establish a property of (nonuniform) continuity instead.)

Proof of Theorem 1. For $1\leqq i\leqq n'$, define

$$y_i^{n'}=\sup\left\{y;\ 0\leqq y\leqq 1,\ \mu([y, 1])\geqq\frac{i}{n'}\right\}.$$

Then, for each $t\in[0, 1]$,

$$\left|\frac{1}{n'}\,card\{i;\ 1\leqq i\leqq n',\ y_i^{n'}\geqq t\}-\mu([t, 1])\right|\leqq\frac{1}{n'}.$$

Thus, by Proposition 1, the sequence $b_{n'}=B(y_1^{n'}, \cdots, y_{n'}^{n'})$ is a Cauchy sequence. Denote by $b(\mu)$ its limit. Using (3) for $x_i=X_i$, $y_i=y_i^{n'}$, n' large enough, and letting $n'\to\infty$ gives

$$\left|\frac{1}{n}B(X_1, \cdots, X_n)-b(\mu)\right|\leqq\varphi(2\delta(X_1, \cdots, X_n), n),$$

where

$$\delta(X_1, \cdots, X_n)=\sup_{t\geqq 0}\left|\frac{1}{n}\,card\{i\leqq n;\ X_i\geqq t\}-\mu([t, 1])\right|.$$

Consider now $\varepsilon > 0$, and let $\delta_0 > 0$, n_0, such that $\delta \leqq \delta_0$, $n \geqq n_0 \Rightarrow \varphi(2\delta, n) \leqq \varepsilon$. To prove (2), it suffices to show that

$$\sum_{n \geqq 1} P(\delta(X_1, \cdots, X_n) \geqq \delta_0) < \infty.$$

But it is a well-known property of the Kolmogorov-Smirnov statistics that

$$P(\delta(X_1, \cdots, X_n) \geqq t) \leqq 2 \exp(-2nt^2).$$

2. The basic matching procedure. The space left in a bin at any point of the packing procedure is called its vacancy. We agree that we begin the procedure with a sufficient number of size-one bins (say, as many as items). Bins having not yet received an item have a vacancy one. The number of bins having actually received items is the number of vacancies less than 1. The difficulty in analyzing the algorithm BFD is that the packing of any item changes the list of vacancies, and thus influences the sequel of the packing. This difficulty will be solved by dividing the packing into stages in which it has a simpler structure. In order to do that, we need to analyze a simpler procedure called *basic matching*. In this procedure, a list of items $x_1 \geqq \cdots \geqq x_p$ is attributed to a list of (bins with) vacancies $v_1 \leqq \cdots \leqq v_q$. Each x_i is attributed (=matched) in turn, starting by x_1, to the smallest unattributed $v_j \geqq x_i$. The procedure stops when either the list of items or the list of vacancies is exhausted. (For definiteness, say that ties are broken by the index, but this is irrelevant since only the sizes of the items matter.) The following simple lemma will be essential.

LEMMA 1. *Consider the basic matching of two lists* $x_1, \cdots, x_p$, $v_1, \cdots, v_q$. *Then for* $0 \leqq t \leqq u \leqq 1$,

(i) *The number of items* $x_i \geqq t$ *that are attributed to a bin* $v_j \leqq u$ *is given by*

$$\inf_v (card\{i \leqq p;\ t \leqq x_i \leqq v\} + card\{j \leqq q;\ v < v_j \leqq u\})$$

$$= \inf_v (card\{i \leqq p;\ t \leqq x_i < v\} + card\{j \leqq q;\ v \leqq v_j \leqq u\}),$$

where v *is allowed to take any value in* $\mathbf{R}$.

(ii) *The number of items* $x_i \geqq t$ *that are not attributed to a vacancy* $v_j \leqq u$ *is given by*

$$\sup_{t \leqq v \leqq u} (card\{i \leqq p;\ v \leqq x_i \leqq u\} - card\{j \leqq q;\ v \leqq v_j \leqq u\}).$$

(iii) *The number of vacancies less than or equal to* u *that are not attributed an item* $x_i \geqq t$ *is given by*

$$\sup_{t \leqq v \leqq u} (card\{j \leqq q;\ t \leqq v_j \leqq v\} - card\{i \leqq p;\ t \leqq x_i \leqq v\}).$$

Proof. These statements are equivalent and well known (e.g., see [5] for the proof of (ii)).

The main result of the section is as follows.

PROPOSITION 2. *Consider two lists of items,* $x_1, \cdots, x_p$, $x'_1, \cdots, x'_{p'}$, *and two lists of vacancies* $v_1, \cdots, v_q$, $v'_1, \cdots, v'_{q'}$. *Consider* n, n' *(to be thought of as normalizing factors) and* $\delta > 0$. *Assume that*

$$\forall t \geqq 0, \left| \frac{1}{n} card\{i \leqq p;\ x_i \geqq t\} - \frac{1}{n'} card\{i \leqq p';\ x'_i \geqq t\} \right| \leqq \delta, \tag{4}$$

$$\forall t \geqq 0, \left| \frac{1}{n} card\{j \leqq q;\ v_j \geqq t\} - \frac{1}{n'} card\{j \leqq q';\ v'_j \geqq t\} \right| \leqq \delta. \tag{5}$$

For $s \geqq 0$, denote by $M(s)$ (respectively, $M'(s)$) the number of couples (x_i, v_j), $v_j \geqq x_i + s$ (respectively, (x'_i, v'_j), $v'_j \geqq x'_i + s$) that are matched by the basic matching of the lists $x_1, \cdots, x_p$ and $v_1, \cdots, v_q$ (respectively, $x'_1, \cdots, x'_{p'}$ and $v'_1, \cdots, v'_{q'}$). Then

$$\left| \frac{1}{n} M(s) - \frac{1}{n'} M'(s) \right| \leqq \left(10 + \frac{32}{s} \right) \delta. \tag{6}$$

Proof. Denote by S the set of couples (x_i, v_j) that are matched in the basic matching of the lists $x_1, \cdots, x_p$ and $v_1, \cdots, v_q$. By induction, we define a sequence (t_i) as follows. We set $t_0 = 0$, and we define by induction

$$t_{r+1} = \sup \{ t;\ t_r < t \leqq 1,\ S \cap \{(x, v);\ x < t,\ t_r + s < v < x + s\} = \varnothing \}.$$

Obviously, there is a matched couple $(x_{i(r+1)}, v_{j(r+1)})$ such that $x_{i(r+1)} = t_{r+1}$ and $t_r + s < v_{j(r+1)} < t_{r+1} + s$. Thus

$$t_r + s < v_{j(r+1)} < t_{r+1} + s < v_{j(r+2)}.$$

Since $x_{i(r+1)} = t_{r+1} \leqq x_{i(r+2)} = t_{r+2}$ is matched later than $x_{i(r+2)}$ and is matched with $v_{j(r+1)}$, this means that the vacancy $v_{j(r+1)}$ has not yet received an item right after $x_{i(r+2)}$ is packed. Since $t_{r+2} = x_{i(r+2)}$ is matched with $v_{j(r+2)} > v_{j(r+1)}$, we have $t_{r+2} > v_{j(r+1)}$, and hence $t_{r+2} > t_r + s$. This shows that the construction of the points t_r stops with a last point $t_h = 1$, where $h \leqq 1 + 2/s$. Let

$$A = (\{(x, v) \in [0, 1]^2;\ \exists r,\ 0 \leqq r < h;\ t_r < x < t_{r+1},\ t_r + s < v \text{ or } \exists r,\ 0 \leqq r \leqq h,\ x = t_r,\ t_r + s \leqq v\}).$$

First we show that

$$A \supset \{(x, v) \in [0, 1]^2;\ v \geqq x + s\}.$$

Consider (x, v) with $v \geqq x + s$. Let $r < h$ be the smallest such that $x < t_{r+1}$. Then if $x = t_r$, we have $(x, v) \in A$. Otherwise, $x > t_r$, so $v \geqq x + s > t_r + s$ and again $(x, v) \in A$.

Next, we show that each matched couple (x_i, v_j) that belongs to A satisfies $v_j \geqq x_i + s$. This is obvious if $x_i = t_r$. Otherwise, if $t_r < x_i < t_{r+1}$, the definition of t_{r+1} shows that $v_j \geqq x_i + s$. Thus, we have

$$M(s) = card\{(x_i, v_j);\ (x_i, v_j) \text{ is matched},\ (x_i, v_j) \in A\}$$

$$M'(s) \leqq card\{(x'_i, v'_j);\ (x'_i, v'_j) \text{ is matched},\ (x'_i, v'_j) \in A\}.$$

The set A is a disjoint union $A = \bigcup_{i \leqq 2h+1} B_i$ of $2h + 1$ sets of either type

$$B = \{(x, v);\ x = t_r,\ v \geqq t_r + s\} \quad \text{or} \quad B = \{(x, v);\ t_r < x < t_{r+1},\ v > t_r + s\}.$$

For a subset X of $[0, 1]^2$, denote by $N(X)$ (respectively, $N'(X)$) the number of matched couples (x_i, v_j) (respectively, (x'_i, v'_j)) that belong to X. If X is of the type $\{(x, v);\ t \leqq x,\ v \leqq u\}$, it follows from Lemma 1(i), (4), and (5) that

$$\left| \frac{1}{n} N(X) - \frac{1}{n'} N'(X) \right| \leqq 2\delta.$$

It follows that if X is the product of two intervals (that may contain their endpoints or not), we have

$$\left| \frac{1}{n} N(X) - \frac{1}{n'} N'(X) \right| \leqq 8\delta.$$

This holds in particular if X is one of the sets B_i. Thus

$$\frac{1}{n'}M'(s) \leqq \frac{1}{n'}N'(A) \leqq (2h+1)8\delta + \frac{1}{n}N(A)$$

$$\leqq \left(10+\frac{32}{s}\right)\delta + \frac{1}{n}N(A) = \frac{1}{n}M(s) + \left(10+\frac{32}{s}\right)\delta$$

from which (6) follows by symmetry.

Comment. It is easy to improve on the constants 10 and 32. However, we make no attempt at giving sharp constants, but we always give the simplest estimates suitable for our purpose.

3. Stages of packing. We decompose the packing of a list (according to the BFD rule) into "stages" where BFD has a simpler structure. In the first stage, we put all items of size greater than 1/2 each in a single bin. If no such item exists, we do nothing.

After the $(k-1)$th stage has been completed, the largest unpacked item x_1 has a size $2^{-l-1} < x_1 \leqq 2^{-l}$ for some $l = l(k) \geqq 0$. If w_2 denotes the *second* smallest of the vacancies greater than 2^{-l}, it satisfies $rx_1 \leqq w_2 < (r+1)x_1$ for some $r = r(k) \geqq 1$. Denote by $x_1 \geqq x_2 \geqq \cdots \geqq x_a$ the unpacked items greater than 2^{-l-1}. Denote by $v_1 \geqq \cdots \geqq v_q$ the vacancies less than or equal to 2^{-l}, and by $w_1 \leqq \cdots \leqq w_m$ the vacancies greater than 2^{-l}. We now pack in turn $x_1, \cdots, x_a$, following the BFD rule. If an item x_i is attributed to one of the vacancies v_j, we call it an *ordinary* item and an *excess* item otherwise. For an ordinary item, the new vacancy $v_j - x_i$ created by its packing is less than 2^{-l-1}, and thus does not have to be taken into account when packing $x_1, \cdots, x_a$. The first excess item $x_{i(1)}$ (if such an item exists) will be attributed to w_1. If no excess items exists, the kth stage of packing finishes with the packing of x_a. In the sequel of this discussion, we will not mention anymore the case when the items $x_1, \cdots, x_a$ are all packed before the event described occurs. It is understood that in this case, the kth stage finishes with the packing of x_a. The vacancy created by attributing $x_{i(1)}$ to w_1 is $w_1 - x_{i(1)}$.

Case 1. $w_1 < rx_1$. We attribute consecutive excess items $x_{i(1)}, \cdots, x_{i(s)}$ to w_1 until either of the following happens.

Case 1a. For some $s \leqq r$, $w_1 - x_{i(1)} - \cdots - x_{i(s)} \leqq 2^{-l}$. When $x_{i(s)}$ is attributed, we end the kth stage. The first item not packed at the kth stage is $x_b = x_{i(s)+1}$. For the $(k+1)$th stage, w_2 is the smallest vacancy greater than 2^{-l}, and $w_2 \geqq rx_1 \geqq rx_b$. In particular, $r(k+1) \geqq r(k)$.

Case 1b. $w_1' = w_1 - x_{i(1)} - \cdots - x_{i(r)} > 2^{-l}$. When $x_{i(r)}$ is attributed, we end the kth stage. The first item not packed at the kth stage is $x_b = x_{i(r)+1}$. Since $w_1' > 2^{-l} \geqq x_b$, we have $w_1 \geqq (r+1)x_b$, so $w_2 \geqq (r+1)x_b$. Since w_2 is the second largest vacancy greater than 2^{-l} at the $(k+1)$th stage, we have $r(k+1) \geqq r(k)+1$.

Case 2. $w_1 \geqq rx_1$. After r excess items have been attributed to w_1, we obtain a vacancy $u_1 = w_1 - x_{i(1)} - \cdots - x_{i(r)}$ that we call the first *added* vacancy. (The name suggests that this is the first vacancy added to our list that might make the packing of $x_1, \cdots, x_a$ different from the basic matching of the lists $x_1, \cdots, x_a$ and $v_1, \cdots, v_q$.) In a similar fashion, we create our second added vacancy u_2, etc. The first time we meet an item x_b for which there exists an added vacancy $u \geqq x_b$, we do *not* pack x_b, and we declare that the kth stage is finished. (This occurs in particular if we create an added vacancy greater than or equal to 2^{-l}). If w_m is the first vacancy greater than 2^{-l} that received no item during the kth stage, and if x_b is the first item greater than 2^{-l-1} not packed during that stage, we have $x_b \leqq u$, where u is of the type $w_{m_1} - (x_{i(1)} + \cdots + x_{i(r)})$ for $i(1) < \cdots < i(r) < b$, $m_1 < m$, so that $(r+1)x_b \leqq w_{m_1} \leqq w_m$. Since

there is at most one vacancy $2^{-l} < w < w_m$ (it is of the type $w_{m-1} - x_{i(1)} - \cdots - x_{i(s)}$ for $s \leqq r$), we have $r(k+1) \geqq r(k)+1$.

Observation 1. We show that at most 2^{l+2} stages are needed to pack the items of size between 2^{-l} and 2^{-l-1}.

Since $r(k) \leqq 2^{l(k)+1}$, it suffices to show that if $l(k+2) = l(k) = l$, we have $r(k+2) > r(k)$. □

We have shown that $r(k+1) > r(k)$ except possibly in Case 1a, where nonetheless $r(k+1) \geqq r(k)$. Thus it suffices to observe that either $w_3 \geqq (r+1)x_b$, and then $r(k+1) > r(k)$, or that $(r+1)x_b > w_3 \geqq w_2 \geqq rx_b$, and then Case 2 occurs at stage $k+1$, so that $r(k+2) > r(k+1) \geqq r(k)$.

Observation 2. We set $p = b-1$, so that x_p is the last item $> 2^{l-1}$ packed at the kth stage. An alternative way to look at the packing of the items $x_1, \cdots, x_p$ is as follows. We perform the basic matching of the lists $x_1, \cdots, x_p$ and of the list $v_1, \cdots, v_q$ of vacancies less than 2^{-l}. The unmatched items (excess items) are then attributed to the vacancies $w_1 \leqq w_2 \leqq \cdots$ that are greater than or equal to 2^{-l}, putting in each w_j as many consecutive excess items as will fit. In the first case, only w_1 receives excess items. In the second case, all the vacancies $w_1, w_2, \cdots$ that receive excess items receive exactly $r(k)$ items, with the possible exception of the last one.

Thus, if $h+1$ denotes the number of vacancies w_i that receive excess items, we have at most $(h+1)r$ excess items, and the first rh of these items, that we call $g_1, \cdots, g_{rh}$, fit r at a time in turn in the vacancies $w_1, w_2, \cdots, w_h$. Moreover, since x_p is larger than the last added vacancy, we have

$$\begin{aligned} x_p &> w_h - (g_{(r-1)h+1} + \cdots + g_{rh}) \\ &\geqq w_h - rg_{(r-1)h}. \end{aligned}$$

Proposition 3 is the crucial part of our proof. In this proposition, we keep the notations of Observation 2. We consider a list of items $2^{-l} \geqq x_1 \geqq \cdots \geqq x_p > 2^{-l-1}$ that are packed during one stage of the packing in bins with vacancies $(z_i)_{i \leqq n}$.

PROPOSITION 3. *Consider a list of items $x'_1, \cdots, x'_{p'}$ and a list of vacancies $z'_1, \cdots, z'_{n'}$. For $u \geqq 0$, let*

$$\delta(u) = \sup_{t \geqq u} \left| \frac{1}{n} \operatorname{card}\{i \leqq n;\ z_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq n';\ z'_i \geqq t\} \right|. \tag{7}$$

Consider a number $\delta \geqq \max(1/n, 1/n')$, such that $\delta \geqq \delta(2^{-l-1})$, and that

$$\delta \geqq \sup_{t \geqq 0} \left| \frac{1}{n} \operatorname{card}\{i \leqq p;\ x_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq p';\ x'_i \geqq t\} \right|. \tag{8}$$

Denote by $\bar{z}_1, \cdots, \bar{z}_n$ (respectively, $\bar{z}'_1, \cdots, \bar{z}'_{n'}$) the list of vacancies after the items $x_1, \cdots, x_p$) (respectively, $x'_1, \cdots, x'_{p'}$) are attributed according to the BFD *rule to the vacancies $z_1, \cdots, z_n$ (respectively, $z'_1, \cdots, z'_{n'}$). Let*

$$\bar{\delta}(u) = \sup_{t \geqq u} \left| \frac{1}{n} \operatorname{card}\{i \leqq n;\ \bar{z}_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq n';\ \bar{z}'_i \geqq t\} \right|.$$

Then for some universal constant K, we have

$$\begin{aligned} &\bar{\delta}(u) \leqq K\delta \text{ for } u \geqq 2^{-l-1} \text{ and} \\ &\bar{\delta}(u) \leqq K(\max(\delta(u), \delta/u)) \text{ for } u < 2^{-l-1}. \end{aligned} \tag{9}$$

Proof. We will denote by K a universal constant, not necessarily the same in each occurrence. The idea of the proof is to show that within a small perturbation, the

packing of $x'_1, \cdots, x'_{p'}$, according to the BFD rule, has the same structure as the packing of that list by the following procedure. We perform the basic matching of $x'_1, \cdots, x'_{p'}$ with the vacancies $z'_j \leqq 2^{-l}$. The unmatched items are then attributed to the vacancies $z'_j > 2^{-l}$, putting in turn as many items as possible in each such vacancy (which is the procedure by which the items $x_1, \cdots, x_p$ are packed). First we observe that there are at most $n'\delta$ items x'_i that are greater than x_1. BFD packs these items first. Packing them will affect at most $n'\delta$ vacancies. Thus if we consider the list of items and vacancies after these items are packed, (7) and (8) still hold with $\delta(u)+\delta$ and 2δ, respectively, instead of $\delta(u)$ and δ. Thus one can assume that $x'_i \leqq x_1$ for all i. A similar argument shows that we can assume $x_p \leqq x'_i$. We say that an item x'_i is an *excess* item if it is unmatched in the basic matching of the list of items $(x'_i)_{i \leqq p'}$ and of the list of vacancies $\{z'_i;\ i \leqq n',\ z'_i \leqq 2^{-l}\}$. From Lemma 1(iii), (7), and (8), we see that the numbers n_1 (respectively, n_2) of excess items greater than or equal to $g_{r(h-1)}$ (respectively, $< g_{r(h-1)}$) satisfy

$$\frac{n_1}{n'} \leqq \frac{r(h+1)}{n} + K\delta; \qquad \frac{n_2}{n'} \leqq \frac{2h}{n} + K\delta. \tag{10}$$

In the process of packing, items are added to bins. Call a bin *irregular* if at some point of the packing of $x'_1, \cdots, x'_{p'}$ after it has received at least one of these items, its vacancy falls in the interval $[x_p, x_1[$. Call a bin *regular* if it is not irregular.

After a bin becomes irregular, it can accept at most one item. Call these items *irregular*. Call an item *regular* if it is not irregular.

We now observe that the final list of the regular vacancies in the BFD packing of $x'_1, \cdots, x'_{p'}$ is identical to the list of vacancies obtained by the following procedure. We perform the BFD packing of the list $y'_1, \cdots, y'_{p_1}$ of regular items with the list of vacancies z'_i, removing a bin from the list whenever it becomes irregular. Equivalently, we first perform the basic matching of the list $y'_1, \cdots, y'_{p_1}$ with the list of vacancies less than or equal to 2^{-l}. Call the unmatched items the *special* items. As many of the special items as will fit are attributed in turn to the bins with vacancies greater than 2^{-l}, starting with the smallest vacancy.

Our first task is to bound the number of irregular bins, and hence, of irregular items. An irregular bin can be created only when a special item is attributed to a vacancy greater than $2^{-\ell}$. If a special item is attributed to a bin whose original vacancy was in the interval $]2^{-l}, w_1[$, it might create an irregular bin. There are at most $2n'\delta$ such vacancies, so at most that many irregular bins will arise this way.

We recall the inequalities $w_1 \geqq rx_1$, $w_h - rg_{r(h-1)} < x_p$. From (7), we have at least $n'(h/n - 2\delta)$ vacancies in the interval $[w_1, w_h]$. When attributing to these vacancies special items of size greater than of equal to $g_{r(h-1)}$, we never create irregular vacancies. Indeed, if we attribute $(r-1)$ items, we create a vacancy greater than or equal to $w_1 - (r-1)x_1 \geqq x_1$, but if we attribute r items, the vacancy will be less than x_p. Thus in these bins, there is enough room to accept $r\lfloor n'(h/n - 2\delta)\rfloor$ special items greater than or equal to $g_{r(h-1)}$. Observing that special items are excess items, from (10), we see that there is at most $rK\delta n'$ special items left. These might be attributed to vacancies greater than w_h. They will create a maximum of $K\delta n'$ irregular bins. Thus we have at most $K\delta n'$ irregular bins, and hence, at most $K\delta n'$ irregular items. From (7), we see that

$$\forall t \geqq 0, \left| \frac{1}{n} \operatorname{card}\{i \leqq p;\ x_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq p_1;\ y'_i \geqq t\} \right| \leqq K\delta. \tag{11}$$

Thus, we have reduced the problem to the case where we do not pack the items $x'_1, \cdots, x'_{p'}$ packed according to the BFD but where we first perform the basic matching

of these items with the vacancies less than or equal to 2^{-l}, and then attribute the excess (=unmatched) items to the vacancies greater than 2^{-l}. (This is the procedure by which the items $x_1, \cdots, x_p$ are packed.) We can still assume $x_p \leqq x'_i \leqq x_1$ for $i \leqq p'$. In that procedure, vacancies are produced in four different manners, and we will show that for each of the sources of vacancies, an estimate of the type (9) holds. First, there are those of the vacancies of size in $]2^{-l-1}, 2^{-l}]$ that receive no item. For this class, the result follows from Lemma 1(iii). Second, there are the vacancies less than 2^{-l-1}, none of which receive items. For this class, the result is obvious. Third, there are the vacancies created by matching an item and a vacancy less than 2^{-l}. These are all less than 2^{-l-1}. For this class, the result follows from Proposition 2.

Finally, there are the vacancies created by packing the excess items in the vacancies greater than 2^{-l}. If we denote by $g_1 \geqq \cdots \geqq g_b$; $g'_1 \geqq \cdots \geqq g'_{b'}$ the excess items in the basic matching of $x_1, \cdots, x_p$ (respectively, $x'_1, \cdots, x'_{p'}$) and of the vacancies $z_i \leqq 2^{-l}$ (respectively, $z'_i \leqq 2^{-l}$) we see from Lemma 1, (ii), that

$$\forall t \geqq 0, \left| \frac{1}{n} \operatorname{card}\{i \leqq b;\ g_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq b';\ g'_i \geqq t\} \right| \leqq K\delta. \tag{12}$$

Recall also that $rh \leqq b \leqq r(h+1)$. Also, the lists $w_1 \leqq \cdots \leqq w_m$, $w'_1 \leqq \cdots \leqq w'_{m'}$ of vacancies greater than 2^{-l} satisfy

$$\forall t \geqq 0, \left| \frac{1}{n} \operatorname{card}\{i \leqq m;\ w_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq m';\ w'_i \geqq t\} \right| \leqq K\delta. \tag{13}$$

From (12) and (13), we get easily that

$$g_{b(i)} \geqq g'_i \geqq g_{a(i)},\ w_{a(i)} \geqq w'_i \geqq w_{b(i)},$$

where $a(i) = \lceil ni/n' + K\delta \rceil$, $b(i) = \lfloor ni/n' - K\delta \rfloor$, whenever all the quantities are defined. A vacancy $w' \leqq w_h$ will accept at most r excess items greater than or equal to $g_{r(h-1)}$, while a vacancy $w' \geqq w_1$ will accept at least r excess items. Consider a bin with vacancy w'_i. We will have no control of what happens in the bin if either $w'_i < w_1$ or $w'_i > w_h$, or $g'_{ri} \leqq g_{r(h-1)}$, but since $g'_{ri} \geqq g_{a(ri)}$, there are at most $K\delta n'$ such bins. Otherwise, the smallest excess item that fits in the bin is larger than g'_{ri}, so the final vacancy $\bar{w}'_i$ in this bin will be greater than or equal to $w'_i - rg'_{ri}$. Also, if $ni > n'K\delta$, the largest excess item that fits in the bin is less than or equal to $g'_{r(b(i)-1)}$, so the final vacancy $\bar{w}'_i$ is less than or equal to $w'_i - rg'_{r(b(i)-1)}$. For $k \leqq h$, the kth final vacancy produced by the packing of $g_1, \cdots, g_b$ is $\bar{w}_k = w_k - (g_{k(r-1)+1} + \cdots + g_{kr})$. The sequence $\bar{w}_k$ increases. For $t > 0$, the number $n(t)$ of final vacancies in the packing of $g_1, \cdots, g_b$ that are less than t is such that $|n(t) - k| \leqq 1$, where k is the largest such that $\bar{w}_k < t$ (recall that we do not control what happens in the bin w_{h+1}). Thus $\bar{w}'_i < t$ whenever $w'_i \leqq w_k$, $g'_{r(b(i)-1)} \geqq g_{k(r-1)+1}$, and this occurs as soon as $a(i) \leqq k$, $a(r(b(i)-1)) \leqq k(r-1)+1$. Also, $\bar{w}'_i \geqq t$ as soon as $w'_i > w_{k+1}$, $g'_{ri} \leqq g_{(k+1)r}$, and this occurs as soon as $b(i) \geqq k+1$, $a(ri) \geqq (k+1)r$. The conclusion follows easily.

4. Proof of Proposition 2. Consider two lists of items $x_1 \geqq \cdots \geqq x_n$, $x'_1 \geqq \cdots \geqq x'_{n'}$. Set

$$\delta' = \sup_{t \geqq 0} \left| \frac{1}{n} \operatorname{card}\{i \leqq n;\ x_i \geqq t\} - \frac{1}{n'} \operatorname{card}\{i \leqq n';\ x'_i \geqq t\} \right|.$$

We divide the packing of $x_1, \cdots, x_n$ in stages, as described in § 3. Let $x_{b(k)}$ be the first element that is packed at stage k. Let

$$m(k) = \min\left(\lfloor n'b(k)/n \rfloor, \operatorname{card}\{i \leqq n';\ x'_i \geqq x_{b(k)}\}\right).$$

It is easy to see that for each $k \geqq 0$

$$\sup_{t \geqq 0} \left| \frac{1}{n} card\{b(k) \leqq i < b(k+1);\ x_i \geqq t\} - \frac{1}{n'} card\{m(k) \leqq i < m(k+1);\ x'_i \geqq t\} \right| \leqq \delta,$$

where $\delta = 2(\delta' + 1/n + 1/n')$.

We start the packing of $x_1, \cdots, x_n$ (respectively, $x'_1, \cdots, x'_{n'}$) with n (respectively n') empty bins. Denote by $z_{1,k}, \cdots, z_{n,k}$ (respectively, $z'_{1,k}, \cdots, z'_{n',k}$) the list of vacancies after the items $x_1, \cdots, x_{b(k)-1}$ (respectively, $x'_1, \cdots, x'_{m(k)-1}$) have been packed according to the BFD rule. Set

$$\delta_k(u) = \sup_{t \geqq u} \left| \frac{1}{n} card\{i \leqq n;\ z_{i,k} \geqq t\} - \frac{1}{n'} card\{i \leqq n';\ z'_{i,k} \geqq t\} \right|.$$

Denote by K the constant of Proposition 3. We recall that when the last element of size in $]2^{-l-1}, 2^{-l}]$ is packed, a new stage begins. Consider the following statement:

$H(l, m)$: If k is such that m stages have occurred since the last item x_i of size greater than 2^{-l} was packed, then

$$\delta_k(u) \leqq K^{m+2^{l+2}} 2^{(l+1)(l+2)/2} \delta \quad \text{if } u \geqq 2^{-l-1},$$
$$\delta_k(u) \leqq u^{-1} K^{m+2^{l+2}} 2^{(l+1)(l+2)/2} \delta \quad \text{if } u < 2^{-l-1}.$$

When l is fixed, this statement follows by induction over m from Proposition 3. Since 2^{l+2} stages are enough to pack all the items of size in $]2^{-l-1}, 2^{-l}]$ and after all these items are packed, we have

$$\delta_k(u) \leqq K^{2^{l+3}} 2^{(l+1)(l+2)/2} \delta \quad \text{if } u \geqq 2^{-l-1},$$
$$\delta_k(u) \leqq u^{-1} K^{2^{l+3}} 2^{(l+1)(l+2)/2} \delta \quad \text{if } u < 2^{-l-1}.$$

Thus in particular

$$\delta_k(u) \leqq K^{2^{l+3}} 2^{(l+2)(l+3)/2} \delta \quad \text{if } u \geqq 2^{-l-2}, \tag{14}$$
$$\delta_k(u) \leqq u^{-1} K^{2^{l+3}} 2^{(l+2)(l+3)/2} \delta \quad \text{if } u < 2^{-l-2},$$

which is $H(l+1, 0)$. Thus the statement $H(l, 0)$ follows by induction over l.

Let $k(p)$ be the smallest such that $x_{b(k(p))} \leqq 2^{-p}$. Then by definition of $m(k)$, $x'_{m(k(p))} \leqq x_{b(k(p))} \leqq 2^{-p}$. Now (14) shows that for $n, n' \geqq \delta'^{-1}$, $\delta_{k(p)}(2^{-p}) \leqq K^{2^p} \delta'$ (for a new constant K). Denote by $B(p)$ (respectively, $B'(p)$) the number of bins that have received items before $x_{b(k(p))}$ (respectively, $x'_{m(k(p))}$) is packed. Since $B(p) = n - card\{i \leqq n;\ z_{i,k(p)} < 1\}$, and since a similar formula holds for $B'(p)$, we have

$$\left| \frac{B(p)}{n} - \frac{B'(p)}{n'} \right| \leqq K^{2^p} \delta'.$$

Obviously $B(p) \leqq B(x_1, \cdots, x_n) \leqq B(p) + \lceil 2^{-p} n \rceil$. A similar formula for $B'(p)$ implies

$$\left| \frac{1}{n} B(x_1, \cdots, x_n) - \frac{1}{n'} B'(x'_1, \cdots, x'_n) \right| \leqq K^{2^p} \delta' + 2^{-p} + \frac{1}{n} + \frac{1}{n'}.$$

This concludes the proof. (The choice of p such that $2^p \sim \alpha(\log 1/\delta')$ for α small enough gives the quantitative estimate mentioned in the Introduction.)

REFERENCES

[1] E. G. COFFMAN, JR., G. S. LUEKER, AND A. H. G. RINNOOY KAN, *An introduction to the probabilistic analysis of sequencing and packing heuristics*, Management Sci., 34 (1988), pp. 266–290.

[2] D. S. JOHNSON, A. DEMERS, J. D. ULLMAN, M. R. GAREY, AND R. L. GRAHAM, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM J. Comput., 3 (1974), pp. 299–325.
[3] F. D. MURGOLO, *Anomalous behavior in bin packing algorithms*, Discrete Appl. Math. (1989), to appear.
[4] W. RHEE, *Optimal bin packing with items of random sizes*, Math. Oper. Res., 13 (1988), pp. 140–151.
[5] ———, *Stochastic analysis of a modified first fit decreasing packing*, 1988, preprint.
[6] W. RHEE AND M. TALAGRAND, *Optimal bin packing with items of random sizes* II, SIAM J. Comput., 18 (1989), pp. 139–151.
[7] ———, *Optimal bin packing with items of random sizes* III, SIAM J. Comput., 18 (1989), pp. 473–486.
[8] ———, *Complete convergence of first fit decreasing*, SIAM J. Comput., 18 (1989), this issue, pp. 919–938.
[9] P. W. SHOR, *The average-case analysis of some on-line algorithms for bin packing*, Combinatorica, 6 (1986), pp. 179–200.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 919-938, October 1989

006

THE COMPLETE CONVERGENCE OF FIRST FIT DECREASING*

WANSOO T. RHEE† AND MICHEL TALAGRAND‡

Abstract. Consider a probability measure μ on $[0, 1]$ and an independent sequence of random variables $X_1, \cdots, X_n, \cdots$ distributed according to μ. No regularity assumptions are made on μ. Denote by $F_n(X_1, \cdots, X_n)$ the number of unit-size bins that are used by First Fit Decreasing to pack $X_1, \cdots, X_n$. The existence of a constant $f(\mu)$ is proven such that for each $\varepsilon > 0$, we have

$$\sum_{n \geqq 1} P(|n^{-1}F_n(X_1, \cdots, X_n) - f(\mu)| \geqq \varepsilon) < \infty.$$

Key words. probabilistic analysis, approximation algorithm, bin packing

AMS(MOS) subject classifications. 90B99, 60K30

1. Introduction. The bin-packing problem requires finding the minimum number of unit-size bins needed to pack a given collection of items with sizes between zero and one. The problem has many applications and has been shown to be NP-complete. There is a growing literature devoted to the analysis of stochastic models for bin packing (see the survey paper [2]). Most authors have analyzed approximation algorithms under a model of items drawn independently from a special distribution, e.g., uniform on some subinterval of $[0, 1]$. In this paper, we are interested in the algorithm First Fit Decreasing (FFD). In FFD, we first order the items according to decreasing sizes. We start with an ordered collection of unit-size bins, and we put each item in turn in the first bin in which it fits. We refer the reader to [4] for the worst-case analysis of FFD. In the model where the items are drawn independently from the uniform distribution on $[0, 1]$, it is easy to see that the expected wasted space by FFD is $\Theta(\sqrt{n})$ [6]. When the distribution has a decreasing density on $[0, 1/2]$, deep recent results show that the expected wasted space is bounded [1], [3]. Two challenging aspects of FFD are (1) that it can produce very complicated patterns and (2) that it has erratic behavior, in the sense that the packing of a list with smaller items can use more bins. These interesting aspects completely disappear in the two models mentioned above, where the patterns are simple and where the good behavior of FFD is obtained by showing that it uses fewer bins than a much better behaved algorithm called mFFD. The algorithm mFFD has been analyzed by Rhee [9] in the same model as the present paper, and we refer the reader to [9] to fully appreciate the dramatic difference in complexity between mFFD and FFD. In our model, the items will be distributed according to a distribution μ, on which no restriction whatsoever is made. Thus, it can happen that the packing by FFD of a typical random list is very complicated. Nonetheless, our proof will show that the packing pattern of a typical random list exhibits some kind of convergence. This is not obvious beforehand, since one can find lists for which a tiny decrease in the size of a single item produces a quite long packing. Such lists are, however, exceptional and, on average, FFD behaves reasonably well.

The complete convergence result claimed in the abstract is an immediate consequence of the properties of the Kolmogorov statistic and of the following deterministic result.

* Received by the editors March 21, 1988; accepted for publication October 3, 1988.

† Academic Faculty of Management Sciences, Ohio State University, Columbus, Ohio 43210-1399. The research of this author was supported in part by National Science Foundation grant CCR-8801517.

‡ Equipe d'Analyse-Tour 46, Université de Paris VI, 4 Place Jussieu, 75230, Paris, France, and Department of Mathematics, Ohio State University, Columbus, Ohio 43210. The research of this author was supported in part by National Science Foundation grant DMS-8801180.

THEOREM 1. *For each probability measure μ on $[0, 1]$, there exists a number $f(\mu)$, such that for each $\varepsilon > 0$, there exists $\delta > 0$, $N_0 \geqq 1$, such that for each sequence $(z_1, \cdots, z_N)$ of items, $N \geqq N_0$, that satisfies*

$$(1) \qquad \forall t \leqq 1, \quad \left| \frac{1}{N} \operatorname{card}\{i \leqq N;\ z_i \geqq t\} - \mu([t, 1]) \right| \leqq \delta,$$

we have

$$\left| \frac{1}{N} F_N(z_1, \cdots, z_N) - f(\mu) \right| \leqq \varepsilon.$$

Our proof will contain explicit constructions that allow, at least in principle, us to compute $f(\mu)$ when μ is given.

One can look at Theorem 1 as a continuity property of FFD. It should be noted that, given $\varepsilon > 0$, the choice of δ given by our proof will critically depend on μ. The existence of the pathological lists mentioned above shows that given ε, δ cannot be chosen independently of μ. This is an interesting contrast with the case of Best Fit Decreasing that is studied in the companion paper [10].

The principle of the approach is very simple. The first step is to define for measures a "packing" operation that mimicks FFD. Then one shows that the packing of lists of items that satisfy (1) proceeds very much as the "packing" of measure. To complete this program, we will, however, have to develop new tools and to proceed to an in-depth study. This might not be suprising in view of the delicate nature of FFD. (Working with measures instead of lists of items was the philosophy of [7], [8], and [9].)

2. The basic matching procedure. The space left in a bin at any point of the packing is called its vacancy. For our purposes, only the vacancies matter. Thus, for simplicity, we will talk of the list of vacancies and use expressions such as "attribute an item to a vacancy," which has the obvious meaning. We will denote items and their sizes by the same letter.

Each time a bin receives an item, the list of vacancies changes, which influences the way in which later items are packed. This difficulty will be solved in § 3 by dividing the packing into steps that have a simpler structure. Our first task is to analyze a simpler procedure that we call basic matching. In this procedure, we have a sorted list of items $z_1 \geqq \cdots \geqq z_p$ and a number $\alpha \geqq z_1$. We have an ordered list of vacancies $v_1, v_2, \cdots, v_N$. For each vacancy, we define its *multiplicity* by $m_j = \lfloor v_j/\alpha \rfloor$ if $v_j \geqq 2\alpha$ and $m_j = 1$ otherwise. To each vacancy v_j, we attribute in turn the largest unattributed item $z_i \leqq v_j$ if $m_j = 1$ and the m_jth largest unattributed item z_i otherwise.

Comment. (1) To see the relevance of such a procedure, let us recall the following fact brought to light in [1]. When attributing the sorted items $z_1, \cdots, z_p$ to the vacancies $v_1, \cdots, v_N$ by the FFD rule, one can think of the bins being filled in turn. To fill each bin, the remaining items are examined in turn, and each item that fits is put in the bin.

(2) Basic matching is rather different from FFD. In particular, a vacancy v_i of size $2\alpha - \varepsilon$ is always attributed at most one item, while if, when its turn comes to be filled, the largest unattributed item is of size less than or equal to $v_i/2$, FFD should attribute to it at least two items. But the basic matching will be used only on suitably restricted lists for which this difference plays no role.

Our starting point is the following lemma which is probably known.

LEMMA 1. *The number of items of size greater than or equal to t that are attributed to a vacancy v_j, $j \leqq k$ is given by*

$$(2) \qquad \inf_{u \geqq t} \{\operatorname{card}\{i \leqq p;\ t \leqq z_i < u\} + \sum \{m_j;\ j \leqq k,\ v_j \geqq u\}\}.$$

In this formula, values of $u > 1$ are allowed, giving a value card$\{i \leqq p;\ t \leqq z_i\}$.

Proof. By replacing in the list each vacancy $v_j > \alpha$ by m_j consecutive vacancies of size α, we can assume that $m_j = 1$ for each j. If the sequence $(v_j)_{j \leqq k}$ decreases, (2) is proved, e.g., in [9, Lem. 1]. For the general case, it suffices to show that transposing two consecutive vacancies v_i, v_{i+1}, $i+1 \leqq k$, does not change the collection of items that are attributed to vacancies of index less than or equal to k. Indeed, we can go from the decreasing case to the general case by performing a finite number of such transpositions. If, when v_i is attributed its item, there remains one item of size s such that $\min(v_i, v_{i+1}) < s \leqq \max(v_i, v_{i+1})$, the transposition changes nothing. Otherwise, it simply exchanges the items attributed to v_i and v_{i+1}. □

Consider a positive measure ν on $[0, 1] \times [0, 1]$. For simplicity, set $H(x, t) = \nu([0, x] \times [t, 1])$. The idea is that for a long sequence of vacancies, $(v_i)_{i \leqq N}$, $NH(x, t)$ is the sum of the multiplicities of the vacancies of index less than or equal to N_x and size greater than or equal to t. Consider a positive measure μ on $[0, 1]$. The idea is that $N\mu([t, 1]) = card\{i \leqq p;\ z_i \geqq t\}$ for our list of items under consideration.

We observe the following immediate properties. The first expresses the fact that $\nu(]x, y] \times [a, b[) \geqq 0$.

$$0 \leqq x \leqq y \leqq 1,\ 0 \leqq a \leqq b \leqq 1, \Rightarrow H(y, a) + H(x, b) \geqq H(x, a) + H(y, b). \tag{3}$$

$H(x, t)$ increases in x, decreases in t. If $t_n \nearrow t$, $x_n \searrow x$, we have

$$H(x, t) = \lim_{n \to \infty} H(x_n, t_n). \tag{4}$$

In view of (2), it is natural to consider the function

$$G(x, t) = \inf_{u \geqq t} (H(x, u) + \mu([t, u[)).$$

In this formula, values of $u > 1$ are allowed, giving a right-hand side $\mu([t, 1])$.

LEMMA 2. (a) *G increases in x, decreases in t.*

(b) *For $s \geqq t$,*

$$G(x, t) \leqq \mu([t, s[) + G(x, s). \tag{5}$$

Proof. Lemma 2(a) is obvious. For (b), observe that for any $u \geqq s$,

$$G(x, t) \leqq H(x, u) + \mu([t, u[) = \mu([t, s[) + H(x, u) + \mu([s, u[),$$

which implies (5).

LEMMA 3. *If $x_n \searrow x$, $t_n \nearrow t$, then $G(x, t) = \lim_{n \to \infty} G(x_n, t_n)$.*

Proof. Let $u \geqq t$. Then, since $u \geqq t_n$,

$$G(x_n, t_n) \leqq H(x_n, u) + \mu([t_n, u[)$$

so that by (4)

$$\limsup_{n \to \infty} G(x_n, t_n) \leqq H(x, u) + \mu([t, u[)$$

and hence

$$\limsup_{n \to \infty} G(x_n, t_n) \leqq G(x, t).$$

Since $G(x, t) \leqq G(x_n, t_n)$, the proof is complete.

From Lemmas 2 and 3, we see that we can define a positive measure μ_z by

$$\mu_z([t, s[) = \mu([t, s[) + G(z, s) - G(z, t). \tag{6}$$

This represents the distribution of items that have not been attributed to vacancies of index less than or equal to Nz.

A nice feature of basic matching is that the items attributed to the vacancies of index greater than k depend only on the list of items that have not been attributed to the vacancies of index less than or equal to k. We now prove a result of the same nature. This useful technical point will be of much help. We define, for $x \geqq z$,

$$H_z(x, t) = H(x, t) - H(z, t) = \nu(]z, x] \times [t, 1]),$$

and we set

$$G_z(x, t) = \inf_{u \geqq t} (H_z(x, u) + \mu_z([t, u[)). \tag{7}$$

PROPOSITION 1. *For* $x \geqq z$,

$$G(x, t) = G(z, t) + G_z(x, t).$$

Proof. In (7), we substitute the values of μ_z and H_z. After cancellation, we obtain

$$G(z, t) + G_z(x, t) = \inf_{u \geqq t} (\mu([t, u[) + H(x, u) + G(z, u) - H(z, u)).$$

By definition of $G(z, u)$, we obtain

$$G(z, t) + G_z(x, t) = \inf_{v \geqq u \geqq t} (\mu([t, v[) + H(x, u) - H(z, u) + H(z, v)).$$

Taking $v = u$, we get

$$G(z, t) + G_z(x, t) \leqq \inf_{u \geqq t} (\mu([t, u[) + H(x, u)) = G(x, t).$$

On the other hand, since $z \leqq x$, $u \leqq v$, (3) shows that

$$H(x, u) - H(z, u) + H(z, v) \geqq H(x, v),$$

and thus

$$G(z, t) + G_z(x, t) \geqq \inf_{v \geqq u \geqq t} (\mu([t, v[) + H(x, v)) = G(x, t). \qquad \square$$

In the sequel, we will always assume that $\nu(\{x\} \times [0, 1]) = 0$ for all x. In particular, $\mu_0 = \mu$, $G(0, t) = 0$ for all t.

COROLLARY 1. (a) *For* $y \leqq z$, $\mu_z \leqq \mu_y$.

(b) *For* $y \leqq z$, $s \leqq t$,

$$G(y, s) - G(y, t) \leqq G(z, s) - G(z, t). \tag{8}$$

(c) *For* $y \leqq z$,

$$G(z, 0) - G(y, 0) \leqq H(z, 0) - H(y, 0). \tag{9}$$

Proof. By Proposition 1, it suffices to consider the case $y = 0$, in which case this is obvious. $\square$

From Lemma 3, and (8), we conclude that there exists a positive measure η on $[0, 1] \times [0, 1]$ such that $G(x, t) = \eta([0, x] \times [t, 1])$ (see [5, §§ 4-5]). We note the formula

$$\eta(]x, y] \times [s, t[) = G(y, s) + G(x, t) - G(y, t) - G(x, s). \tag{10}$$

From (9), we observe that

$$\eta(]x, y] \times [0, 1]) \leqq \nu(]x, y] \times [0, 1]) \tag{11}$$

for $x \leqq y$. In particular, since we assume $\nu(\{x\} \times [0, 1]) = 0$, we have $\eta(\{x\} \times [0, 1]) = 0$ for all x. Thus, when computing with η, we can always disregard any set that is contained in a countable union of vertical lines, since such a set is negligible.

The measure η can be thought of as the distribution of the couple $(j/N, x_j)$, where x_j is the size of the item(s) attributed to the vacancy v_j. Unfortunately, we are not interested in that, but in the new list of vacancies after the basic matching has been completed. Although this information is contained in η, to extract it in a useful way will require the development of more tools.

We define

$$\varphi(x, t) = \inf \{u;\ 0 \leqq u < t,\ \mu_x(]u, t]) = 0\}$$

when this set is not empty, and $\varphi(x, t) = t$ otherwise. Thus we have

$$\mu_x(]\varphi(x, t), t]) = 0; \qquad \forall u < t,\ \mu_x(]u, t]) > 0. \tag{12}$$

LEMMA 4. *Consider $y > x$. Then*

$$u > \varphi(x, t);\ t \geqq \varphi(y, u) \Rightarrow \varphi(y, u) \leqq \varphi(x, t).$$

Proof. From (12), and Corollary 1, we have

$$\mu_y(]\varphi(x, t), t]) \leqq \mu_x(]\varphi(x, t), t]) = 0,$$

and from (12) we also have

$$\mu_y(]\varphi(y, u), u]) = 0.$$

Hence, since $u > \varphi(x, t)$, $t \geqq \varphi(y, u)$, we have

$$\mu_y(]\varphi(x, t), u]) = 0,$$

so $\varphi(y, u) \leqq \varphi(x, t)$ by definition of $\varphi(y, u)$. □

We will prove later that "the item size attributed to a vacancy v_i of size t is either t or $\varphi(i/N, t)$." The following is the main step toward that result.

LEMMA 5. *Let $0 \leqq b < d \leqq 1$. Then*

$$\nu(\{(x, t);\ t < d,\ \varphi(x, t) > b\}) \leqq \mu(]b, d[). \tag{13}$$

Proof. Step 1. We define

$$h(x) = \sup \{t;\ \mu_x(]b, t[) = 0\}$$

if this set is not empty, $h(x) = b$ otherwise. Then

$$\mu_x(]b, h(x)[) = 0,$$

and

$$\forall u > h(x), \qquad \mu_x(]b, u[) > 0. \tag{14}$$

If $\varphi(x, t) > b$, by definition of $\varphi(x, t)$, we have $\mu_x(]b, t]) > 0$, and hence $\mu_x([h(x), t]) > 0$ if $h(x) > b$. So we have

$$\begin{aligned} &\{(x, t);\ t < d,\ \varphi(x, t) > b\} \\ &\qquad = \{(x, t);\ h(x) = b,\ t > b \text{ or } h(x) > b,\ \mu_x([h(x), u]) \geqq 0\} \\ &\qquad \subset A' =: \{(x, t);\ t > h(x) \text{ or } t = h(x) > b,\ \mu_x(\{h(x)\}) > 0\}. \end{aligned}$$

Hence, it is enough to prove that $\nu(A') \leqq \mu(]b, d[)$. It might happen that ν charges (i.e., gives positive measure to) the set of points $(x, h(x))$ for which $\mu_x(\{h(x)\}) > 0$. Since a careful accounting of these points is needed, this makes the proof very tedious.

Step 2. Fix x such that $h(x) > b$. Let $t = h(x)$. Consider a sequence $v_n \geqq t$ such that

$$G(x, t) = \lim_{n\to\infty} \mu([t, v_n[) + H(x, v_n).$$

Without loss of generality, we can assume that (v_n) converges. We claim that $\lim_{n\to\infty} v_n = t$. For, otherwise, taking w with $t < w < \lim_{n\to\infty} v_n$, we would have

$$G(x, t) = \mu([t, w[) + \lim_{n\to\infty} (\mu([w, v_n[+ H(x, v_n))) \geqq \mu([t, w[) + G(x, w),$$

i.e., $\mu_x([t, w[) = 0$, so that $\mu_x(]b, w[) = 0$, which contradicts (14).

Since $t = \lim_{n\to\infty} v_n$, we get

$$G(x, t) = \min(H(x, t), \mu(\{t\}) + H^+(x, t)), \tag{15}$$

where, for simplicity, we set $H^+(x, t) = \lim_{u\to t^+} H(x, u) = \nu([0, x] \times]t, 1])$. Thus, for $u > t$,

$$\min(H(x, t) - H(x, u), \mu(\{t\}) + H^+(x, t) - H(x, u)) = G(x, t) - H(x, u)$$
$$\leqq G(x, t) - G(x, u) \leqq \mu([t, u[)$$

or, equivalently,

$$\nu([0, x] \times]h(x), u[) + \min(\nu([0, x] \times \{h(x)\}), \mu(\{h(x)\})) \leqq \mu([h(x), u[). \tag{16}$$

This holds for all $u > t$, and hence

$$\nu([0, x] \times]h(x), u]) + \min(\nu([0, x] \times \{h(x)\}), \mu(\{h(x)\})) \leqq \mu([h(x), u]). \tag{17}$$

Step 3. When $\mu_x(\{h(x)\}) > 0$, we have (still with $t = h(x)$) $G(x, t) - G^+(x, t) < \mu(\{t\})$. Since $G^+(x, t) \leqq H^+(x, t)$, by (15), we have

$$\mu(\{t\}) > G(x, t) - G^+(x, t)$$
$$\geqq G(x, t) - H^+(x, t) = \min(H(x, t) - H^+(x, t), \mu(\{t\})),$$

and then $H(x, t) - H^+(x, t) < \mu(\{t\})$. So

$$A' \subset A =: \{(x, t);\ t > h(x) \quad \text{or } t = h(x) > b,\ H(x, t) - H^+(x, t) < \mu(\{t\})\}.$$

Hence, it is enough to prove that $\nu(A) \leqq \mu(]b, d[)$.

Step 4. Suppose now (with $t = h(x)$) that $H^+(x, t) + \mu(\{t\}) \leqq H(x, t)$. Hence, by (5), (15), for $u > t$,

$$H^+(x, t) + \mu(\{t\}) = G(x, t) \leqq \mu([t, u[) + G(x, u).$$

Thus, $H^+(x, t) \leqq \mu(]t, u[) + G(x, u)$, and letting $u \to t$, we have $H^+(x, t) \leqq G^+(x, t)$, so $G^+(x, t) = H^+(x, t)$, since $G \leqq H$. Hence for $u > t$, by (5),

$$H^+(x, t) - H(x, u) = G^+(x, t) - H(x, u)$$
$$\leqq G^+(x, t) - G(x, u) \leqq \mu(]t, u[)$$

or, equivalently

$$\nu([0, x] \times]h(x), u[) \leqq \mu(]h(x), u[). \tag{18}$$

Since this holds for $u > t$, we also have

$$\nu([0, x] \times]h(x), u]) \leqq \mu(]h(x), u]). \tag{19}$$

Step 5. Consider $k>1$. We denote by $h_1<\cdots<h_{m-1}$ the *different* values taken by h at the points $i2^{-k}$, $0\leqq i\leqq 2^k$, and we set $h_m=d$. For $l\leqq m-1$, we set $X_l=\{x;\ h(x)=h_l\}\neq\varnothing$, and $x_l=\inf X_l$, $x'_l=\sup X_l$, so that $x'_{l-1}\leqq x_l\leqq x'_l\leqq x_{l+1}$. For $0\leqq l\leqq m-1$, we set

$$y_l=\sup\{y\leqq 1;\ H(y,h_l)-H^+(y,h_l)<\mu(\{h_l\})\}$$
$$=\sup\{y\leqq 1;\ \nu([0,y]\times\{h_l\})<\mu(\{h_l\})\}.$$

Setting $x'_0=0$, for $1\leqq l\leqq m-1$, we define

$$z_l=\begin{cases}\min(y_l,x'_l) & \text{if } y_l>x_l\\ x'_{l-1} & \text{otherwise.}\end{cases}$$

We set

$$A_k=\{(x,t);\ \exists l,\ 1\leqq l<m,\ t>h_l,\ x<x'_l \text{ or } t=h_l>b,\ x<z_l\}.$$

The sequence (A_k) obviously increases, and $A\backslash\bigcup_k A_k$ is contained in the countable collection of vertical line whose abscissa n has one of the following properties:

$$x=\sup\{z;\ H(z,t)-H^+(z,t)<\mu(\{t\})\},\quad \text{where } \mu(\{t\})>0,$$

or

$$h \text{ is not continuous at } x.$$

Thus, $\nu(A\backslash\bigcup A_k)=0$, and hence it is enough to show that for each k, we have $\nu(A_k)\leqq\mu(]b,d[)$.

Step 6. We define, for $0\leqq l<m-1$,

$$B'_l=\{(x,t);\ h_l<t<h_{l+1},\ x<x'_l\}$$

$$C_l=\begin{cases}\{(x,t);\ t=h_l,\ x<z_l\} & \text{if } y_l>x_l,\ h_l>b\\ \varnothing & \text{otherwise}\end{cases}$$

$$D_l=\begin{cases}\{(x,t);\ t=h_l,\ x<x'_{l-1}\} & \text{if } y_l\leqq x_l,\ h_l<d\\ \varnothing & \text{otherwise.}\end{cases}$$

Observe that by definition of y_l,

$$C_l\neq\varnothing\Rightarrow\nu(C_l)\leqq\min(\mu(\{h_l\}),\nu([0,x'_l]\times\{h_l\})). \tag{20}$$

Let $B_l=C_l\cup B'_l\cup D_{l+1}$, so that $A_k=\bigcup_{l=1}^{m-1}B_l$. Define, for $l\leqq m+1$,

$$\gamma_l=\mu(]b,h_l]) \quad \text{if } C_l=\varnothing,\ D_l\neq\varnothing,$$

and

$$\gamma_l=\mu(]b,h_l[) \quad \text{if } D_l=\varnothing,\ C_l\neq\varnothing.$$

To prove that $\nu(A_k)\leqq\mu(]b,d[)$ it is enough to show that $\nu(B_l)\leqq\gamma_{l+1}-\gamma_l$. In the case $C_l\neq\varnothing$, $D_{l+1}\neq\varnothing$ (respectively $C_l\neq\varnothing$, $D_{l+1}=\varnothing$; $C_l=\varnothing$, $D_{l+1}\neq\varnothing$; $C_l=\varnothing$, $D_{l+1}=\varnothing$) this follows from (20) and (17) (respectively (16), (19), (18)) applied with $u=h_{l+1}$, $x\in X_l$, and having $x\to x'_l$. □

We now consider a positive measure λ on $[0,1]$ and a Borel measurable map $\theta:[0,1]\to[0,1]$. We suppose that ν is the image of λ by the map $x\to(x,\theta(x))$. This means that for each Borel subset G of $[0,1]^2$ we have

$$\nu(G)=\lambda(\{x\in[0,1];\ (x,\theta(x))\in G\}).$$

We define $\psi(x)=\varphi(x,\theta(x))$.

THEOREM 2. *When ν is as above, η is supported by the union of the graph of θ and of the graph of ψ.*

Proof. Consider $\gamma > 0$. By Lusin's theorem [12, p. 56], we can find a compact subset K of $[0, 1]$ such that $\lambda([0, 1]\backslash K) \leqq \gamma$, and that θ and ψ are continuous on K. Thus, we can find a sequence $x_0 = 0 < x_1 < x_2 < \cdots < x_n = 1$, and numbers $(a_i)_{i\leqq n}$, $(b_i)_{i\leqq n}$ such that $x_{i-1} < x < x_i$, $x \in K \Rightarrow a_i < \theta(x) < a_i + \gamma$; $b_i < \psi(x) < b_i + \gamma$. Since $\varphi \leqq \theta$, we can assume that $b_i \leqq a_i$. Let $\gamma_i = \lambda([x_{i-1}, x_i]\backslash K)$. Hence $\sum_{i=1}^n \gamma_i \leqq \gamma$. The main point in the proof is to show that

$$\eta([x_{i-1}, x_i] \times ([0, 1]\backslash [b_i, b_i + \gamma] \cup [a_i, a_i + \gamma])) \leqq 3\gamma_i. \tag{21}$$

Indeed, this implies that

$$\eta(\{(x, t), t \notin [\psi(x) - \gamma, \psi(x) + \gamma] \text{ or } t \notin [\theta(x) - \gamma, \theta(x) + \gamma]\}) \leqq 5\gamma,$$

and letting $\gamma \to 0$ proves the theorem.

To prove (21), we can assume by Proposition 1 that $x_{i-1} = 0$, $i = 1$. We have

$$\begin{aligned} \eta([0, x_1] \times [a_1 + \gamma, 1]) &= G(x_1, a_1 + \gamma) \leqq H(x_1, a_1 + \gamma) \\ &= \lambda(\{x \leqq x_1;\ \theta(x) \geqq a_1 + \gamma\}) \leqq \gamma_1. \end{aligned} \tag{22}$$

It may or may not be the case that $b_1 + \gamma < a_1$. If $b_1 + \gamma < a_1$, we estimate $\eta([0, x_1] \times]b_1 + \gamma, a_1[)$. Let $\tau = \inf K$, so that $\lambda([0, \tau]) \leqq \gamma_1$. Since $\varphi(\tau, \theta(\tau)) = \psi(\tau) < b_1 + \gamma$, $\theta(\tau) > a_1$, we must have $\mu_\tau(]b_1 + \gamma, a_1[) = 0$, so that $\mu_{x_1}(]b_1 + \gamma, a_1[) = 0$. Hence

$$\eta(]\tau, x_1] \times]b_1 + \gamma, a_1[) = \mu_\tau(]b_1 + \gamma, a_1[) - \mu_{x_1}(]b_1 + \gamma, a_1[) = 0$$

and thus, by (11),

$$\begin{aligned} \eta([0, x_1] \times]b_1 + \gamma, a_1[) &\leqq \eta([0, \tau] \times [0, 1]) + \eta(]\tau, x_1] \times]b_1 + \gamma, a_1[) \\ &\leqq \lambda([0, \tau]) \leqq \gamma_1. \end{aligned} \tag{23}$$

We finally evaluate $\eta([0, x_1] \times [0, b_1[)$. By Lemma 5, whenever $c > b_1$, we have

$$\begin{aligned} \lambda(\{x;\ \theta(x) < c, \psi(x) > b_1\}) &= \nu(\{(x, t);\ t < c, \varphi(x, t) > b_1\}) \\ &\leqq \mu(]b_1, c[) \end{aligned}$$

and thus

$$\begin{aligned} \nu(\{(x, t);\ x \leqq x_1, t < c\}) &= \lambda(\{x;\ x \leqq x_1, \theta(x) < c\}) \\ &\leqq \lambda(\{x;\ \theta(x) < c, \psi(x) > b_1\}) + \lambda(\{x;\ x \leqq x_1, x \notin K\}) \\ &\leqq \mu(]b_1, c[) + \gamma_1, \end{aligned}$$

which means

$$H(x_1, 0) \leqq H(x_1, c) + \mu(]b_1, c[) + \gamma_1.$$

Since this holds for all $c > b_1$, we have

$$H(x_1, 0) \leqq G(x, b_1) + \gamma_1,$$

and hence

$$G(x_1, 0) \leqq H(x_1, 0) \leqq G(x, b_1) + \gamma_1$$

so that

$$\eta([0, x_1] \times [0, b_1[) \leqq \gamma_1. \tag{24}$$

Since (20) follows from (22) to (24), the proof is complete. □

We are now ready to construct the distribution of vacancies after the items have been attributed. Consider again a Borel-measurable map $\theta : [0, 1] \to [0, 1]$. Consider a

positive measure λ' on $[0, 1]$. The image ν' of λ' under the map $x \to (x, \theta(x))$ will be our distribution of vacancies. The reader would expect that in a correct model we would have $\nu([a, b] \times [0, 1]) = b - a$ for all $a \leqq b$. However, it is more convenient not to keep track of the vacancies of size zero, which are irrelevant for the packing. If λ_L denotes Lebesgue's measure, we will have $\lambda' \leqq \lambda_L$, and we can think of having unaccounted (irrelevant) vacancies of size zero of distribution $\lambda_L - \lambda'$.

Consider a parameter $\alpha > 0$ that will play a role similar to what it did in the definition of basic matching. We define $s(t, \alpha) = 1$ if $t < 2\alpha$, $s(t, \alpha) = \lfloor t/\alpha \rfloor$ if $t \geqq 2\alpha$. We consider the measure ν on $[0, 1]^2$ that has density $s(t, \alpha)$ at the point (x, t) with respect to ν'. The idea is that ν represents the distribution of the multiplicities of the vacancies. Consider the measure λ of density $s(\theta(x), \alpha)$ with respect to λ'. Then ν is the image of λ by the map $x \to (x, \theta(x))$.

We will assume that μ has a large enough mass at zero (larger than $\|\lambda\|$ is enough). Thus, all vacancies will receive items, and we will not have to consider separately those vacancies that receive no item. Of course, receiving a size zero item is the same as receiving nothing. We leave to the reader to check that the claim "all vacancies receive items" can be formalized by the equality

$$\eta([0, x] \times [0, 1]) = G(x, 0) = H(x, 0) = \nu([0, x] \times [0, 1]).$$

We will not use this fact directly. Actually, the fact that "all vacancies receive items" is mentioned at this point solely for the purpose of convincing the reader that the correct model is being developed. It will remain completely implicit in the rest of this section and will be used only in the proof of the (crucial) Theorem 3.

We assume that $[0, \alpha]$ supports μ. We set $A = \{x;\ \theta(x) \geqq 2\alpha\}$, $C = [0, 1] \backslash A$. We denote by λ'' the measure of $[0, 1]$ given by

$$\lambda''(B) = \eta((B \times [0, 1]) \cap (\text{Graph } \psi \backslash \text{Graph } \theta))$$

for each Borel set B of $[0, 1]$. We define the measure $\bar{\nu}_1$ on $[0, 1] \times [0, 1]$ as the image of λ' by the map $x \to \theta(x) - s(\theta(x), \alpha)\psi(x)$ *restricted to* A. The idea is that, in the basic matching, the vacancies of size $\theta(x) > \alpha$ receive $s(\theta(x), \alpha)$ items of the largest size left, i.e., $\psi(x)$. We define the measure $\bar{\nu}_2$ on $[0, 1] \times [0, 1]$ as the image of λ'' by the map $x \to \theta(x) - \psi(x)$ *restricted to* C. The idea is that (as Theorem 2 expresses) a vacancy of size $\theta(x)$ receives an item of size either $\theta(x)$ or $\psi(x)$. The measure λ'' takes in account the fact that only some vacancies of size $\theta(x)$ receive an item of size $\psi(x)$. The other bins of size $\theta(x)$ receive an item of size $\theta(x)$, so actually we create vacancies of size zero represented by the distribution $\lambda' - \lambda''$. Vacancies of size zero are irrelevant for our purposes, so we do not take this piece into account, and we set $\bar{\nu} = \bar{\nu}_1 + \bar{\nu}_2$. Given ν', μ, and α that satisfy the above conditions, the construction of $\bar{\nu}$, and of the related objects ν, η, μ_x, λ'', etc., will be called "performing the basic matching of μ and ν' with parameter α." When we need to be specific, we will write $\bar{\nu} = \bar{\nu}(\mu, \nu', \alpha)$.

We end this section with technical results that will be needed to stay in control when we iterate the constructions. We say that a measure ν on $[0, 1]^2$ is *decomposable* if it is the image of a measure λ on $[0, 1]$ by the map $x \to (x, \theta(x))$ that has the following property:

$\forall \varepsilon > 0$, there is a partition $L_0, L_1, \cdots, L_n$ (n depending on ε) of $[0, 1]$ in Borel sets such that

(25) $$L_0 = \{\theta < \varepsilon\} \quad \text{and} \quad \forall i \geqq 1,\ x, y \in L_i,\ x \leqq y \Rightarrow \theta(x) \leqq \theta(y).$$

If the map θ satisfies (25), we will call it decomposable.

PROPOSITION 2. *If ν is decomposable, then $\bar{\nu}$ is decomposable.*

Proof. Consider $\varepsilon > 0$ and a Borel set L such that

$$x, y \in L \Rightarrow \varepsilon \leqq \theta(x) \leqq \theta(y).$$

We can partition L into a finite number of (Borel) pieces $(L_j)_{j \leqq p}$ such that

$$x, y \in L_j \Rightarrow |\theta(y) - \theta(x)| < \varepsilon.$$

If $x, y \in L_j$, $x \leqq y$, $\theta(x) - \psi(x) \geqq \varepsilon$, $\theta(y) - \psi(y) \geqq \varepsilon$, we have $\psi(y) < \theta(x)$, and so, by Lemma 4, we have $\psi(y) \leqq \psi(x)$, so that $\theta(x) - \psi(x) \leqq \theta(y) - \psi(y)$. The same holds with $\theta - s\psi$. This completes the proof.

Let us say that a decomposable measure ν has the property $P(y)$ if the following holds:

$P(y)$: for $x < y$, we have $\theta(x) < 1$; for $x > y$, we have $\theta(x) = 1$; and the restriction of λ' to $[y, 1]$ is Lebesgue's measure.

PROPOSITION 3. *If ν has property $P(y)$, then $\bar{\nu}$ has property $P(\bar{y})$, where $y \leqq \bar{y} \leqq y + \|\mu\|/\lfloor 1/\alpha \rfloor$.*

Proof. The decomposable map $\bar{\theta}$ corresponding to $\bar{\nu}$ obviously satisfies $\bar{\theta}(x) < 1$ for $x < y$. Thus, by Proposition 1, we can replace μ by μ_y and suppose that ν has property $P(0)$, i.e., ν is the one-dimensional measure on $[0, 1] \times \{1\}$. But the result is obvious in that case. □

We observe that when ν has property $P(y)$, we have $1 - y = \nu([0, 1] \times \{1\})$, i.e., $y = 1 - \nu([0, 1] \times \{1\})$.

When performing the basic matching of μ and ν' with parameter α, we will say that μ *is packed at* x_0 if $\mu_{x_0} = 0$. (In that definition, we do not insist on x_0 being the smallest possible.)

3. FFD matching. We mentioned earlier that basic matching would be an adequate representation of "small portion" of FFD. Our task in this section is to develop that idea. Consider a distribution ν' on $[0, 1] \times [0, 1]$ that is the image of a positive measure $\lambda' \leqq \lambda_L$ by the map $x \to (x, \theta(x))$, where $\theta(x)$ is a decomposable map. Consider a positive measure μ on $[0, 1]$. Let $\alpha = \inf\{a; \mu(]a, 1])\} = 0$. We will, in that section, assume that μ is supported by the set $\{0\} \cup [\beta, \alpha]$, where $\beta \geqq \alpha/2$. We allow μ to have a (large) mass at zero for the reasons mentioned in § 2. The use of this will soon become apparent. Define, as before, $s(t, \alpha) = \lfloor t/\alpha \rfloor$ for $t \geqq 2\alpha$, $s(t, \alpha) = 1$ otherwise. We perform the basic matching of μ and ν' with parameter α, and we use the notations of § 2. Our whole approach relies on the following theorem.

THEOREM 3. *Suppose that for some $0 \leqq x_1 \leqq 1$ the following occur:*

$$\forall s \geqq 2, \qquad \nu'([0, x_1] \times]s\beta, s\alpha[) = 0. \tag{26}$$

$$\mu \text{ is packed at } x_1. \tag{27}$$

$$\mu \text{ is supported by } \{0\} \cup]\beta, \alpha]. \tag{28}$$

Then, given any $\varepsilon > 0$, $0 < t_0 < \beta$, we can find $\delta > 0$ and $N_0 > 0$ with the following properties. Consider $N \geqq N_0$ and a list of items $z_1, \cdots, z_p$ that satisfies

$$\forall t > 0, \qquad \left| \frac{1}{N} \operatorname{card}\{i \leqq p;\ z_i \geqq t\} - \mu([t, 1]) \right| \leqq \delta. \tag{29}$$

Consider a list of vacancies $(v_i)_{i \leqq N}$ that satisfies

$$\forall x \leqq 1, \quad \forall t \geqq t_0, \qquad \left| \frac{1}{N} \operatorname{card}\{i \leqq xN, v_i \geqq t\} - \nu'([0, x] \times [t, 1]) \right| \leqq \delta. \tag{30}$$

Then, if $(w_i)_{i \leqq N}$ *is the list of vacancies after the items* z_i *are attributed to the vacancies* v_i *following the* FFD *rule, we have*

$$(31) \qquad \forall x \leqq 1, \qquad \forall t \geqq t_0, \qquad \left| \frac{1}{N} \operatorname{card}\{i \leqq xN;\ w_i \geqq t\} - \bar{\nu}([0, x] \times [t, 1]) \right| \leqq \varepsilon.$$

Proof. Step 1. At most δN items z_i are greater than α. Packing them will modify at most δN vacancies. If we delete these items from our list of items, (29) still holds with 2δ instead of δ. Thus we can assume that all items are less than or equal to α. In a similar fashion, we can assume that all items are greater than β.

Step 2. We show that it is enough to prove (31) for $x \leqq x_1$ and to prove the following:

(32) The number of items z_i that are not packed in bins of index less than Nx_1 is less than or equal to εN.

Since ν' and $\bar{\nu}$ coincide on $[x_1, 1]$, from (30) we get that for $t \geqq 0$, $x \geqq x_1$,

$$\left| \frac{1}{N} \operatorname{card}\{i;\ x_1 N < i \leqq xN,\ v_i \geqq t\} - \bar{\nu}(]x_1, x] \times [t, 1]) \right| \leqq 2\delta.$$

On the other hand, since at most εN items z_i have not been packed in bins of index less than Nx_1, for at most εN indices $i > x_1 N$, we can have $w_i \neq v_i$. Thus,

$$\left| \frac{1}{N} \operatorname{card}\{i;\ x_1 N < i \leqq xN,\ w_i \geqq t\} - \bar{\nu}(]x_1, x] \times [t, 1]) \right| \leqq 2\delta + \varepsilon.$$

Together with (31) for $x = x_1$, this implies

$$\left| \frac{1}{N} \operatorname{card}\{i;\ i \leqq xN,\ w_i \geqq t\} - \bar{\nu}([0, x] \times [t, 1]) \right| \leqq 2\delta + 2\varepsilon$$

for $x_1 \leqq x \leqq 1$ and give (31) for all $0 \leqq x \leqq 1$ (with $2\varepsilon + 2\delta$ instead of ε).

In the sequel, we assume $x_1 = 1$, which changes nothing but simplifies notations. Thus (26) becomes now

$$(33) \qquad \forall s \geqq 2, \qquad \nu'([0, 1] \times]s\beta, s\alpha[) = 0.$$

Step 3. We show now that we can assume that the items z_i are attributed to the vacancies v_i according to basic matching of parameter α instead of the FFD rule. Say that a vacancy v_i is *abnormal* (respectively, *normal*) if it belongs to some (does not belong to any) interval $]s\beta, s\alpha[$, $s \geqq 2$. From (33), $\nu'([0, 1] \times]s\beta, s\alpha[) = 0$ for $s \geqq 2$. Since it is enough to consider the case $s \leqq 1/\beta$, by (30) we have at most $2\delta N/\beta$ abnormal vacancies. Thus, if we modify our list of vacancies by replacing the abnormal vacancies by 0, (30) still holds with $\delta(1 + 2/\beta)$ instead of δ. Any vacancy can receive at most $1/\beta$ items, thus the abnormal vacancies can receive at most $2\delta N/\beta^2$ items. Hence, the list of items attributed to the normal vacancies satisfies (29) with $\delta(1 + 2/\beta^2)$ instead of δ. If we attribute these items to the modified list of vacancies following the FFD rule, the ith vacancy w_i' obtained is w_i whenever v_i is normal. Thus if the sequence (w_i') satisfies (31), the sequence w_i will satisfy it with $\varepsilon + 2\delta/\beta^2$ instead of ε. Thus, we see that it is enough to prove the result under the extra assumption that there are no abnormal vacancies. However, in that case the basic matching with parameter α and the attribution of items following the FFD rule coincide, as is seen, e.g., by induction on the number of items, the point being that a vacancy $v_i \geqq \alpha$ is attributed $s = \lfloor v_i/\alpha \rfloor$ consecutive largest items by the FFD rule, since it satisfies $v_i < (s+1)\alpha$, and thus $v_i \leqq (s+1)\beta$ and hence cannot accept $s+1$ items of size greater than β.

Step 4. It is convenient to use the same conventions for our list of vacancies v_i as we do for ν'. Recall that we allow a "defective" ν', i.e., $\nu'([a,b]\times[0,1]) \leqq b-a$. The corresponding idea for the list of vacancies v_i is to allow some indices i to have no vacancy v_i attached to them, or if one prefers, one can achieve the same effect by formally setting this vacancy equal to -1. For the purpose of counting the vacancies w_i, it is very convenient to assume that all the v_i receive items. This is done by adding enough items of size zero to the list (z_i). It is at this point that our hypothesis that μ has a large enough mass at zero is explicitly used, since we can add the size zero items and assume that (29), which is valid for $t>0$, also becomes valid for $t=0$ (provided that $N^{-1} \leqq \delta$).

Consider the restriction ν_1' of ν' to $[0,1]\times[0,\beta[$. Then

$$\forall t \geqq t_0, \qquad \left| \frac{1}{N} \, card\{i;\, i \leqq xN,\, t \leqq v_i < \beta\} - \nu_1'([0,x]\times[t,1]) \right| \leqq 2\delta.$$

By construction of $\bar{\nu}$, if we replace ν' by $\nu' - \nu_1'$ (e.g., by replacing λ' by its restriction to the set $\{\theta \geqq \beta\}$), we replace $\bar{\nu}$ by $\bar{\nu} - \nu_1'$. On the other hand, the vacancies $v_i < \beta$ are not changed by the packing. Thus, to prove the theorem, one can replace ν' by $\nu' - \nu_1'$ and remove the vacancies $v_i < \beta$ from the list (i.e., setting them equal to -1). Hence, we can assume that (30) holds for all $t \geqq 0$.

Step 5. We have

$$\nu([0,x]\times[t,1]) = s\nu'([0,x]\times[t,1]) + \sum_{i>s} \nu'([0,x]\times[i\alpha,1]),$$

where $s = s(t,\alpha)$, and a similar formula holds for the multiplicities m_i of the vacancies. From (30), (which now holds for all $t \geqq 0$), it follows that

$$\forall t \geqq 0, \qquad \left| \frac{1}{N} \sum \{m_i;\, i \leqq xN,\, v_i \geqq t\} - \nu([0,x]\times[t,1]) \right| \leqq \frac{\delta}{\alpha}. \tag{34}$$

To simplify notations, for a subset X of $[0,1]^2$, we set

$$V'(X) = \frac{1}{N} \, card\left\{ i \leqq N;\, \left(\frac{i}{N}, v_i\right) \in X \right\},$$

$$V(X) = \frac{1}{N} \sum \left\{ m_i;\, \left(\frac{i}{N}, v_i\right) \in X \right\},$$

$$Z(X) = \frac{1}{N} \, card\{\text{items attributed to a vacancy } v_i,\, (i/N, v_i) \in X\},$$

$$\bar{V}(X) = \frac{1}{N} \, card\left\{ i \leqq N;\, \left(\frac{i}{N}, w_i\right) \in X \right\}.$$

Thus, (30) and (34) become

$$\forall t \geqq 0, \qquad |V'([0,x]\times[t,1]) - \nu'([0,x]\times[t,1])| \leqq \delta$$

$$\forall t \geqq 0, \qquad |V([0,x]\times[t,1]) - \nu([0,x]\times[t,1])| \leqq \delta/\alpha.$$

From (29), (2), and the definition of G, we see that for $t \geqq 0$,

$$|Z([0,x]\times[t,1]) - \eta([0,x]\times[t,1])| \leqq \delta(1+(1/\alpha)).$$

It follows that if R is a rectangle $I\times J$, where I, J are intervals (that may contain their endpoints or not), we have

$$|Z(R) - \eta(R)| \leqq 4\delta(1+(1/\alpha)).$$

A similar statement holds for V' and ν' and V and ν. For simplicity, we set $\delta' = 4\delta(1+1/\alpha)$. If X is the union of k disjoint rectangles, then $|Z(X)-\eta(X)| \leqq k\delta'$.

Step 6. We are already in position to prove (32). Since μ is packed at $x_1 = 1$, we have $\mu_1([\beta, 1]) = 0$, i.e., $\eta([0,1]\times[\beta,1]) = \mu([\beta,1])$. Thus,

$$Z([0,1]\times[\beta,1]) \geqq \eta([0,1]\times[\beta,1]) - \delta' = \mu([\beta,1]) - \delta'$$

$$\geqq \frac{1}{N}\, card\{i;\, z_i \geqq \beta\} - \delta - \delta'$$

and thus at most $N(\delta+\delta')$ nonzero items are not attributed to the vacancies v_i. Since we can always assume $\delta+\delta' \leqq \varepsilon$, this proves (32).

Step 7. Since θ is decomposable, we can find a finite Borel partition $M_0, \cdots, M_n$ of $[0,1]$, where $M_0 = \{\theta < \beta\}$, such that

$$\forall i \geqq 1, \qquad x, y \in M_i,\ x < y \Rightarrow \theta(x) \leqq \theta(y). \tag{35}$$

Let $\bar{\gamma}$ be a positive number to be determined later. We can approximate the sets $M_i \cap \{\theta - \psi \geqq t_0\}$, $M_i \cap \{\theta - \psi < t_0\}$ by finite unions of intervals. That is, we can find a partition $(I_j)_{j \leqq m}$ of $[0,1]$ in intervals, and for each $j \leqq m$ there is a measurable subset L_j of I_j such that for some $i = i(j)$, $0 \leqq i \leqq n$, we have

$$L_j = I_j \cap M_i \cap \{\theta - \psi \geqq t_0\} \quad \text{or} \quad L_j = I_j \cap M_i \cap \{\theta - \psi < t_0\}$$

and such that if we set

$$\gamma_j = \lambda'(I_j \backslash L_j),$$

we have $\sum_{j \leqq m} \gamma_j \leqq \bar{\gamma}$ (see [11, Prop. 15, p. 63). When $L_j \subset M_i$ for $i \geqq 1$, we can moreover assume from (35) that

$$\sup_{L_j} \theta(x) - \inf_{L_j} \theta(x) < t_0/2, \tag{36}$$

by cutting I_j in at most $2/t_0$ subintervals. When $L_j \subset \{\theta - \psi \geqq t_0\}$, it follows from Lemma 4 (as in the proof of Proposition 3) that

$$x, y \in L_j,\ x \leqq y \Rightarrow \psi(y) \leqq \psi(x). \tag{37}$$

Thus, we can also assume

$$\sup_{L_j} \psi(x) - \inf_{L_j} \psi(x) < t_0/2, \tag{38}$$

but splitting again I_j in at most $2/t_0$ subintervals. A further splitting ensures that on each L_j either $\beta \leqq \theta \leqq 2\alpha$ or $s_j\alpha \leqq \theta < (s_j+1)\alpha$ for some integer $s_j \geqq 2$.

Step 8. Let $t \geqq t_0$ be fixed. We can split each of our intervals I_j in at most two pieces in such a way that for all j, one of the following occurs:

$$L_j \subset M_0, \text{ i.e., } x \in L_j \Rightarrow \theta(x) < \beta; \tag{39}$$

$$x \in L_j \Rightarrow \beta \leqq \theta(x) \leqq 2\alpha,\ \theta(x) - \psi(x) < t; \tag{40}$$

$$x \in L_j \Rightarrow \beta \leqq \theta(x) \leqq 2\alpha,\ \theta(x) - \psi(x) \geqq t; \tag{41}$$

$$x \in L_j \Rightarrow s_j\alpha \leqq \theta(x) < (s_j+1)\alpha,\ \theta(x) - s_j\psi(x) < t; \tag{42}$$

$$x \in L_j \Rightarrow s_j\alpha \leqq \theta(x) < (s_j+1)\alpha,\ \theta(x) - s_j\psi(x) \geqq t. \tag{43}$$

Our family of intervals $(I_j)_{j \leqq m}$ now depends on t, but its cardinality m is bounded independently of t. Also if $\gamma_j = \lambda'(I_j \backslash L_j)$, we have $\sum_{j \leqq m} \gamma_j \leqq \bar{\gamma}$, independently of t.

Step 9. In all the preceding five cases, we are going to show

$$|\bar{V}(I_j\times[t,1])-\bar{\nu}(I_j\times[t,1])|\leqq\frac{10}{\alpha}(\delta+\gamma). \tag{44}$$

Since we can assume that $\bar{\nu}(I_j\times[0,1])\leqq\bar{\gamma}$ for all j, we get by summation over $j\leqq m$ that for all $x\leqq 1$,

$$\left|\frac{1}{N}\,card\{i;\, xN\leqq i,\, w_j\geqq t\}-\bar{\nu}([0,x]\times[t,1])\right|\leqq\frac{10}{\alpha}\,m\delta+\frac{11}{\alpha}\,\bar{\gamma}.$$

We choose $\bar{\gamma}=\alpha\varepsilon/22$ (which determines m) and then $\delta=\alpha\varepsilon/22m$ to obtain (31) (in which N_0 is any integer $\geqq 1/\delta$).

We turn to the proof of (44). We fix $j\leqq m$. We denote J and J' respectively, the smallest intervals such that $x\in L_j\Rightarrow\theta(x)\in J$, $\psi(x)\in J'$. We observe that by (36) J is of length less than $t_0/2$ unless (39) occurs and that J' is of length less than $t_0/2$ if (41) occurs. For simplicity of notation, we set $I=I_j$, $L=L_j$, $\gamma=\gamma_j$. The most interesting of the following cases will be the third one.

Case 1. (39) holds. In Step 4 we have reduced the proof to the case where $\lambda'(M_0)=0$. Thus,

$$\bar{\nu}(I\times[0,1])\leqq\lambda'(I)=\lambda'(I\backslash M_0)\leqq\lambda'(I\backslash L)\leqq\gamma.$$

Also

$$\bar{V}'(I\times[0,1])\leqq V'(I\times[0,1])\leqq\nu'(I\times[0,1])+\delta'\leqq\gamma+\delta'$$

so that (44) holds.

Case 2. (40) holds. We set $J-J'=\{u-u';\, u\in J,\, u'\in J'\}$. We first show that $J-J'\subset[0,t[$. Since $\theta(x)-\psi(x)<t$ on L, and since θ increases and ψ decreases on L, we have $\sup_L\theta-\inf_L\psi\leqq t$. If $\sup_L\theta-\inf_L\psi=t$, since $\theta(x)-\psi(x)<t$ for $t\in L$, either $\sup_L\theta$ or $\inf_L\psi$ is not attained and thus does not belong to J (respectively, J'). Thus $J-J'\subset[0,t[$. Since $\theta(x)-\psi(x)<t$ for $x\in L$, we have

$$\bar{\nu}(I\times[t,1])\leqq\lambda'(I\backslash L)\leqq\gamma.$$

Since $\theta(x)\in J$ for $x\in L$, we have

$$\nu'(I\times([0,1]\backslash J))\leqq\lambda'(I\backslash L)\leqq\gamma,$$

and hence, since $I\times([0,1]\backslash J)$ is the union of at most two disjoint rectangles,

$$V'(I\times([0,1]\backslash J))\leqq\gamma+2\delta'. \tag{45}$$

Since η is supported by the union of the graphs of θ and ψ, we have

$$\eta(I\times([0,1]\backslash(J\cup J')))\leqq\lambda(I\backslash L)\leqq\frac{\gamma}{\alpha},$$

and thus, since $I\times([0,1]\backslash(J\cup J'))$ is the union of at most three disjoint rectangles,

$$Z(I\times([0,1]\backslash(J\cup J')))\leqq 3\delta'+\frac{\gamma}{\alpha}. \tag{46}$$

Attributing an item of size in $J\cup J'$ to a vacancy in J creates a vacancy in $J-(J\cup J')$. Since J has length less than $t_0/2$, we have $J-J'\subset[0,t_0[\subset[0,t[$, thus $J-(J\cup J')\subset[0,t[$. Thus

$$\bar{V}(I\times[t,1])\leqq 5\delta'+\frac{2\gamma}{\alpha}$$

and (44) holds again.

Case 3. (41) holds. In that case, both J and J' have length less than $t_0/2$. Thus they must be disjoint, for otherwise $x \in L \Rightarrow \theta(x) - \psi(x) < t_0$. Since $\theta(x) - \psi(x) \geqq t$ on L, we have $\inf_L \theta - \sup_L \psi \geqq t$, so that $J - J' \subset [t, 1]$.

Since η is supported by the union of the graphs of θ and ψ, and since for $x \in L$, we have $\theta(x) \notin J'$, $\psi(x) \in J'$, we have

$$\bar{\nu}(L \times [t, 1]) = \lambda''(L) = \eta(L \times [0, 1] \cap \mathit{graph}\ \psi) = \eta(L \times J').$$

We also have

$$\eta(L \times J') \geqq \eta(I \times J') - \eta((I \backslash L) \times [0, 1]) \geqq \eta(I \times J') - \gamma,$$

and

$$\begin{aligned} \bar{\nu}(I \times [t, 1]) &\leqq \bar{\nu}((I \backslash L) \times [0, 1]) + \bar{\nu}(L \times [0, 1]) \\ &\leqq \lambda'(I \backslash L) + \lambda''(L) \leqq \gamma + \eta(I \times J') \end{aligned}$$

so that

$$|\bar{\nu}(I \times [t, 1]) - \eta(I \times J')| \leqq \gamma. \tag{47}$$

We note that (45) and (46) still hold. An item of size in J' that is attributed to a vacancy in J creates a vacancy greater than or equal to t. Then, from (45),

$$\bar{V}(I \times [t, 1]) \geqq Z(I \times J') - \gamma - \delta'.$$

On the other hand, an item of size in J that is attributed to a vacancy in J creates a vacancy less than t_0. Thus vacancies greater than or equal to t can come only from item sizes not in J or vacancies not in J and hence

$$\begin{aligned} \bar{V}(I \times [t, 1]) &\leqq Z(I \times ([0, 1] \backslash J)) + V'(I \times ([0, 1] \backslash J)) \\ &\leqq Z(I \times J') + Z(I \times ([0, 1] \backslash (J \cup J')) + V'(I \times ([0, 1] \backslash J)) \\ &\leqq Z(I \times J') + 5\delta' + \frac{2\gamma}{\alpha}, \end{aligned}$$

and thus

$$|\bar{V}(I \times [t, 1]) - Z(I \times J')| \leqq 5\delta' + \frac{2\gamma}{\alpha}.$$

Since $|\eta(I \times J') - Z(I \times J')| \leqq \delta'$, (47) shows that (44) holds again.

Case 4. (42) holds. We set $s = s_j$, $J - sJ' = \{u - su';\ u \in J,\ u' \in J'\}$. As in Case 2, we see that $J - sJ' \subset [0, t[$, and thus, by definition of $\bar{\nu}$,

$$\bar{\nu}(I \times [t, 1]) \leqq \lambda'(I \backslash L) \leqq \gamma.$$

Since $\theta(x) \in J$ for $x \in L$, we have

$$\nu'(I \times ([0, 1] \backslash J)) \leqq \lambda'(J \backslash L) \leqq \gamma$$

and hence

$$V'(I \times ([0, 1] \backslash J)) \leqq \gamma + 2\delta'. \tag{48}$$

Since η is supported by the union of the graphs of θ and ψ, and also by $[0, \alpha]\times[0, 1]$, and since $\theta \geqq 2\alpha$ on L, on $L\times[0, 1]$, η is supported by the graph of ψ, hence by $L\times J'$ and thus

$$\eta(I\times([0, 1]\backslash J')) \leqq \lambda(I\backslash L) \leqq \frac{\gamma}{\alpha}.$$

So we have

$$Z(I\times([0, 1]\backslash J')) \leqq 2\delta' + \frac{\gamma}{\alpha}. \tag{49}$$

To vacancies in J the basic matching attributes s consecutive largest unpacked items. If all these items belong to J', one checks easily that the resulting vacancy belongs to $J - sJ' \subset [0, t[$. Thus

$$\bar{V}(I\times[t, 1[) \leqq 4\delta' + \frac{2\gamma}{\alpha}$$

and (44) holds again.

Case 5. (43) holds. Using the notation of Case 5, we see as in Case 3 that $J - sJ' \subset [t, 1]$. By definition of $\bar{\nu}$,

$$\lambda'(I) - \gamma \leqq \lambda'(L) \leqq \bar{\nu}(I\times[t, 1]) \leqq \lambda'(I).$$

We observe that (48) and (49) still hold. Also we have

$$\nu'(I\times J) \geqq \lambda'(L) \geqq \lambda'(I) - \gamma,$$

so that

$$V'(I\times J) \geqq \lambda'(I) - \gamma - 2\delta'.$$

To vacancies in J the basic matching attributes s consecutive largest unpacked items. If all these items belong to J', the resulting vacancy belongs to $J - sJ' \subset [t, 1]$. Thus,

$$\bar{V}(I\times[t, 1[) \geqq V'(I\times J) - \left(2\delta' + \frac{\gamma}{\alpha}\right) \geqq \lambda'(I) - 4\delta' - \frac{2\gamma}{\alpha}$$

$$\geqq \bar{\nu}(I\times[t, 1]) - 4\delta' - \frac{2\gamma}{\alpha}.$$

Also

$$\bar{V}(I\times[t, 1[) \leqq V'(I\times[0, 1]) \leqq \nu'(I\times[0, 1]) + \delta'$$

$$= \lambda'(I) + \delta' \leqq \bar{\nu}(I\times[t, 1]) + \delta' + \gamma.$$

Thus, (44) holds again and the proof is complete.

The proof of Theorem 3 needs only obvious modifications to give Theorem 4.

THEOREM 4. *Same as Theorem* 3, *with* (26) *and* (27) *replaced, respectively, by*

$$\forall s \geqq 2, \qquad \nu'([0, 1]\times[s\beta, s\alpha[) = 0, \tag{50}$$

$$\mu \text{ is supported by } \{0\}\cup[\beta, \alpha]. \tag{51}$$

4. Definition of $f(\mu)$ and proof of Theorem 1. Our last significant task will be to prove Theorem 5.

THEOREM 5. *Given a probability measure μ on $[0,1]$, one can find four sequences $(u_k)_{k\geqq 1}$, $(u'_k)_{k\geqq 1}$, $(x_k)_{k\geqq 1}$, $(y_k)_{k\geqq 1}$ of numbers, a sequence $(\nu'_k)_{k\geqq 0}$ of decomposable measures on $[0,1]^2$, and a sequence $(\mu_k)_{k\geqq 1}$ of positive measures on $[0,1]$, with the following properties*:

(52) ν'_0 *is the one-dimensional Lebesgue measure on* $[0,1]\times\{1\}$,

(53) ν'_k *is the measure* $\bar{\nu}(\mu_k, \nu'_{k-1}, u_k)$, *and* μ_k *is packed at* x_k.

(54) ν'_k *has the property* $P(y_k)$ *for* $k\geqq 1$.

$$\mu = \sum_{k\geqq 1} \mu_k \tag{55}$$

$$\lim_{k\to\infty} u_k = 0;\ u_{k+1} \leqq u'_k \leqq u_k \tag{56}$$

(57) *For each* $k\geqq 1$, *either of the following occurs*:

(58) $]u_{k+1}, u'_k]$ *supports* μ_k, *and* $\nu'_k([0, x_k]\times]su_{k+1}, su'_k[) = 0$ *whenever* $s\geqq 2$.

(59) $[u_{k+1}, u'_k]$ *supports* μ_k, *and* $\nu'_k([0, x_k]\times[su_{k+1}, su'_k[) = 0$ *whenever* $s\geqq 2$.

Before we prove this theorem, we turn to the definition of $f(\mu)$ and the proof of Theorem 1. From Proposition 3, we see that the sequence (y_k) increases and that $y_{k+1}\leqq y_k + \|\mu_{k+1}\|/\lfloor 1/u_{k+1}\rfloor$. Thus, for $l\geqq 1$,

$$y_{k+l} \leqq y_k + \sum_{0\leqq i\leqq l} \frac{\|\mu_{k+i}\|}{\lfloor 1/u_{k+i}\rfloor} \leqq y_k + \sum_{1\leqq i\leqq l} \frac{\|\mu_{k+i}\|}{\lfloor 1/u_{k+1}\rfloor}.$$

We set $f(\mu) = \lim_{k\to\infty} y_k$. Thus, for all k,

$$f(\mu) \leqq y_k + \left(\left\lfloor \frac{1}{u_{k+1}}\right\rfloor\right)^{-1}. \tag{60}$$

We should observe that the quantities involved in Theorem 5 have no reason to be uniquely determined. It is a priori unclear that $f(\mu) = \lim_{k\to\infty} y_k$ depends on μ only, and not on these quantities. But we are going to show that Theorem 1 holds for this value of $f(\mu)$, thereby proving that it depends on μ only. Consider a list of items $z_1\geqq\cdots\geqq z_N$, and let

$$\tau = \sup_{0\leqq t\leqq 1} \left| \frac{1}{N} \operatorname{card}\{i\leqq N;\ z_i\geqq t\} - \mu([t,1])\right|. \tag{61}$$

We define $i_k = \lfloor N\|\sum_{i\leqq k}\mu_i\|\rfloor$. Thus, it is clear that

$$\sup_{0\leqq t\leqq 1} \left| \frac{1}{N} \operatorname{card}\{i;\ i_k\leqq i < i_{k-1},\ z_i\geqq t\} - \mu_k([t,1])\right| \leqq 2\tau + \frac{2}{N}. \tag{62}$$

Consider now N empty unit-size bins, and denote by $v_1^k, \cdots, v_N^k$ the list of vacancies after the items $(z_i)_{i\leqq i_k}$ have been packed according to the FFD rule.

Claim. Given $k\geqq 1$, $\varepsilon > 0$, $t_0 > 0$, there exists N_0 and $\delta > 0$, such that if (61) holds for $\tau\leqq\delta$, and if $N\geqq N_0$, then for all $x\leqq 1$ and $t\geqq t_0$, we have

$$\left| \frac{1}{N} \operatorname{card}\{i;\ i\leqq Nx,\ v_i^k\geqq t\} - \nu'_k([0,x]\times[t,1])\right| \leqq \varepsilon. \tag{63}$$

This statement is proved by induction over k, using (62) and using in the induction step, Theorem 3 when (58) holds and Theorem 4 when (59) holds.

Denote by F_k the number of bins that have received items after z_{i_k} is packed. Thus,

$$F_k = N - card\{i \leqq N;\ v_i^k = 1\}.$$

From (63), with $x = t = 1$, we have

$$\left| \frac{1}{N} card\{i \leqq N;\ v_i^k = 1\} - \nu_k'([0,1]\times\{1\}) \right| \leqq \varepsilon$$

so that we have $|N^{-1}F_k - y_k| \leqq \varepsilon$, since $y_k = 1 - \nu_k'([0,1]\times\{1\})$. Denote by F the total number of bins needed to pack $z_1, \cdots, z_N$. Then $F_k \leqq F \leqq F_k + 1 + N(\lfloor 1/u_{k+1} \rfloor)^{-1}$, since new bins accept at least $\lfloor 1/u_{k+1} \rfloor$ items of size $\leqq u_{k+1}$. Thus, with (60) we have

$$|N^{-1}F - f(\mu)| \leqq \varepsilon + N^{-1} + 2(\lfloor 1/u_{k+1} \rfloor)^{-1}.$$

Since $\lim_{k\to\infty} u_{k+1} = 0$, given $\varepsilon' > 0$, we can first pick k such that $2(\lfloor 1/u_{k+1} \rfloor)^{-1} \leqq \varepsilon'/3$; we then pick δ such that (63) holds for $\varepsilon' = \varepsilon/3$, then, if needed, N large enough that $1/N \leqq \varepsilon'/3$. This completes the proof of Theorem 1.

To prove Theorem 5, it suffices to apply inductively the following to the restriction of μ to the interval $]2^{-l-1}, 2^{-l}]$, $l \geqq 0$.

THEOREM 6. *Given a positive measure μ on $[0,1]$, supported by $]2^{-l-1}, 2^{-l}]$ $(l \geqq 0)$, and a decomposable measure ν' on $[0,1]^2$ that has property $P(y)$, we can find a finite sequence $2^{-l} = u_1 \geqq u_1' \geqq u_2 \geqq u_2' \geqq \cdots \geqq u_{p+1} = 2^{-l-1}$, increasing sequences $(y_k)_{k\leqq p}$, $(x_k)_{k\leqq p}$ of numbers, a sequence $(\nu_k')_{0\leqq k\leqq p}$ of decomposable measures on $[0,1]^2$, and a sequence $(\mu_k)_{k\leqq p}$ of positive measures on $[0,1]$ with the following properties*:

$$\nu_0' = \nu', \tag{64}$$

$$\nu_k' \text{ is the measure } \bar{v}(\mu_k, \nu_{k-1}', u_k'), \text{ and } \mu_k \text{ is packed before } x_k. \tag{65}$$

$$\nu_k' \text{ has property } P(y_k), \tag{66}$$

$$\mu = \sum_{1\leqq k\leqq p} \mu_k. \tag{67}$$

$$\text{For each } 1 \leqq k \leqq p, \text{ either of the following occurs:} \tag{68}$$

$$]u_{k+1}, u_k'] \text{ supports } \mu_k \text{ and } \nu_k'([0, x_k]\times]su_{k+1}, su_k'[) = 0 \tag{69}$$

whenever $s \geqq 2$, or

$$[u_{k+1}, u_k'] \text{ supports } \mu_k \text{ and } \nu_k'([0, x_k]\times[su_{k+1}, su_k'[) = 0 \tag{70}$$

whenever $s \geqq 2$.

Proof. Since ν' is decomposable, we can find a finite Borel partition $L_1, \cdots, L_n$ of the set $\{\theta \geqq 2^{-l-1}\}$, such that for $i \leqq n$,

$$x, y \in L_i \Rightarrow \theta(x) \leqq \theta(y).$$

The procedure we will define will take at most $n2^{l+1}$ steps. We define, for $i \leqq n$,

$$\theta_i(x) = \inf(\{\theta(y);\ y \geqq x,\ y \in L_i\}).$$

Thus θ_i is increasing, and $\theta_i(x) = \theta(x)$ if $x \in L_i$.

We define $u_1' = \inf\{a > 0;\ \mu(]a, 1]) = 0\}$. We use u_1' as a parameter for the basic matching, and we consider the measure ν of density $s(t, u_1')$ with respect to ν'. We perform the basic matching of μ and ν. Define

$$\Gamma_1(x) = \inf\{a > 0;\ \mu_x(]a, 0]) = 0\}.$$

This is a decreasing function. It is right continuous, as follows from the relation (see (8)) for $x \leqq y$.

$$\mu_x(]a, 0]) - \mu_y(]a, 0]) \leqq \eta([x, y] \times [0, 1]) \leqq \lambda([x, y]) \leqq 2^{l+1}(y - x).$$

For $i \leqq n$, we define $s_i^! = \inf(\{s(\theta_i(y), u_1'), y > 0\})$. Since θ is increasing, $\theta_i(y) < (s_i^! + 1)u_1'$ for y small enough. Since Γ_1 is right continuous, we have $\lim_{y \to 0^+} \Gamma_1(y) = u_1'$, so we can find $y > 0$ such that $\theta_i(y) < (s_i^! + 1)\Gamma_1(y)$ for $i \leqq n$. Define

$$x_1 = \sup\{y > 0; \forall i \leqq n, \theta_i(y) < (s_i^! + 1)\Gamma_1(y)\}$$

so that $x_1 > 0$. We define $u_2 = \lim_{x \to x_1^-} \Gamma_1(x)$.

Case 1. $\Gamma_1(x) > u_2$ for $x < x_1$. Let $x < x_1$. For $x < x' < x_1$, we have $\theta_i(x) \leqq \theta_i(x') \leqq (s_i^! + 1)\Gamma_1(x')$. Letting $x' \to x_1$ gives $\theta_i(x) \leqq (s_i^! + 1)u_2$. Since $s_i^! u_1' \leqq \theta_i(x) \leqq (s_i^! + 1)u_2$ for $0 < x < x_1$, we have $\nu'([0, x_1] \times]su_2, su_1'[) = 0$ for all $s \geqq 2$. We define μ_1 as the restriction of μ to $]u_2, u_1]$. It is supported by $]u_2, u_1']$, since $(\mu_1)_{x_1} \leqq \mu_{x_1}$, $(\mu_1)_{x_1}$ is supported by $[0, u_2]$. Since $(\mu_1)_{x_1} \leqq \mu_1$, it is supported by $]u_2, u_1']$, so we have $(\mu_1)_{x_1} = 0$. This shows that μ_1 is packed before x_1 and that (69) holds.

Case 2. For some $x_0 < x_1$, we have $\Gamma_1(x_0) = u_2$. For $x < x_1$, we have

$$\theta_i(x) \leqq \theta_i(\max(x, x_0)) < (s_i^! + 1)\Gamma_1(\max(x, x_0)) = (s_i^! + 1)u_2.$$

Thus, $\nu'([0, x_1] \times [su_2, su_1']) = 0$ for $s \geqq 2$.

We define μ_1 as the restriction of μ to $]u_2, u_1]$, to which we add the mass $\mu(\{\mu_2\}) - \mu_{x_1}(\{u_2\})$ at u_2. It is supported by $[u_2, u_1']$. To finish this case, we prove that μ_1 is packed before x_1. Since μ_1 is supported by $[u_2, u_1']$, so is $(\mu_1)_{x_1}$. Since $(\mu_1)_{x_1} \leqq \mu_{x_1}$, $(\mu_1)_{x_1}$ is supported by $\{u_2\}$. If $\mu_{x_1}(\{u_2\}) = 0$, the proof is finished. So suppose $h = \mu_{x_1}(\{u_2\}) > 0$. For $t > u_2$, we have $\mu_{x_1}([t, 1]) = 0$. Thus, for $u > u_2$,

$$\mu([t, 1]) = G(x_1, t) \leqq \mu([t, u[) + \nu([0, x_1] \times [u, 1]).$$

So $\mu([u, 1]) \leqq \nu([0, x_1] \times [u, 1])$ for $u > u_2$. This shows that

$$\inf_{u > u_2} (\mu([u_2, u[) + \nu([0, x_1] \times [u, 1])) \geqq \mu([u_2, 1]). \tag{71}$$

Since $\mu_{x_1}(]u_2, 1]) = 0$, we have

$$h = \mu_{x_1}(\{u_2\}) = \mu_{x_1}([u_2, 1]) = \mu([u_2, 1]) - G(x_1, u_2)$$

so that

$$\mu([u_2, 1]) - h = G(x_1, u_2) = \inf_{u \geqq u_2} (\mu([u_2, u[) + \nu([0, x_1] \times [u, 1])).$$

By (71), we have $G(x_1, u_2) = \nu([0, x_1] \times [u_2, 1])$, and thus, for $u > u_2$, by (71) again

$$\begin{aligned} \nu([0, x_1] \times [u_2, 1]) &= G(x_1, u_2) = \mu([u_2, 1]) - h \\ &\leqq \mu([u_2, u[) + \nu([0, x_1] \times [u, 1]) - h. \end{aligned}$$

Thus, replacing μ by μ_1 replaces $\mu([u_2, 1])$ by $\mu([u_2, 1]) - h$ but does not change $G(x_1, u_2)$. Thus, $(\mu_1)_{x_1}([u_2, 1]) = 0$, and this finishes the proof that μ_1 is packed before x_1.

For $x < x_1$, we have $\theta_i(x) < (s_i^! + 1)\Gamma_1(x)$, so that $\theta_i(x) - s_i^! \Gamma(x) \leqq \Gamma_1(x) \leqq u_1$. When $x \in L_i$ and $\theta(x) > u_1'$, we have $\Gamma_1(x) = \psi(x)$, and thus $\theta_i(x) - s(\theta(x), u_1')\psi(x) \leqq u_1 = 2^{-l}$. When $\theta(x) < u_1$, we also have $\theta(x) - \psi(x) \leqq 2^{-l}$. The definition of $\bar{\nu}$ shows that $\nu_1' = \bar{\nu}(\mu_1, \nu_0', u_1')$ does not charge $[0, x_1] \times]2^{-l}, 1]$. Also, ν_1' and ν' coincide on $[x_1, 1] \times [0, 1]$.

Consider $\mu_1' = \mu - \mu_1$. We perform the basic matching of μ_1' and ν_1' with parameter 1. Define

$$\Gamma_2'(x) = \inf\{a > 0;\ (\mu_1')_x(]a, 1]) = 0\}.$$

Obviously $\Gamma_2'(x) \leqq u_2$. We set $u_2' = \Gamma_2'(x_1)$, so $u_2' \leqq u_2$. We now perform the basic matching of μ_1' and ν_1' with parameter u_2', and we define

$$\Gamma_2(x) = \inf\{a > 0;\ (\mu_1')_x(]a, 1]) = 0\}.$$

Obviously, since ν_1' does not charge $[0, x_1] \times [2^{-l}, 1]$, Γ_2 and Γ_2' coincide on $[0, x_1]$, thus $u_2' = \Gamma_2(x_1)$. We define, for $i \leqq n$,

$$s_i^2 = \inf(\{s(\theta_i(y), u_2'), y > x_1\}).$$

We note that, by definition of x_1, there exists one $i \leqq n$ such that $\theta_i(y') \geqq (s_i^1 + 1)\Gamma_1(y')$ whenever $y' \geqq x_1$. So, if $x_1 < y' < y$, $\theta_i(y) \geqq \theta_i(y') \geqq (s_i^1 + 1)\Gamma_1(y')$. Letting $y' \to x_1$, since Γ_1 is right continuous, we get $\theta_i(y) \geqq (s_i^1 + 1)u_2 \geqq (s_i^1 + 1)u_2'$. Thus we have $s_i^2 \geqq s_i^1 + 1$.

We define

$$x_2 = \sup\{y > 0; \quad \forall i \leqq n,\ \theta_i(y) < (s_i^2 + 1)\Gamma_2(y)\}$$

$$u_3 = \lim_{x \to x_2^-} \Gamma_2(y),$$

and we proceed as in the first step of the construction to define μ_2. The measure $\nu_2' = \bar{\nu}(\mu_2, \nu_1', u_2')$ is then such that $\nu_2'([0, x_2] \times]2^{-l}, 1]) = 0$.

We continue this procedure until μ is exhausted. At each step k of the procedure there is one $i \leqq n$ such tht $s_i^{k+1} \geqq s_i^k + 1$. The numbers s_i^k are always less than $1/2^{-l-1} = 2^{l+1}$. This shows that the procedure terminates in at most $n2^{l+1}$ steps. The proof of Theorem 6 (and thus of Theorem 1) is complete.

REFERENCES

[1] J. L. BENTLEY, D. S. JOHNSON, F. T. LEIGHTON, C. C. MCGEOCH, AND L. A. MCGEOCH, *Some unexpected expected behavior results for bin packing*, Proc. ACM 16th Annual Symposium Theory of Computing, 1984, pp. 279-288.

[2] E. G. COFFMAN, JR., G. S. LUEKER, AND A. H. G. RINNOY KAN, *An introduction to the probabilistic analysis of sequencing and packing heuristics*, Management Sci., 34 (1988), pp. 266-290.

[3] S. FLOYD AND R. KARP, FFD *bin packing for item sizes with distributions on* [0, 1/2], Proc. IEEE 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 322-330.

[4] D. S. JOHNSON, A. DEMERS, J. D. ULLMAN, M. R. GAREY, AND R. L. GRAHAM, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM J. Comput., 3 (1974), pp. 299-325.

[5] J. F. C. KINGMAN AND S. J. TAYLOR, *Introduction to Measure and Probability*, Cambridge University Press, Cambridge, 1966.

[6] G. S. LUEKER, *An average-case analysis of bin packing with uniformly distributed item sizes*, Tech. Report No. 181, Department of Information and Computer Science, Univ. of California, Irvine, CA, 1982.

[7] W. RHEE AND M. TALAGRAND, *Optimal bin packing with items of random size* II, SIAM J. Comput., 18 (1989), pp. 139-151.

[8] ———, *Optimal bin packing with items of random size* III, SIAM J. Comput., 18 (1989), pp. 473-486.

[9] W. RHEE, *Stochastic Analysis of a Modified Version of First Fit Decreasing Packing*, 1988, preprint.

[10] W. RHEE AND M. TALAGRAND, *The complete convergence of best fit decreasing*, SIAM J. Comput., this issue, pp. 909-918.

[11] H. L. ROYDEN, *Real Analysis*, MacMillan, New York, 1968.

[12] W. RUDIN, *Real and Complex Analysis*, McGraw Hill, New York, 1974.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 939-954, October 1989

007

IMPROVED TIME BOUNDS FOR THE MAXIMUM FLOW PROBLEM*

RAVINDRA K. AHUJA†‡, JAMES B. ORLIN†, AND ROBERT E. TARJAN§

Abstract. Recently, Goldberg proposed a new approach to the maximum network flow problem. The approach yields a very simple algorithm running in $O(n^3)$ time on n-vertex networks. Incorporation of the dynamic tree data structure of Sleator and Tarjan yields a more complicated algorithm with a running time of $O(nm \log (n^2/m))$ on m-arc networks. Ahuja and Orlin developed a variant of Goldberg's algorithm that uses scaling and runs in $O(nm + n^2 \log U)$ time on networks with integer arc capacities bounded by U. In this paper possible improvements to the Ahuja-Orlin algorithm are explored. First, an improved running time of $O(nm + n^2 \log U/\log\log U)$ is obtained by using a nonconstant scaling factor. Second, an even better bound of $O(nm + n^2(\log U)^{1/2})$ is obtained by combining the Ahuja-Orlin algorithm with the wave algorithm of Tarjan. Third, it is shown that the use of dynamic trees in the latter algorithm reduces the running time to $O(nm \log((n/m)(\log U)^{1/2}+2))$. This result shows that the combined use of three different techniques results in speed not obtained by using any of the techniques alone. The above bounds are all for a unit-cost random access machine. Also considered is a semilogarithmic computation model in which the bounds increase by an additive term of $O(m \log_n U)$, which is the time needed to read the input in the model.

Key words. maximum flow, network algorithm, combinatorial optimization, scaling

AMS(MOS) subject classifications. 68Q20, 68Q25, 90C35

1. Introduction.

We consider algorithms for the classical maximum network flow problem [5], [6], [13], [15], [20]. We formulate the problem as follows. Let $G = (V, E)$ be a directed graph with vertex set V and arc set E. The graph G is a *flow network* if it has two distinct distinguished vertices, a *source* s and a *sink* t, and a nonnegative real-valued *capacity* $u(v, w)$ on each arc $(v, w) \in E$. We assume that G is *symmetric*, i.e., $(v, w) \in E$ if and only if $(w, v) \in E$. We denote by n, m, and U the number of vertices, the number of arcs, and the maximum arc capacity, respectively. For ease in stating time bounds, we assume $m \geqq n$ and $U \geqq 4$. Bounds containing U are subject to the assumption that all arc capacities are integral. All logarithms in the paper are base two unless an explicit base is given.

A *flow* f on a network G is a real-valued function f on the arcs satisfying the following constraints:

$$f(v, w) \leqq u(v, w) \text{ for all } (v, w) \in E \text{ (capacity constraint);} \tag{1}$$

$$f(v, w) = -f(w, v) \text{ for all } (v, w) \in E \text{ (antisymmetry constraint);} \tag{2}$$

$$\sum_{(v,w)\in E} f(v, w) = 0 \text{ for all } w \in V - \{s, t\} \text{ (conservation constraint).} \tag{3}$$

* Received by the editors November 25, 1987; accepted for publication (in revised form) September 26, 1988.

† Sloan School of Management, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The research of these authors was supported by a Presidential Young Investigator Fellowship for the National Science Foundation, contract 8451517 ECS, grant AFOSR-88-0088 from the Air Force Office of Scientific Research, and grants from Analog Devices, Apple Computer Inc., and Prime Computer.

‡ This work was done while the author was on leave from the Indian Institute of Technology, Kanpur, India.

§ Department of Computer Science, Princeton University, Princeton, New Jersey 08544, and AT&T Bell Labs, Murray Hill, New Jersey 07974. The research of this author was partially supported by National Science Foundation grant DCR-8605962 and Office of Naval Research contract N00014-87-K-0467.

The *value* $|f|$ of a flow f is the net flow into the sink:

$$|f| = \sum_{(v,t)\in E} f(v, t).$$

A *maximum flow* is a flow of maximum value. The *maximum flow problem* is that of finding a maximum flow in a given network.

Remark. We assume that all arc capacities are finite. If some arc capacities are infinite but no path of infinite-capacity arcs from s to t exists, then each infinite capacity can be replaced by the sum of the finite capacities without affecting the problem. □

The maximum flow problem has a long, rich history, and a series of faster and faster algorithms for the problem has been developed. (See [10] for a brief survey.) Until now the fastest known algorithms have been the one of Goldberg and Tarjan [9], [10] (running in time $O(nm \log (n^2/m))$) and the one of Ahuja and Orlin [1] (running in time $O(nm + n^2 \log U)$). Both of these algorithms are refinements of a generic method proposed by Goldberg [8], which we shall call the *preflow algorithm.* For networks with $m = \Omega(n^2)$, the Goldberg-Tarjan bound is $O(n^3)$, which matches the bound of several earlier algorithms [12], [14], [16], [21]. For networks with $m = O(n^{2-\varepsilon})$ for some constant $\varepsilon > 0$, the Goldberg-Tarjan bound is $O(nm \log n)$, which matches the bound of the earlier Sleator-Tarjan algorithm [17], [18]. Under the *similarity assumption* [7], namely, $U = O(n^\gamma)$ for some constant γ, the Ahuja-Orlin bound beats the Goldberg-Tarjan bound unless $m = O(n)$ or $m = \Omega(n^2)$.

The Goldberg-Tarjan and Ahuja-Orlin algorithms obtain their speed from two different techniques. The former uses a sophisticated data structure, the dynamic tree structure of Sleator and Tarjan [18], [19], [20], whereas the latter uses scaling. In this paper we explore improvements in the Ahuja-Orlin algorithm obtained by incorporating other ideas, including the use of dynamic trees. We begin in § 2 by reviewing the generic preflow algorithm [8], [9], [10]. In § 3, we develop a version of the Ahuja-Orlin algorithm that uses a stack-based vertex selection rule and a nonconstant scaling factor to obtain a time bound of $O(nm + n^2 \log U/\log\log U)$. In § 4, we describe an even faster variant that uses a constant scaling factor but combines the scaling idea with the *wave algorithm* of Tarjan [21]. This algorithm has a running time of $O(nm + n^2 (\log U)^{1/2})$. In § 5, we add dynamic trees to the method of § 4, thereby obtaining a running time of $O(nm \log ((n/m)(\log U)^{1/2}+2))$. The results in §§ 3-5 rely on the assumption that pointer manipulations and arithmetic operations on integers of magnitude U take $O(1)$ time. In § 6, we consider the effect on our time bounds of a weaker assumption, namely, that arithmetic on integers of magnitude n takes $O(1)$ time. The extra time needed by our algorithms in this *semilogarithmic computation model* is an additive term of $O(m \log_n U)$, which is the same as the time needed to read the input. We conclude in § 7 with some final remarks.

2. The preflow algorithm. In contrast to the classical *augmenting path method* of Ford and Fulkerson [6] that moves flow along an entire path from s to t at once, the preflow method moves flow along a single arc at a time. The key concept underlying the algorithm is that of a *preflow*, introduced by Karzanov [12]. A preflow f is a real-valued function on the arcs satisfying constraints (1), (2), and a relaxation of (3). For any vertex w, let the *flow excess* of w be $e(w) = \sum_{(v,w)\in E} f(v, w)$. The required constraint is the following:

$$e(w) \geqq 0 \quad \forall w \in V - \{s\} \text{ (preflow constraint).} \tag{4}$$

We call a vertex v *active* if $v \neq t$ and $e(v) > 0$. Observe that the preflow constraint implies that $e(s) \leqq 0$.

The *residual capacity* of an arc (v, w) with respect to a preflow f is $u_f(v, w) = u(v, w) - f(v, w)$. An arc is *saturated* if $u_f(v, w) = 0$ and *unsaturated* otherwise. (The capacity constraint implies that any unsaturated arc (v, w) has $u_f(v, w) > 0$.)

The preflow algorithm maintains a preflow and moves flow from active vertices through unsaturated arcs toward the sink, along paths estimated to contain as few arcs as possible. Excess flow that cannot be moved to the sink is returned to the source, also along estimated shortest paths. Eventually the preflow becomes a maximum flow.

As an estimate of path lengths, the algorithm uses a *valid labeling* that is a function d from the vertices to the nonnegative integers such that $d(s) = n$, $d(t) = 0$, and $d(v) \leqq d(w) + 1$ for every unsaturated arc (v, w). A proof by induction shows that, for any valid labeling d, $d(v) \leqq \min\{d_f(v, s) + n, d_f(v, t)\}$, where $d_f(v, w)$ is the minimum number of arcs on a path from v to w consisting of arcs unsaturated with respect to the flow f. We call an arc (v, w) *eligible* if (v, w) is unsaturated and $d(v) = d(w) + 1$.

The algorithm begins with an initial preflow f and a valid labeling d defined as follows:

$$f(v, w) = \begin{cases} u(v, w) & \text{if } v = s, \\ -u(w, v) & \text{if } w = s, \\ 0 & \text{if } v \neq s \text{ and } w \neq s, \end{cases}$$

$$d(v) = \min\{d_f(v, s) + n, d_f(v, t)\}.$$

The algorithm consists of repeating the following two steps, in any order, until no vertex is active:

Push(v, w).

Applicability: Vertex v is active and arc (v, w) is eligible.

Action: Increase $f(v, w)$ by $\min\{e(v), u_f(v, w)\}$. The push is *saturating* if (v, w) is saturated after the push and *nonsaturating* otherwise.

Relabel(v).

Applicability: Vertex v is active and no arc (v, w) is eligible.

Action: Replace $d(v)$ by $\min\{d(w) + 1 \mid (v, w) \text{ is unsaturated}\}$.

When the algorithm terminates, f is a maximum flow. Goldberg and Tarjan [10] derived the following bounds on the number of steps required by the algorithm:

LEMMA 2.1 [10]. *Relabeling a vertex v strictly increases $d(v)$. No vertex label exceeds $2n - 1$, and the total number of relabelings is $O(n^2)$.*

LEMMA 2.2 [10]. *There are at most $O(mn)$ saturating pushes and at most $O(n^2m)$ nonsaturating pushes.*

Efficient implementations of the above algorithm require a mechanism for selecting pushing and relabeling steps to perform. Goldberg and Tarjan proposed the following method: For each vertex, construct a (fixed) list $A(v)$ of the arcs out of v. Designate one of these arcs, initially the first on the list, as the *current arc* out of v. To execute the algorithm, repeat the following step until there are no active vertices:

Push/Relabel(v).

Applicability: Vertex v is active.

Action: If the current arc (v, w) of v is eligible, perform *push*(v, w). Otherwise, if (v, w) is not the last arc on $A(v)$, make the next arc after (v, w) the

current one. Otherwise, perform *relabel*(v) and make the first arc on $A(v)$ the current one.

With this implementation, the algorithm runs in $O(nm)$ time plus $O(1)$ time per nonsaturating push. This gives an $O(n^2m)$ time bound for any order of selecting vertices for push/relabel steps. Making the algorithm faster requires reducing the time spent on nonsaturating pushes. The number of such pushes can be reduced by selecting vertices for push/relabel steps carefully. Goldberg and Tarjan showed that first-in, first-out selection (first active, first selected) reduces the number of nonsaturating pushes to $O(n^3)$. Cheriyan and Maheshwari [3] showed that highest label selection (always pushing flow from a vertex with highest label) reduces the number of nonsaturating pushes to $O(n^2m^{1/2})$. (The latter rule was first proposed by Goldberg [8], who gave an $O(n^3)$ bound.) Ahuja and Orlin proposed a third selection rule, which we discuss in the next section.

3. The scaling algorithm. The intuitive idea behind the Ahuja-Orlin algorithm, henceforth called the *scaling algorithm*, is to move large amounts of flow when possible. The same idea is behind the maximum capacity augmenting path method of Edmonds and Karp [4] and the capacity scaling algorithm of Gabow [7]. One way to apply this idea to the preflow algorithm is to always push flow from a vertex of large excess to a vertex of small excess, or to the sink. The effect of this is to reduce the maximum excess at a rapid rate.

Making this method precise requires specifying when an excess is large and when it is small. For this purpose the scaling algorithm uses an *excess bound* Δ and an integer *scaling factor* $k \geqq 2$. A vertex v is said to have *large excess* if its excess exceeds Δ/k and *small excess* otherwise. As the algorithm proceeds, k remains fixed, but Δ periodically decreases. Initially, Δ is the smallest power of k such that $\Delta \geqq U$. The algorithm maintains the invariant that $e(v) \leqq \Delta$ for every active vertex v. This requires changing the pushing step to the following:

Push (v, w).

Applicability: Vertex v is active and arc (v, w) is eligible.

Action: If $w \neq t$, increase $f(v, w)$ by $\min\{e(v), u_f(v, w), \Delta - e(w)\}$. Otherwise, $(w = t)$, increase $f(v, w)$ by $\min\{e(v), u_f(v, w)\}$.

The algorithm consists of a number of *scaling phases*, during each of which Δ remains constant. A phase consists of repeating push/relabel steps, using the following selection rule, until no active vertex has large excess, and then replacing Δ by Δ/k. The algorithm terminates when there are no active vertices.

Large excess, smallest label selection: Apply a push/relabel step to an active vertex v of large excess; among such vertices, choose one of smallest label.

If the edge capacities are integers, the algorithm terminates after at most $\lfloor \log_k U + 1 \rfloor$ phases. After $\lfloor \log_k U + 1 \rfloor$ phases, $\Delta < 1$, which implies that f is a flow, since the algorithm maintains integrality of vertex excesses. Ahuja and Orlin derived a bound of $O(kn^2 \log_k U)$ on the total number of nonsaturating pushes. We repeat the analysis here, since it provides motivation for our first modification of the algorithm.

LEMMA 3.1 [1]. *The total number of nonsaturating pushes in the scaling algorithm is* $O(kn^2(\log_k U + 1))$.

Proof. Consider the function $\Phi = \sum_{v \text{ active}} e(v)\, d(v)/\Delta$. We call Φ the *potential* of the current preflow f and labeling d. Since $0 < e(v)/\Delta \leqq 1$ and $0 \leqq d(v) \leqq 2n$ for every active vertex v, $0 \leqq \Phi \leqq 2n^2$ throughout the algorithm. Every pushing step decreases Φ. A nonsaturating pushing step decreases Φ by at least $1/k$, since the push is from a

vertex v with excess more than Δ/k to a vertex w with $d(w) = d(v) - 1$, and $e(w) \leqq \Delta/k$ or $w = t$. The value of Φ can increase only during a relabeling or when Δ changes. A relabeling of a vertex v increases Φ by at most the amount $d(v)$ increases. Thus the total increase in Φ due to relabelings, over the entire algorithm, is at most $2n^2$. When Δ changes, Φ increases by a factor of k, to at most $2n^2$. This happens at most $\lfloor \log_k U + 1 \rfloor$ times. Thus the total increase in Φ over the entire algorithm is at most $2n^2 \lfloor \log_k U + 2 \rfloor$. The total number of nonsaturating pushes is at most k times the sum of the initial value of Φ and the total increase in Φ. This is at most $2kn^2 \lfloor \log_k U + 3 \rfloor$. □

Choosing k to be a constant independent of n gives a total time bound of $O(nm + n^2 \log U)$ for the scaling algorithm, given an efficient implementation of the vertex selection rule. One way to implement the rule is to maintain an array of sets indexed by vertex label, each set containing all large excess vertices with the corresponding label, and to maintain a pointer to the nonempty set of smallest index. The total time needed to maintain this structure is $O(nm + n^2 \log U)$.

Having described the scaling algorithm, we consider the question of whether its running time can be improved by reducing the number of nonsaturating pushes. The proof of Lemma 3.1 bounds the number of nonsaturating pushes by estimating the total increase in the potential Φ. Observe that there is an imbalance in this estimate: $O(n^2 \log_k U)$ of the increase is due to phase changes, whereas only $O(n^2)$ is due to relabelings. Our plan is to improve this estimate by decreasing the contribution of the phase changes, at the cost of increasing the contribution of the relabelings. Making this plan work requires changing the algorithm.

We use a nonconstant scale factor k and a slightly more elaborate method of vertex selection. We make use of the *stack-push/relabel* step defined below, which performs a sequence of push and relabel steps using a stack. The stack provides an alternative way of avoiding pushes to large-excess vertices (other than t).

Stack-Push/Relabel(r).

Applicability: Vertex r is active.

Action: Initialize a stack S to contain r. Repeat the following step until S is empty:

Stack Step. Let v be the top vertex on S and let (v, w) be the current arc out of v. Apply the appropriate one of the following cases:

Case 1: (v, w) *is not eligible.*

Case 1a: (v, w) *is not last on* $A(v)$. Replace (v, w) as the current arc out of v by the next arc on $A(v)$.

Case 1b: (v, w) *is last on* $A(v)$. Relabel v and pop it from S. Replace (v, w) as the current arc out of v by the first arc on $A(v)$.

Case 2: (v, w) *is eligible.*

Case 2a: $e(w) > \Delta/2$ and $w \neq t$. Push w onto S.

Case 2b: $e(w) \leqq \Delta/2$ or $w = t$. Perform *push*(v, w) (modified as at the beginning of this section to maintain $e(w) \leqq \Delta$ if $w \neq t$). If $e(v) = 0$, pop v from S.

Some remarks about *stack-push/relabel* are in order. Let us call a nonsaturating push *big* if it moves at least $\Delta/2$ units of flow and *little* otherwise. During an execution of *stack-push/relabel*, every vertex pushed onto S, except possibly the first, has an excess of at least $\Delta/2$ when it is added to S. A vertex v can be popped from S only after it is relabeled or its excess is reduced to zero. Of the pushes from v while v is on S, at most two are nonsaturating, only the last of which can be little.

Our variant of the scaling algorithm, called the *stack scaling algorithm*, consists of phases just as in the scaling algorithm. A phase consists of repeatedly applying the *stack-push/relabel* step to a large-excess active vertex of highest label; a phase ends when there are no large-excess active vertices.

LEMMA 3.2. *The total number of nonsaturating pushes made by the stack scaling algorithm is* $O(kn^2+n^2(\log_k U+1))$.

Proof. To bound the number of nonsaturating pushes, we use an argument like the proof of Lemma 3.1, but with two potentials instead of one. The first potential is that of Lemma 3.1, namely, $\Phi=\sum_{v \text{ active}} e(v)\, d(v)/\Delta$. By the analysis in the proof of Lemma 3.1, every push decreases Φ, the total increase in Φ over all phases is $O(n^2 \log_k U)$, and the difference between the initial and the final values of Φ is $O(n^2)$. Each big push moves at least $\Delta/2$ units of flow and hence decreases Φ by at least $\frac{1}{2}$. Thus the number of big pushes is $O(n^2(\log_k U+1))$.

To count little pushes, we divide them into two kinds, those that result in an empty stack S, called *emptying pushes*, and those that do not, called *nonemptying pushes*. A nonemptying push from a vertex v is such that $e(v)$ was at least $\Delta/2$ when v was added to S, and the push results in $e(v)$ decreasing to zero. We can charge such a push against the cumulative decrease of at least $\frac{1}{2}$ in Φ resulting from moving the original excess on v to vertices of smaller label. Hence there can be only $O(n^2(\log_k U+1))$ nonemptying pushes.

An emptying push from a vertex v can be associated with a decrease of at least $1/k$ in Φ, namely, the drop in Φ caused by the movement of the original excess on v, which is at least Δ/k, to smaller labeled vertices. But using this drop gives a bound on the number of emptying pushes of only $O(kn^2(\log_k U+1))$. We count emptying pushes more carefully by using a second potential, Φ_2. The definition of Φ_2 involves two parameters, an integer l and a set P. The value of l is equal to the minimum of $2n$ and the smallest label of a vertex added to S while S was empty during the current phase. Observe that $l=2n$ at the beginning of a phase and that l is nonincreasing during a phase. The set P contains all vertices that have label greater than l, and also all vertices that have label equal to l and from which an emptying push has been made during the current phase. Observe that P is empty at the beginning of a phase and P never loses a vertex during a phase. The definition of Φ_2 is

$$\Phi_2=\sum_{v\in P:e(v)>0} e(v)(d(v)-l+1)/\Delta. \text{ (If } P=\varnothing, \text{ then } \Phi_2=0.)$$

Observe that $0\leqq\Phi_2\leqq 2n^2$. Any emptying push can be associated either with an addition of a vertex to P or with a decrease in Φ_2 of at least $1/k$. (If the push is from v, either $v\notin P$ when v is added to S but $v\in P$ after the push, or $v\in P$ when v is added to S and Φ_2 drops by at least $1/k$ because of pushes from v while v is on S.) The number of vertices added to P is at most $n\lfloor\log_k U+1\rfloor$ over all phases, and hence so is the number of emptying pushes not associated with decreases in Φ_2.

To bound the number of emptying pushes associated with decreases in Φ_2, we bound the total increase in Φ_2. Increases in Φ_2 are due to relabelings and to decreases in l. (A vertex added to P because of an emptying push has zero excess and hence adds nothing to Φ_2.) A relabeling of a vertex v increases Φ_2 by at most the increase in $d(v)$ plus one; the "plus one" accounts for the fact that the relabeling may add v to P. Thus relabelings contribute at most $4n^2$ to the growth of Φ_2.

There are at most $2n$ decreases in l per phase. A decrease in l by one adds at most n/k to Φ_2, since when the decrease occurs every vertex in P has small excess. (Such a decrease occurs because some vertex v of label less than l is added to S while

S is empty. The label of v becomes the new value of l. When this happens, P contains only vertices of label exceeding the new value of l, all of which must have small excess.) Thus the total increase in Φ_2 due to decreases in l is at most $2n^2\lfloor \log_k U+1\rfloor/k$ over all phases.

The total number of emptying pushes associated with decreases in Φ_2 is at most k times the total increase in Φ_2, since Φ_2 is initially zero. Thus the total number of such pushes is at most $4kn^2+2n^2\lfloor \log_k U+1\rfloor$. □

As in the Goldberg-Tarjan and Ahuja-Orlin algorithms, the time to perform saturating pushes and relabeling operations and to examine ineligible arcs is $O(nm)$. The only significant remaining issue is how to choose vertices to add to S when S is empty. For this purpose, we maintain a data structure consisting of a collection of doubly linked lists: $list(j)=\{i\in N: e(i)>\Delta/k \text{ and } d(i)=j\}$ for each $j\in\{1, 2, \cdots, 2n-1\}$. We also maintain a pointer to indicate the largest index j for which $list(j)$ is nonempty. Maintaining this structure requires $O(1)$ time per push operation plus time to maintain the pointer. Each pointer increase is due to a relabeling (the increase is at most the amount of change in label) or due to a phase change (the increase is at most $2n$). Consequently, the number of times the pointer needs to be incremented or decremented is $O(n^2+n(\log_k U+1))$. The overall running time of the algorithm is thus $O(nm+kn^2+n^2(\log_k U+1))$. Choosing $k=\lceil \log U/\log\log U\rceil$ gives the following result:

THEOREM 3.3. *The stack scaling algorithm, with an appropriate choice of k, runs in* $O(nm+n^2\log U/\log\log U)$ *time.*

4. The wave scaling algorithm. Another way to reduce the number of nonsaturating pushes in the scaling algorithm is to keep track of the *total excess*, defined to be the sum of the excesses of all active vertices. The key observation is that if the total excess is sufficiently large, the algorithm can make significant progress by applying *stack-push/relabel* to each active vertex in turn, processing vertices in topological order with respect to the set of eligible arcs. Even though some vertices of very small excess are processed, the overall benefits of this approach yield an even better running time than that of the stack scaling algorithm. The idea of processing vertices in topological order originated in the *wave algorithm* of Tarjan [20], [21]; therefore, we call the new algorithm the *wave scaling algorithm.*

The wave scaling algorithm seems to derive no benefit from using a nonconstant scaling factor; therefore, we fix $k=2$. The algorithm uses another parameter, $l\geqq 1$, whose exact value we shall choose later. A phase of the wave scaling algorithm consists of two parts. First, the *wave step* below is repeated until the total excess is less than $n\Delta/l$. Then *stack-push/relabel* steps are applied to large-excess active vertices in any order until there are no large-excess active vertices.

Wave: Construct a list L of the vertices in $V-\{s, t\}$ in topological order with respect to the set of eligible arcs. (Ordering L in nonincreasing order by vertex label suffices.) For each vertex v on L, if v is active and v has not been relabeled during the current wave, then perform *stack-push/relabel*(v).

Observe that the total excess is a nonincreasing function of time. Constructing L at the beginning of a wave takes $O(n)$ time using a radix sort by vertex label. The time spent during a single wave is $O(n)$ plus time for the relabelings and pushes. Thus the total time required by the wave scaling algorithm is $O(nm)$ plus $O(n)$ per wave plus $O(1)$ per nonsaturating push. We complete the running time analysis with two lemmas.

LEMMA 4.1. *The number of nonsaturating pushes done by the wave scaling algorithm is* $O(n^2+(n^2/l)\log U)$ *plus* $O(n)$ *per wave.*

Proof. We count big pushes (those that move at least $\Delta/2$ units of flow) and little pushes separately. The analysis in the proof of Lemma 3.1 applies to bound the number of big pushes, with the following improvement. At the end of a phase, the total excess is less than $n\Delta/l$. Thus the potential at the beginning of the next phase is at most $4n^2/l$. (The new value of Δ is half of the old; each vertex label is at most $2n$.) Thus the sum of the increases in Φ caused by changes in Δ is $O((n^2/l)\log U)$, which implies that the number of big pushes is $O(n^2+(n^2/l)\log U)$. The same argument gives the same bound on the number of little pushes from vertices that have large excess when added to the stack S, since as in the proof of Lemma 3.2 each such push can be charged against a drop of at least $\frac{1}{2}$ in Φ. The remaining little pushes consist of at most one per vertex to which *stack-push/relabel* is applied during waves, totaling at most n per wave. □

LEMMA 4.2. *The number of waves in the wave scaling algorithm is* $O(\min\{nl+\log U, n^2\})$.

Proof. Consider any wave except the last in a phase. At the end of the wave, the total excess is at least $n\Delta/l$. The only way for excess to remain on a vertex x at the end of a wave is for x to have been relabeled during the wave; once x is relabeled, $e(x)$ remains constant until the end of the wave. The maximum excess on any single vertex is Δ. It follows that at least n/l relabelings must have occurred during the wave. Since the total number of relabelings is $O(n^2)$, the total number of waves is $O(nl+\log U)$. Furthermore, a wave during which no relabeling occurs causes all vertices to become inactive, and hence terminates the entire algorithm. Thus the number of waves is $O(n^2)$. □

THEOREM 4.3. *The running time of the wave scaling algorithm is* $O(nm+n^2(\log U)^{1/2})$ *if* l *is chosen equal to* $(\log U)^{1/2}$.

Proof. By Lemmas 4.1 and 4.2, the running time of the wave scaling algorithm is $O(nm+(n^2/l)\log U+\min\{n^2l+n\log U, n^3\})$. Choosing $l=(\log U)^{1/2}$ gives a bound of $O(nm+n^2(\log U)^{1/2}+\min\{n\log U, n^3\})$. But $\min\{n\log U, n^3\}\leqq n^2(\log U)^{1/2}$, since $n\log U\geqq n^2(\log U)^{1/2}$ implies $(\log U)^{1/2}\geqq n$. □

5. Use of dynamic trees. The approach taken in §§ 3 and 4 reduced the total number of pushes. An orthogonal approach is to reduce the total time of the pushes without necessarily reducing their number. This can be done by using the dynamic tree data structure of Sleator and Tarjan [18], [19], [20]. We conjecture that, given a version of the preflow algorithm with a bound of $p\geqq nm$ on the total number of pushes, the running time can be reduced from $O(p)$ to $O(nm\log((p/nm)+1))$ by using dynamic trees. Although we do not know how to prove a general theorem to this effect, we have been able to obtain such a result for each version of the preflow algorithm that we have considered. As an example, the $O(nm\log(n^2/m))$ time bound of Goldberg and Tarjan results from using dynamic trees with the first-in, first-out selection rule; the bound on the number of pushes in this case is $O(n^3)$. In this section we shall show that the same idea applies to the wave scaling algorithm of § 4, resulting in a time bound of $O(nm\log((n/m)(\log U)^{1/2}+2))$. This idea can also be applied to the stack scaling algorithm of § 3, giving a bound of $O(nm\log((n\log U/m\log\log U)+2))$; we omit a description of the latter result since the former bound is better.

The dynamic tree data structure allows the maintenance of a collection of vertex-disjoint rooted trees, each arc of which has an associated real value. We regard each tree arc as being directed from child to parent, and we regard every vertex as being

both an ancestor and a descendant of itself. The data structure supports the following seven operations:

find-root(v):	Find and return the root of the tree containing vertex v.
find-size(v):	Find and return the number of vertices in the tree containing vertex v.
find-value(v):	Find and return the value of the tree arc leaving v. If v is a tree root, the value returned is infinity.
find-min(v):	Find and return the ancestor w of v with minimum *find-value*(w). In case of a tie, choose the vertex w closest to the tree root.
change-value(v, x):	Add real number x to the value of every arc along the path from v to the root of its tree.
link(v, w, x):	Combine the trees containing v and w by making w the parent of v and giving the arc (v, w) the value x. This operation does nothing if v and w are in the same tree or if v is not a tree root.
cut(v):	Break the tree containing v into two trees by deleting the arc joining v to its parent; return the value of the deleted arc. This operation breaks no arc and returns infinity if v is a tree root.

A sequence of h tree operations, starting with an initial collection of singleton trees, takes $O(h \log (z+1))$ time if z is the maximum tree size [10], [18], [19], [20].

In the network flow application, the dynamic tree arcs are a subset of the current arcs out of the vertices. Only eligible arcs are tree arcs. The value of a tree arc is its residual capacity. The dynamic tree data structure allows flow to be moved along an entire path at once, rather than along a single arc at a time. We shall describe a version of the wave scaling algorithm, which we call the *tree scaling algorithm*, that uses this capability. Two parameters govern the behavior of the algorithm, a variable bound Δ on the maximum excess at an active vertex and a fixed bound z, $1 \leqq z \leqq n$, on the maximum size of a dynamic tree. The algorithm is identical to the wave scaling algorithm except that it uses the following step in place of *stack-push/relabel*:

Tree-Push/Relabel(r)

Applicability: Vertex r is active.

Action: Initialize a stack S to contain r. Repeat the following step until S is empty.

Stack Step. Let v be the top vertex on S and let (v, w) be the current arc out of S. Apply the appropriate one of the following cases:

Case 1: (v, w) *is not eligible.*

Case 1a: (v, w) *is not last on* $A(v)$. Replace (v, w) as the current arc out of v by the next arc on $A(v)$.

Case 1b: (v, w) *is last on* $A(v)$. Relabel v and pop it from S. Replace (v, w) as the current arc out of v by the first arc on $A(v)$. For every tree arc (y, v), perform *cut*(y).

Case 2: (v, w) *is eligible.* Let $x =$ *find-root*(w).

Case 2a: $e(x) > \Delta/2$ *and* $x \neq t$. Push x onto S.

Case 2b: $e(x) \leqq \Delta/2$ *or* $x = t$. Let $\varepsilon = \min \{e(v), u_f(v, w),$ *find-min*$(w)\}$. Let $\delta = \varepsilon$ if $x = t$, $\delta = \min \{\varepsilon, \Delta - e(x)\}$ if $x \neq t$. Send δ units of flow from v to x by increasing $f(v, w)$ by δ and performing *change-value*$(w, -\delta)$. (This is called a *tree push* from v to x. The tree push is *saturating* if $\delta = \min \{u_f(v, w),$ *find-min*$(w)\}$ before the push and *nonsaturating* otherwise.) While it is the case that *find-value*(*find-*

min$(w)) = 0$, perform *cut*(*find-min*(w)). If $e(v) = 0$, pop v from S, and if in addition $u_f(v, w) > 0$ and *find-size*(w) + *find-size*$(v) \leqq z$, perform *link*$(v, w, u_f(v, w))$.

The tree scaling algorithm stores flow in two different ways; explicitly for arcs that are not dynamic tree arcs and implicitly (in the dynamic tree data structure) for arcs that are dynamic tree arcs. After each cut, the flow on the arc cut must be restored to its correct current value. In addition, when the algorithm terminates, the correct flow value on each remaining tree arc must be computed. For arcs cut during the computation, the desired flow values are returned by the corresponding *cut* operations. Computing correct flow values on termination can be done using at most $n-1$ *find-value* operations. We have omitted these bookkeeping steps from our description of the algorithm.

The algorithm maintains the following invariants: every active vertex is a tree root, every tree arc is eligible, no excess exceeds Δ, and no tree size exceeds z. Let us call a nonsaturating tree push *big* if it moves at least $\Delta/2$ units of flow, and *little* otherwise. Of the pushes from a given vertex v while v is on S, at most two are nonsaturating and at most one (the last) is small.

The tree scaling algorithm is a variant of the dynamic tree algorithm of Goldberg and Tarjan [9], [10]. Their analysis applies to give the following result:

LEMMA 5.1 [10]. *The total time required by the tree scaling algorithm is* $O(nm \log (z+1))$ *plus* $O(\log (z+1))$ *time per tree push plus the time needed to construct and scan the list* L *during wave steps. The number of links, cuts, and saturating tree pushes is* $O(nm)$.

Making the tree scaling algorithm efficient requires careful implementation of the list L used in *wave* steps. For the moment we shall ignore the time spent manipulating L. The remaining issue in analyzing the algorithm is bounding the number of nonsaturating pushes. To do this we need two lemmas:

LEMMA 5.2. *The number of waves in the tree scaling algorithm is* $O(\min\{nl + \log U, n^2\})$.

Proof. Identical to the proof of the same result for the wave scaling algorithm (Lemma 4.2). □

LEMMA 5.3. *The number of nonsaturating tree pushes done by the tree scaling algorithm is* $O(nm + (n^2/l) \log U)$ *plus* $O(n/z)$ *per wave.*

Proof. The potential function Φ used in § 3, as applied in the proof of Lemma 3.3, serves to bound the number of big tree pushes by $O(n^2 + (n^2/l) \log U)$. We count little tree pushes as follows. Any little tree push, say from a vertex v, reduces $e(v)$ to zero. We shall count such a push against the most recent previous push that made $e(v)$ nonzero; or, if there is no such previous push, against the preflow initialization. We call the event of $e(v)$ becoming nonzero an *activation* of v. We shall derive a bound of $O(nm + (n^2/l) \log U)$ plus $O(n/z)$ per wave on the number of activations, thereby proving the lemma.

Initializing the preflow f at the beginning of the entire algorithm causes $O(n)$ vertex activations. Every other activation is due to a tree push (Case 2b of *tree-push/relabel*). If such a tree push results in a link or cut, we charge the activation against the corresponding link or cut. Similarly, if the tree push is saturating, we charge the activation against the arc saturation. Such charges account for $O(nm)$ activations.

Any remaining activation, say of a vertex x, is produced by a tree push through an arc (v, w) to $x =$ *find-root*(w), after which *find-size*(v) + *find-size*$(w) > z$, since no

link is performed. Let T_v and T_w be the dynamic trees containing v and w, respectively. We call a dynamic tree *large* if it contains more than $z/2$ vertices and *small* otherwise. Just after the push, one of T_v and T_w must be large.

If the tree push is big, we charge the activation of x against the push. By the argument above counting big tree pushes, this accounts for $O(n^2+(n^2/l)\log U)$ activations. Otherwise, the push is little, and thus it reduces $e(v)$ to zero. There can be at most one such push from v per wave. If T_v has changed between the beginning of the most recent wave and the push, we charge the activation of x to the link or cut that most recently changed T_v. There are $O(nm)$ such charges, at most one per link and at most two per cut, since vertex v is the root of T_v when the push occurs. Similarly, if T_w has changed between the beginning of the most recent wave and the push, we charge the activation of x to the link or cut that most recently changed T_w. There are $O(nm)$ such charges, since a vertex x can be activated at most once per wave, and x is the root of T_w when the push occurs. If neither T_v nor T_w has changed, we charge the activation of x to whichever of T_v or T_w is large. Each large tree existing at the beginning of a wave can be charged at most twice (once as a T_v, once as a T_w). Since there are fewer than $2n/z$ large trees at the beginning of any wave, the total number of such charges is $O(n/z)$ per wave. Combining all our estimates gives the claimed bound on vertex activations. □

THEOREM 5.4. *The number of tree pushes done by the tree scaling algorithm is* $O(nm)$ *if* z *is chosen equal to* $\min\{n, \max\{1, (n^2/m^2)\log U\}\}$, *and* l *is chosen equal to* m/n *if* $z=1$, $(n/m)\log U$ *if* $1<z\leqq n$.

Proof. By Lemmas 5.2 and 5.3, the number of tree pushes done by the algorithm is $O(nm+(n^2/l)\log U+(n/z)\min\{nl+\log U, n^2\})$. We consider three cases:

Case 1. $(n^2/m^2)\log U\leqq 1$. Then $z=1$ and $l=m/n\geqq 1$. The number of tree pushes is $O(nm+(n^3/m)\log U+n\min\{m+\log U, n^2\})$. Since $\log U\leqq m^2/n^2$, the bound on tree pushes is $O(nm+\min\{m^2/n, n^3\})=O(nm)$, because $m^2/n\leqq n^3$ implies $m\leqq n^2$, which means $m^2/n\leqq nm$, and $m^2/n\geqq n^3$ implies $m\geqq n^2$, which means $n^3\leqq nm$. (This analysis allows for the possiblity of multiple arcs in G, i.e., the possibility that $m>n^2$.)

Case 2. $1<(n^2/m^2)\log U<n$. Then $z=(n^2/m^2)\log U$ and $l=(n/m)\log U\geqq 1$. The number of tree pushes is $O(nm+(m^2/n\log U)\min\{(n^2/m)\log U+\log U, n^2\})=O(nm+\min\{m^2/n, m^2n/\log U\})=O(nm+\min\{m^2/n, n^3\})$, since $\log U>m^2/n^2$. Since $\min\{m^2/n, n^3\}\leqq nm$ (see Case 1), the number of tree pushes is $O(nm)$. (As in Case 1, this analysis allows for the possiblity that $m>n^2$.)

Case 3: $(n^2/m^2)\log U\geqq n$. Then $z=n$ and $l=(n/m)\log U\geqq 1$. The number of tree pushes is $O(nm+\min\{(n^2/m)\log U+\log U, n^2\})=O(nm)$. □

Observe that if z is chosen as in Theorem 5.4, i.e., equal to $\min\{n, \max\{1, (n^2/m^2)\log U\}\}$, the time per dynamic tree operation is $O(\log(z+1))=O(\log((n/m)(\log U)^{1/2}+2))$. Thus the running time of the tree scaling algorithm, with k and z chosen as in Theorem 5.4, is $O(nm\log((n/m)(\log U)^{1/2}+2))$ (ignoring time spent manipulating L).

All that remains is to show that the manipulations of the list L in *wave* steps can be performed in $O(nm\log((n/m)(\log U)^{1/2}+2))$ time. It is not sufficient to represent L simply as a linked list, for then the time spent scanning L will be $O(n)$ per wave, and this scanning time will dominate the time for the rest of the computation. Instead, we represent L as follows. For each integer i in the range $1\leqq i\leqq 2n-1$, we maintain the set $K(i)$ of active vertices with label i that have not been relabeled during the current wave. For each integer j in the range $1\leqq j\leqq\lceil(2n-1)/z\rceil$, we maintain a heap (priority queue) $H(j)$ of the integers i with $K(i)\neq\varnothing$ and $\lceil i/z\rceil=j$. We also maintain a current index j^*, initially equal to $\lceil(2n-1)/z\rceil$.

During a wave, we find the next vertex v to which to apply a *tree-push/relabel* step by decrementing j^* until $H(j^*) \neq \emptyset$, then finding the maximum integer $i \in H(j^*)$, and finally selecting v to be any vertex in $K(i)$. During each *tree push/relabel* step, the sets $K(i)$ and heaps $H(j)$ are updated as vertices become active or inactive and labels change. Each insertion or deletion in a set $K(i)$ takes $O(1)$ time. The heaps $H(j)$ are implemented using any standard heap implementation (see [20]), so that insertion or deletion in a heap $H(j)$ takes $O(\log(z+1))$ time, as does finding the maximum integer in a given heap $H(j)$. (The maximum number of elements in any heap $H(j)$ is z.) A set $K(i)$ or a heap $H(j)$ can change only as the result of a tree push or a relabeling, and the time required for the associated updating of sets and heaps is $O(\log(z+1))$. Thus we obtain the following result:

LEMMA 5.5. *The total time spent maintaining the sets $K(i)$, the heaps $H(j)$, and the index j^* during the tree scaling algorithm is $O(nm \log(z+1))$, if z and l are chosen as in Theorem* 5.4.

Proof. The number of relabelings is $O(nm)$, as is the number of tree pushes by Theorem 5.4. There is $O(n/z)$ time per wave spent decrementing j^*, but this amount of time is included in the $O(nm)$ bound of Theorem 5.4. □

THEOREM 5.6. *The tree scaling algorithm runs in $O(nm \log((n/m)(\log U)^{1/2}+2))$ time if z is chosen equal to $\min\{n, \max\{1, (n^2/m^2)\log U\}$ and l is chosen equal to m/n if $z=1$ or to $(n/m)\log U$ if $z>1$.*

Proof. Immediate from Theorem 5.4 and Lemma 5.5, since $\log(z+1) = O(\log((n/m)(\log U)^{1/2}+2))$. □

6. Bounds in a semilogarithmic computation model. The time bounds derived in §§ 3–5 are all based on the assumption that addition and comparison of integers of magnitude at most U takes $O(1)$ time. If U is huge, this assumption may be unjustified. In this section we explore the consequences to our results of using a semilogarithmic computation model in which each computer word is allowed to hold $O(\log n)$ bits, and any operation involving $O(1)$ words takes $O(1)$ time. In this model, all the elementary graph and list manipulation operations needed by our algorithms take $O(1)$ time each, but adding or comparing two capacity or flow values can take $O(\log_n U)$ time if an exact answer is required. Thus a straightforward translation of our results into the semilogarithmic model increases each of the time bounds by a factor of $O(\log_n U)$.

By suitably modifying the algorithms, however, it is possible to reduce the extra time each of our algorithms requires in the semilogarithmic model to an additive term of $O(m \log_n U)$. Thus, for example, the running time of the tree scaling algorithm becomes $O(nm \log((n/m)(\log U)^{1/2}+2) + m \log_n U)$. Note that the time needed to read all the arc capacities is $\Theta(m \log_n U)$ in this model. Thus, as U grows exponentially large, the total running time becomes linear in the size of the input.

The general approach is to approximately solve a sequence of $O(\log_n U)$ closer-and-closer approximations to the original problem. Each approximation uses as a starting point the approximate solution to the previous problem. Solving each problem requires manipulation of integers of only $O(\log n)$ bits.

Making this approach work involves a number of messy technical details. Since the result is mainly of theoretical interest, we shall merely sketch the ideas involved in modifying the algorithms of §§ 4 and 5. We assume (without loss of generality) that n is a power of two and that U is a power of n. For $k = 1, 2, \cdots, \log_n U$, we define *problem k* to be the maximum flow problem on the graph G with arc capacities $u_k(v, w)$ defined by $u_k(v, w) = \lceil u(v, w)/\delta_k \rceil \delta_k$, where $\delta_k = U/n^k$. Observe that for $k = \log_n U$,

$\delta_k = 1$; thus, problem $\log_n U$ is the original problem. Also, for a general value of k, all capacities in problem k are divisible by δ_k; and, for every arc (v, w), $0 \leqq u_{k-1}(v, w) - u_k(v, w) \leqq \delta_{k-1}$.

We solve problem k for $k = 1, 2, \cdots, \log_n U - 1$ approximately and then solve problem $\log_n U$ exactly. Throughout the computation we maintain a preflow f and a valid labeling d for the current problem. Preflow f satisfies the following constraint:

$$(5) \qquad e(v) \geqq \min \left\{ \sum_{f(u,v)>0} f(u, v), n(\delta_k - 1) \right\} \forall v \neq s \text{ (flow holdback constraint).}$$

(In inequality (5), the sum over $f(u, v)$ is taken to be zero if $f(u, v) \leqq 0$ for all arcs (u, v).)

For a vertex v, the *available excess* at v, which is the amount actually allowed to be pushed from v, is $a(v) = \max\{0, e(v) - n(\delta_k - 1)\}$. Throughout the computation, the available excesses are used in place of the actual excesses to determine when vertices are active and how much flow can be pushed.

The flow holdbacks are needed to maintain the preflow property when converting a solution for one problem into a good initital preflow for the next problem. For the first problem, the preflow f and valid labeling d are initialized exactly as in § 2 (using the arc capacities $u_1(v, w)$). Once a solution to problem $k-1$ is computed, it is converted into an initial preflow for problem k by replacing the current preflow f by the preflow f' defined by $f'(v, w) = \min\{f(v, w), u_k(v, w)\}$. Since $0 \leqq u_{k-1}(v, w) - u_k(v, w) \leqq \delta_{k-1}$, f' is obtained from f by decreasing the flow on each arc (v, w) by an amount between zero and $\min\{f(v, w), \delta_{k-1}\}$ (inclusive). Since $\delta_{k-1} = n\delta_k$, the validity of the flow holdback constraint for f in problem $k-1$ implies the validity of the flow holdback constraint for f' in problem k. The flow holdback constraint implies that f' is a preflow, since f' satisfies the capacity constraint by construction. Since every arc saturated by f in problem $k-1$ remains saturated by f' in problem k, d is a valid labeling for f' in problem k.

For $1 \leqq k < \log_n U$, a preflow f and valid labeling d constitute a solution to problem k if $a(v) \leqq 2n(\delta_k - 1)$ for every active vertex v. Thus a solution to problem $\log_n U$ gives a maximum flow in the original network. After the initialization of the preflow for problem k, every active vertex has an excess of at most $4n^2\delta_k$, of which $3n^2\delta_k = 3n\delta_{k-1}$ comes from the excess on v in problem $k-1$ (since $a(v) \leqq 2n(\delta_{k-1} - 1)$), and $n^2\delta_k = n\delta_{k-1}$ comes from the changes made to f in initializing the preflow for problem k. (These are overestimates.) The initial value of Δ (the bound on maximum available excess) for problem k is $4n^2\delta_k$. Problem k is solved by performing a number of phases, continuing until the maximum available excess is at most $2n(\delta_k - 1)$, which happens by the time Δ is reduced to $n\delta_k$ (or to $\frac{1}{2}$ if $k = \log_n U$). If $k < \log_n U$, the number of phases needed is at most $\log n + 2$; if $k = \log_n U$, the number of phases needed is at most $2 \log n + 3$. Thus each problem requires $O(\log n)$ phases, and the total number of phases over all subproblems is $O(\log n \log_n U) = O(\log U)$.

During the solution of problem k, the preflow f is maintained so that $f(v, w)$ is a multiple of δ_k for each arc (v, w). (This happens automatically, since Δ, the initial flow values, and all capacities are multiples of δ_k.) Thus the flow values and capacities can be represented in units of δ_k. Furthermore, the flow values and capacities are not represented explicitly, but rather as differences from the initial flow values. To be more precise: let f_{k-1} be the final preflow for problem $k-1$; let $f_0 = 0$. Then the current preflow f is represented by the value $f(v, w) - f_{k-1}(v, w)$ for each arc (v, w). The capacity function u_k is represented by the value $\min\{u_k(v, w) - f_{k-1}(v, w), 9n^5\delta_k\}$ for each arc (v, w). We claim that the amount by which the flow on an arc can change during the

solution of problem k and subsequent problems is at most $8n^5\delta_k$. This claim implies that if $u_k(v, w) - f_k(v, w) > 9n^5\delta_k$, the extra residual capacity (beyond $9n^5\delta_k$) on arc (v, w) will never be used and can be ignored. To prove the claim, we note that between relabelings at most $n\Delta \leq 4n^3\delta_k$ units of flow can be moved through any given arc, and there are at most $2n^2$ relabelings over the entire algorithm. This accounts for $8n^5\delta_k$ units of change. The change on an arc due to modifying f between subproblems is at most $\sum_{j=k-1}^{\log_n U} \delta_j \leq 2\delta_{k-1} = 2n\delta_k$. Thus the claim is true.

Storing f and u_k in such a difference form, in units of δ_k, allows the algorithm to manipulate numbers consisting of only $O(\log n)$ bits, and all necessary additions and comparisons of flows and capacities take $O(1)$ time. Constructing the final preflow is merely a matter of adding together the successive flow differences $f_1 - f_0, f_2 - f_1, \cdots$ computed when solving successive problems. By adding these differences from right to left, the time per addition can be made $O(m)$, and the total time is $O(m \log_n U)$. This is also the total time required for initializing the preflow for each problem ($O(m)$ time per problem).

If the wave scaling algorithm is used, the bounds in § 4 remain valid almost without modification and apply to the calculations involved in solving all $\log_n U$ problems. Specifically, the number of relabelings is $O(n^2)$, the number of saturating pushes is $O(nm)$, the number of nonsaturating pushes is $O(n^2 + (n^2/l) \log U)$ plus $O(n)$ per wave, and the number of waves is $O(\min\{nl + \log U, n^2 + \log_n U\})$. The bound on waves is the only one that changes, in that n^2 becomes $n^2 + \log_n U$; this occurs because a wave without relabelings terminates only the solution of the current problem and not the entire algorithm. Choosing $l = (\log U)^{1/2}$ gives an overall bound of $O(nm + n^2(\log U)^{1/2} + m \log_n U)$ time.

Use of the tree scaling algorithm of § 5 results in a time bound of $O(nm \log((n/m)(\log U)^{1/2} + 2) + m \log_n U)$. To obtain this bound, we must verify that the extra time required for initializing the sets $K(i)$, the heaps $H(j)$, and the dynamic trees for each problem, and the extra time for tree pushes associated with vertices that become active at the beginning of a new problem, is $O(m)$ per problem.

We need two extra facts about the dynamic tree data structure. By traversing the entire data structure, the flow values for all the dynamic tree arcs can be computed in $O(n)$ time. Furthermore, a new set of flow values for all these arcs can be installed in the data structure in $O(n)$ time. These facts can easily be verified by checking the description of the data structure [19], [20].

When the flow f is modified at the beginning of a new problem, new vertices with available excess are created, possibly as many as $n - 2$. Handling these newly active vertices is the hardest part of the initialization of the new problem. To begin a new problem, the current flow on all dynamic tree arcs is computed explicitly. (This takes $O(n)$ time as noted above.) Then the flow and capacities are modified to their initial values for the new problem. The bound on maximum available excess is set equal to $4n^3\delta_k$. (This is n times the value proposed earlier in this section and is greater than the total excess on all active nodes.)

Next, the vertices of G are scanned in topological order with respect to the set of eligible arcs. A vertex v is scanned by applying to it a *push/relabel* step of the kind defined in § 2. Such a step either relabels v or reduces its available excess to zero. After all vertices have been scanned, all the available excess is on vertices that have been relabeled during the scanning. The current flow values for dynamic tree arcs are reinstated in the dynamic tree data structure, and every dynamic tree arc that has been saturated during the scanning or that has had one of its end vertices relabeled is cut. Finally, the sets $K(i)$ and heaps $H(j)$ are reinitialized, which takes $O(n)$ time.

(A heap can be initialized in linear time [20].) Then the tree scaling algorithm is begun.

Because the starting value of Δ for each problem is increased, the number of phases increases, but only by $\log n$ per subproblem, or $\log U$ overall. The time to carry out all the scanning in the initialization is $O(m)$. The time to initialize the dynamic tree data structure is $O(m)$ plus $O(\log(z+1))$ per cut; the time for a cut can be charged against the corresponding arc saturation or vertex relabeling. The time to initialize the sets $K(i)$ and heaps $K(j)$ is $O(n)$. At the beginning of a subproblem, the only active vertices are those that have been relabeled during the initialization, and the timing analysis for the tree scaling algorithm is virtually the same as the analysis in § 5. Thus we obtain a total time bound of $O(nm \log((n/m)(\log U)^{1/2}+2)+m \log_n U)$.

The techniques discussed above can also be applied to the maximum flow algorithm of Goldberg and Tarjan [10], resulting in a time bound in the semilogarithmic model of $O(nm \log(n^2/m)+m \log_n U)$. The details are straightforward.

7. Remarks. The algorithms of §§ 3–4 not only have good theoretical time bounds, but they may be very efficient in practice. We hope to conduct experiments to determine whether either of these algorithms is competitive with previously known methods. The tree scaling algorithm of § 5 is perhaps mainly of theoretical interest, although for huge networks there is some chance that the use of dynamic trees may be practical.

The two obvious open theoretical questions are (1) whether further improvement in the time bound for the maximum flow problem is possible and (2) whether our ideas extend to the minimum cost flow problem. It is not unreasonable to hope for a bound of $O(nm)$ for the maximum flow problem; note that the bounds of Theorems 4.3 and 5.6 are $O(nm)$ for graphs that are not too sparse and whose arc capacities are not too large, i.e., $(\log U)^{1/2} = O(m/n)$. Obtaining a bound better than $O(nm)$ would seem to require major new ideas.

As a partial answer to the second question, we have been able to obtain a time bound of $O(nm \log\log U \log(nC))$ for the minimum cost flow problem, where C is the maximum arc cost, assuming all arc costs are integral [2]. This result uses some ideas in the present paper and some additional ones [11].

As a final remark, we note that in certain cases the bounds of our algorithms can be improved by substituting a smaller value for U. To be precise, suppose that f is a flow of value U_1 obtained by some fast heuristic and that U_2 is an upper bound on the maximum flow value, also obtained by some fast heuristic. Then all of our bounds can be improved by substituting $U^* = \max\{4, \min\{U, (U_2 - U_1)/n\}\}$ for U.

To see this, we first construct the residual network for the flow f. This network has the same graph as the original network, with the capacity of each arc (v, w) equal to $u_f(v, w)$. A flow of value x in the residual network corresponds to a flow of value $U_1 + x$ in the original network and vice-versa. It thus suffices to find a maximum flow in the residual network. For this network, $U_2 - U_1$ is an upper bound on the maximum flow value.

We now add to the network a new source s_0, new vertices $v_1, \cdots, v_n$, and arcs (s_0, v_i), (v_i, s), (v_i, s_0), and (s, v_i) for $1 \leqq i \leqq n$. We define $u(s_0, v_i) = u(v_i, s) = \lceil U^* \rceil$, $u(v_i, s_0) = u(s, v_i) = 0$ for $1 \leqq i \leqq n$. This augmented network has the same maximum flow value as the residual network. Then we run one of our excess-scaling algorithms on the augmented network, choosing as the original excess bound Δ the smallest possible allowed value exceeding U^*.

As an application of this result, we can take $U_1 = 0$ and $U_2 = \sum_{v \in V} u(s, v)$, giving a value for U^* of $\max\{4, \sum_{v \in V} u(s, v)/n\}$.

Acknowledgments. We thank Andrew Goldberg for inspiring ideas and stimulating conversations.

REFERENCES

[1] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Oper. Res., to appear.

[2] R. K. AHUJA, A. V. GOLDBERG, J. B. ORLIN, AND R. E. TARJAN, *Finding minimum-cost flows by double scaling*, Math. Prog., submitted.

[3] J. CHERIYAN AND S. N. MAHESHWARI, *Analysis of preflow push algorithms for maximum network flow*, Dept. of Computer Science and Engineering, Indian Institute of Technology, New Delhi, India, 1987.

[4] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[5] S. EVEN, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979.

[6] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[7] H. N. GABOW, *Scaling algorithms for network problems*, J. Comput. System Sci., 31 (1985), pp. 148–168.

[8] A. V. GOLDBERG, *A New Max-Flow Algorithm*, Tech. Report MIT/LCS/TM-291, Laboratory for Computer Science, Mass. Inst. of Technology, Cambridge, MA, 1985.

[9] ———, *Efficient Graph Algorithms for Sequential and Parallel Computers*, Ph.D. thesis, Mass. Inst. of Technology, Cambridge, MA, 1987.

[10] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, in Proc. 18th ACM Symp. on Theory of Computing, 1986, pp. 136–146; J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[11] ———, *Finding minimum-cost circulations by successive approximation*, Math. Oper. Res., to appear.

[12] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[13] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, NY, 1976.

[14] V. M. MALHOTRA, M. PRAMODH KUMAR, AND S. N. MAHESHWARI, *An $O(|V|^3)$ algorithm for finding maximum flows in networks*, Inform. Process. Lett., 7 (1978), pp. 277–278.

[15] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[16] Y. SHILOACH AND U. VISHKIN, *An $O(n^2 \log n)$ parallel max-flow algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[17] D. D. SLEATOR, *An $O(nm \log n)$ algorithm for maximum network flow*, Tech. Report STAN-CS-80-831, Computer Science Dept., Stanford University, Stanford, CA, 1980.

[18] D. D. SLEATOR AND R. E. TARJAN, A data structure for dynamic trees, J. Comput. System Sci., 26 (1983), pp. 362–391.

[19] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[20] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[21] ———, *A simple version of Karzanov's blocking flow algorithm*, Oper. Res. Lett., 2 (1984), pp. 265–268.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 955-976, October 1989

008

VERY FAST PARALLEL POLYNOMIAL ARITHMETIC*

WAYNE EBERLY†

Abstract. Parallel algorithms for polynomial arithmetic—multiplication of n polynomials of degree m, polynomial division with remainder, and polynomial interpolation—are presented. These algorithms can be implemented using polynomial time constructible families of Boolean circuits of polynomial size and optimal order depth, or log space constructible families of polynomial size and near optimal depth, for computations over $\mathbb{Z}$, $\mathbb{Q}$, finite fields, and several other domains. Arithmetic circuits of polynomial size and optimal order depth are obtained for polynomial arithmetic over arbitrary fields.

Key words. polynomial arithmetic, iterated product of polynomials, interpolation, polynomial division, parallel algorithms, circuit depth

AMS(MOS) subject classifications. 68Q40, 12E20, 68Q20

Introduction. A variety of efficient parallel algorithms have appeared in the recent literature. Computation of the product of n polynomials of degree m, polynomial division with remainder, and polynomial interpolation have received much attention, both as the main subject of research (see Reif [17] in particular) and as components in algorithms for related problems, including exponentiation and the factorization of polynomials over finite fields (see, for example, Fich and Tompa [10] and von zur Gathen [13]). Parallel algorithms whose running times are within a constant factor of optimal have been obtained for these problems, for arithmetic computations over fields supporting discrete Fourier transforms (see Bini [2] and Reif [17]). Efficient parallel algorithms for Boolean computations have been obtained for related integer problems, including the computation of the product of n k-bit integers, and integer division with remainder (see Reif [17] and Beame, Cook, and Hoover [1]).

In this paper, we generalize these results. We obtain (log space) NC^1-reductions from the computations for (integer) polynomials described above to the "Iterated Integer Product" problem—one of the integer problems discussed by Reif and by Beame, Cook, and Hoover. As a result, we obtain polynomial time constructible Boolean circuits of logarithmic depth and polynomial size, or log space constructible Boolean circuits of slightly larger depth and polynomial size, for all of these (integer) problems. Generalizing these still further, we obtain small depth Boolean circuits for polynomial arithmetic over finite fields and number fields. We generalize the arithmetic (rather than Boolean) results of Reif to show that the problems in polynomial arithmetic mentioned above can be solved, for polynomials over an arbitrary ground field F, using arithmetic circuits over F of logarithmic depth and polynomial size.

We first discuss Boolean computations. The model used to discuss these computations is a uniform family of Boolean circuits; algorithms are considered to be efficient if they can be implemented using a uniform family of circuits of size polynomial in N, and of depth polynomial in $\log N$, for input size N. We are interested in algorithms that are optimal (or near optimal)—algorithms for which the corresponding circuits

* Received by the editors April 7, 1986; accepted for publication (in revised form) December 14, 1988. This paper contains results from the author's M.Sc. Thesis ([9]). A preliminary version of this paper has appeared in Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science, Springer Island, Florida, 1984, pp. 21-30.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4. Present address, Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4.

have depth $O(\log N)$ (or depth $O(\log^{1+\varepsilon} N)$ for small constants ε). Several definitions of circuit uniformity have been presented; since some algorithms can be implemented by "uniform" circuit families under some definitions of uniformity but (possibly) not for others, the complexity of a problem may depend on the definition used. Ruzzo [18] presents several commonly used definitions of uniformity and discusses relationships between the parallel complexity classes defined using them. Cook [8] gives an overview of parallel Boolean computation by uniform circuit families, including definitions of the model, complexity classes, and a survey of problems having efficient parallel solutions.

To discuss computations for polynomials with integer or rational coefficients, we must begin with algorithms for integer arithmetic—addition, multiplication, iterated product (multiplication of n integers, each of length n), and division with remainder of integers of length n. The sum or product of two n-bit integers can be computed by uniform circuit families using depth $O(\log n)$ and polynomial size (Savage [19], Borodin, Cook, and Pippenger [6]). Applying these results, it is not difficult to obtain uniform circuit families with depth $O(\log^2 n)$ and polynomial size for iterated product and division with remainder. Reif [17] improved this result, obtaining log space constructible families of circuits with depth $O(\log n \log\log n)$ and polynomial size for iterated integer product and integer division with remainder. Beame, Cook, and Hoover [1] get smaller depth using a weaker definition of circuit uniformity: they obtain polynomial time constructible circuit families of depth within a constant factor of optimal ($O(\log n)$) and polynomial size for these problems.

Efficient arithmetic circuits for polynomial arithmetic have been obtained (and will be discussed later). Reasonably efficient Boolean circuits for arithmetic for polynomials with integer coefficients can be obtained if we replace each field operation (indicated by a node) in such an arithmetic circuit by a Boolean circuit implementing that operation (over $\mathbb{Q}$). However, by doing this we increase the depth of the original circuit by a small multiplicative factor; we will not obtain optimal order depth Boolean circuits. Circuits with optimal depth (to within a constant factor) have been obtained: Eberly [9] presents such circuits for computation of the iterated product of polynomials with integer coefficients, for (integer) polynomial division with remainder, and for (integer) polynomial interpolation. Bini and Pan [5] have subsequently used a different method to obtain similar results for (integer) polynomial division with remainder. Their Boolean circuits for this problem are smaller, and hence more efficient, than ours; the circuit depths obtained are the same to within a constant factor. The same technique is used in each case: integer computations are used to obtain the value of a polynomial at a sufficiently large power of two; this value is then used to recover the coefficients of the desired polynomial using circuits of small depth. We will review the reduction from the problem "Iterated Product of Integer Polynomials" to "Iterated Integer Product," and the reductions from other computations for integer polynomials to the latter problem. Applying the results of Reif and Beame, Cook, and Hoover for the integer problem, we conclude that small depth circuits exist for all these problems. We obtain uniform circuit families of the same order depth for arithmetic for polynomials with coefficients in finite fields and in number fields.

We next consider arithmetic computations. As stated above, efficient arithmetic circuits have been obtained for polynomial arithmetic over many fields. Bini [2] presents an optimal order depth algorithm for inversion of a triangular Toeplitz matrix; as indicated by Bini and Pan [3]–[5], Bini's algorithm can be applied to compute the quotient and remainder of polynomials over a large class of fields. Reif [17] presents optimal order depth circuits for computation of the iterated product of polynomials,

interpolation, evaluation of elementary symmetric polynomials, polynomial division with remainder, and for some Taylor series computations. Reif, Bini, and Bini and Pan all assume that the ground field F supports a discrete Fourier transform: that is, F includes nth primitive roots of unity for arbitrarily large n. The assumption is not necessary: Eberly [9] modifies the algorithm presented by Reif to obtain circuits for polynomial arithmetic over arbitrary fields, while Bini and Pan [4] modify Bini's algorithm to obtain such circuits for polynomial division with remainder. We discuss the generalization of Reif's algorithm to computations over arbitrary fields.

As we note above, the definition of uniformity used for Boolean circuits affects the results obtained for these problems. The importance of "uniformity" for arithmetic circuits is less clear—in part, because this has not received much attention. Von zur Gathen [14] presents a survey of results for parallel arithmetic computation, and proposes several definitions of uniformity for arithmetic circuits. We consider two of these definitions. The first is quite generous; we show that our algorithms can be implemented using "uniform" families of circuits when this definition is used. The second definition is much more restrictive; using this definition, we obtain uniform families of arithmetic-Boolean circuits of near optimal depth for these problems, and a reduction to the "Boolean" problem "Iterated Integer Product."

In § 1 we introduce the problems in polynomial arithmetic to be discussed, and describe the representations of field elements to be used for Boolean computations. In §§ 2 and 3 we present results for Boolean computations: for polynomials with integer coefficients in § 2, and for more general domains in § 3. We discuss arithmetic computations in the next sections. "Nonuniform" computations over arbitrary fields are discussed in § 4. We discuss definitions of "uniformity" for arithmetic circuits, and apply them to our algorithms, in § 5. Section 6 includes extensions and open problems.

1. Definitions of problems. We begin with the definition of the "Iterated Integer Product" problem—that to which all others will be reduced. We continue with definitions of problems in polynomial arithmetic over arbitrary domains. These definitions are sufficient for discussions of arithmetic computations. To discuss Boolean computations, however, we must consider "Boolean" representations of elements of various domains. We conclude with a discussion of these representations.

Our "basic" problem for Boolean computations is the following.

ITERATED INTEGER PRODUCT.
Input. Binary representations of integers $n, k > 0$, and of integers $a_1, a_2, \cdots, a_n$ such that $|a_i| < 2^k$ for all i.
Output. Binary representation of the product $\prod_{i=1}^{n} a_i$.

Next we define the problems "Iterated Product of Polynomials," "Polynomial Interpolation," "Evaluation of Elementary Symmetric Polynomials," and "Polynomial Division with Remainder" as computational problems for polynomials with coefficients in a commutative integral domain D. The definitions are made as general as possible to avoid redefining the problems for various domains later.

ITERATED PRODUCT OF POLYNOMIALS (D).
Input. Binary representations of integers $n > 0$, and $m \geqq 0$.
Coefficients $a_{ij} \in D$ (for $0 \leqq j \leqq m$) of polynomials

$$f_i = a_{i0} + a_{i1}x + \cdots + a_{im}x^m \in D[x] \quad \text{for } 1 \leqq i \leqq n.$$

Output. Coefficients $b_0, b_1, \cdots, b_{mn} \in D$ of the product

$$g = b_0 + b_1 x + \cdots + b_{mn} x^{mn} = \prod_{i=1}^{n} f_i \in D[x].$$

POLYNOMIAL INTERPOLATION (D).
Input. Binary representation of an integer $n > 0$. Distinct values $a_1, a_2, \cdots, a_n \in D$, and values $b_1, b_2, \cdots, b_n \in D$ (not necessarily distinct).
Output. Values $d = \prod_{1 \leqq j < i \leqq n} (a_i - a_j) \in D$ and $c_0, c_1, \cdots, c_n \in D$ such that

$$f = c_0 + c_1 x + \cdots + c_{n-1} x^{n-1} \in D[x]$$

is the (unique) interpolating polynomial such that $f(a_i) = db_i$ for $1 \leqq i \leqq n$.

EVALUATION OF ELEMENTARY SYMMETRIC POLYNOMIALS (D).
Input. Binary representations of integers $n \geqq m \geqq 0$. Values $a_1, a_2, \cdots, a_n \in D$.
Output.

$$\alpha_{nm}(a_1, a_2, \cdots, a_n) = \sum_{\substack{J \subseteq \{a_1, a_2, \cdots, a_n\} \\ |J| = m}} \prod_{j \in J} a_j.$$

DIVISION WITH REMAINDER OF POLYNOMIALS (D).
Input. Binary representations of integers $n \geqq m \geqq 0$.
Values $a_0, a_1, \cdots, a_n \in D$ and $b_0, b_1, \cdots, b_m \in D$ such that $b_m \neq 0$.
Output. Values $d = b_m^{n-m+1} \in D$ and $q_0, q_1, \cdots, q_{n-m}, r_0, r_1, \cdots, r_{m-1} \in D$ such that $df = qg + r$ for the polynomials

$$f = a_0 + a_1 x + \cdots + a_n x^n \in D[x], \qquad g = b_0 + b_1 x + \cdots + b_m x^m \in D[x],$$

$$q = q_0 + q_1 x + \cdots + q_{n-m} x^{n-m} \in D[x], \qquad r = r_0 + r_1 x + \cdots + r_{m-1} x^{m-1} \in D[x].$$

We have actually defined "pseudodivision" of polynomials instead of "division," because the former problem is well defined over arbitrary commutative integral domains, while the latter problem is not, as follows: Suppose $f, g \in D[x]$ such that the degree of f is greater than the degree of g, and that the leading coefficient of g is not a unit in D. Then the coefficients of the quotient q and remainder r (such that $f = qg + r$ and r has degree less than that of g) will lie in the quotient field of D, but not in D itself. We have given a nonstandard definition of the problem "Polynomial Interpolation" for the same reason. Note that the "standard" outputs for these problems—coefficients of the quotient and remainder for "Division," and of the interpolating polynomial for "Interpolation"—can be computed efficiently from the outputs of our generalized problems if the domain D is a field and the operation "Division over D" is inexpensive.

We have also included binary representations of integer parameters as inputs for our "arithmetic" problems. These actually specify the size of the input; if desired, they can be left out or, for models allowing elements of the domain D as the only inputs, they can be represented as strings of 0's and 1's (where 0 and 1 represent the arithmetic and multiplicative identify of D, respectively).

These definitions are sufficient for discussions of arithmetic computations. To define Boolean problems, we must describe Boolean representations of elements of the domain D. We also add extra inputs, describing the length of the values used as input or the structure of the domain D. We conclude this section with a discussion of the Boolean representations and additional parameters for the domains to be considered.

$D = \mathbb{Z}$. We use binary representations of integers, with an additional sign bit. We add an additional parameter k, identifying the length of the integers used as coefficients for our input polynomials: each input coefficient a has absolute value less than 2^k.

$D = \mathbb{F}_p$. Each element of the field $\mathbb{F}_p = \mathbb{Z}/(p)$ corresponds to an integer between zero and $p-1$. We use the binary representation of this integer to represent the element of $\mathbb{F}_p$. We add the binary representation of the prime p as an additional parameter.

$D = \mathbb{Q}$. We represent a rational number a by binary representations of an integer numerator and denominator α and β such that $a = \alpha/\beta$ and $\beta > 0$. We do not require that α and β be relatively prime. We add an additional parameter k, identifying the maximum length of the integer numerators and denominators given as input.

$D = \mathbb{Z}[y]$. We use a dense representation of polynomials: a polynomial $\gamma_0 + \gamma_1 y + \cdots + \gamma_k y^k$ (with $\gamma_i \in \mathbb{Z}$ for $0 \leqq i \leqq k$) is represented using binary representations of each of the coefficients $\gamma_0, \gamma_1, \cdots, \gamma_k$. We add two additional parameters, k_1 and k_2. The first is the maximum length of any of the integer coefficients of the elements of D included in the input; the second is the maximum degree (in the indeterminate y) of any of these elements of D.

$D = \mathbb{F}_p[y]$. We use a dense representation of polynomials, as above. We add as additional parameters the prime p and the maximum degree in y of any of the elements of D given as input.

$D = \mathbb{Q}[y]$. Again, we use a dense representation for polynomials. A polynomial

$$\frac{\alpha_0}{\beta_0} + \frac{\alpha_1}{\beta_1} y + \cdots + \frac{\alpha_k}{\beta_k} y^k,$$

with $\alpha_i, \beta_i \in \mathbb{Z}$, and $\beta_i > 0$ for $0 \leqq i \leqq k$, is represented using binary representations of numerators and denominators $\alpha_0, \beta_0, \alpha_1, \beta_1, \cdots, \alpha_k, \beta_k$. As above, we add two parameters: the maximum length of any of the integer numerators and denominators in the input, and the maximum degree (in y) of any of the elements of D used as input.

$D = \mathbb{F}_{p^k}$. We use the fact that $\mathbb{F}_{p^k} \cong \mathbb{F}_p[y]/(\phi)$ for a monic irreducible polynomial $\phi \in \mathbb{F}_p[y]$ with degree k. To each element of $\mathbb{F}_{p^k}$ there corresponds a unique polynomial in $\mathbb{F}_p[y]$ with degree less than k; we use a representation of this polynomial (as described above) to represent the field element. We add as additional parameters binary representations of p and k, and the coefficients of the polynomial $\phi \in \mathbb{F}_p[y]$.

D is a finite algebraic extension of $\mathbb{Q}$. We use the fact that each of these extensions is isomorphic to $\mathbb{Q}[y]/(\phi)$, for some polynomial $\phi \in \mathbb{Z}[y]$ that is irreducible in $\mathbb{Q}[y]$. Each element of the field corresponds to a unique polynomial in $\mathbb{Q}[y]$ with degree less than that of ϕ; we use the representation of the polynomial (described above) to represent the field element. We use as additional parameters the additional parameters described for the domain $\mathbb{Q}[y]$, as well as the degree and coefficients of the polynomial ϕ.

For the sake of completeness, we note here that we will also need to describe the representation of finite algebraic extensions of finite fields to discuss some algorithms for our computational problems. We will state these details in §§ 4 and 5, where they are applied.

2. Boolean computations for integer polynomials. We begin by reducing the problem "Iterated Product of Polynomials ($\mathbb{Z}$)" to "Iterated Integer Product." We use this reduction, and techniques that previously have been used to reduce arithmetic problems to "Iterated Product of Polynomials (D)," to show that the Boolean problems "Polynomial Interpolation ($\mathbb{Z}$)," "Evaluation of Elementary Symmetric Polynomials ($\mathbb{Z}$)," and "Division with Remainder of Polynomials ($\mathbb{Z}$)" can also be reduced to "Iterated Integer Product."

We use uniform families of Boolean circuits as our model for parallel Boolean computation. We consider "log space" uniformity (and call families of circuits "uniform"), and "polynomial time" uniformity (and use the term "P-uniform"). The reductions we state are "NC^1-reductions," as defined by Cook [8].

Our reduction from "Iterated Product of Polynomials ($\mathbb{Z}$)" to "Iterated Integer Product" is based on the fact that the coefficients of an integer polynomial g can be computed from the integer $g(2^L)$ for sufficiently large L. This idea, "Interpolation via Binary Segmentation," has been applied for the sequential computation of polynomials in a number of settings (see Fischer and Paterson [11], Pan [16], [24, § 4], Schönhage [20]). It has also been used to obtain other NC^1-reductions from computations for integer polynomials to integer computations (Eberly [9], Bini and Pan [5]). Bini and Pan include a short history of the use of this method. While it has been presented elsewhere, the method must be modified (slightly) to be used for parallel reductions. Hence we present it in detail.

Suppose now that the polynomial $g \in \mathbb{Z}[x]$ has degree at most d, so that

$$g = g_0 + g_1 x + \cdots + g_d x^d \quad \text{for } g_0, g_1, \cdots, g_d \in \mathbb{Z}.$$

Suppose also that $|g_i| < 2^l$ for $l \geqq 0$, $l \in \mathbb{Z}$, for all i. Let $L = l + 1$.

LEMMA 2.1. *Let d, l, $L \in \mathbb{Z}$, and $g \in \mathbb{Z}[x]$ be as above, and suppose $h_0, h_1, \cdots, h_d \in \mathbb{Z}$. Then the following are equivalent:*

(i) $|h_i| \leqq 2^l$ *for* $0 \leqq i \leqq d$, *and* $g(2^L) = h_0 + h_1 2^L + \cdots + h_d 2^{Ld}$;

(ii) $g_i = h_i$ *for* $0 \leqq i \leqq d$.

Proof. Clearly (ii)$\Rightarrow$(i). We show that (i)$\Rightarrow$(ii) by induction on d. The case $d = 0$ is trivial. For positive d we note that condition (i) implies

$$(h_0 - g_0) + (h_1 - g_1)2^L + \cdots + (h_{d-1} - g_{d-1})2^{L(d-1)} = (g_d - h_d)2^{Ld}.$$

Since $|h_i - g_i| \leqq 2^{l+1} - 1 = 2^L - 1$ for all i, it is easy to show that the value of the left-hand side has magnitude less than 2^{dL}. Since the value of the right-hand side is divisible by 2^{dL}, it is clear that both values equal zero; hence $g_d = h_d$, and the result follows by the inductive hypothesis. □

We compute the coefficients of g from the value $g(2^L)$ by using the binary representation of this number to find a set of integers satisfying the first condition in this lemma, as shown below.

EXTRACTION OF COEFFICIENT ALGORITHM.

Input. Integers G and L, $L > 0$, and $d \geqq 0$, such that $G = g(2^L)$ for some polynomial $g \in \mathbb{Z}[x]$ with degree at most d and whose coefficients have absolute value less than 2^{L-1}.

Output. Coefficients $g_0, g_1, \cdots, g_d$ of the polynomial g.

(1) Compute the values

$$s = \begin{cases} 1 & \text{if } G \geqq 0, \\ -1 & \text{if } G < 0, \end{cases} \quad \text{and } H = sG.$$

Note that $H \geqq 0$.

For $0 \leqq i \leqq d$ in parallel, perform steps (2)-(4).

(2) Compute $k_i = (H \operatorname{div} 2^{Li}) \bmod 2^L$.
(Note. $H = \sum_{i=0}^{d} k_i 2^{Li}$, $0 \leqq k_j < 2^L$ for all j, and $k_d < 2^{L-1}$.)

(3) Compute

$$c_i = \begin{cases} 0 & \text{if } i = 0, \\ 0 & \text{if } i > 0 \text{ and } 0 \leqq k_{i-1} < 2^{L-1}, \\ 1 & \text{if } i > 0 \text{ and } 2^{L-1} \leqq k_{i-1} < 2^L. \end{cases}$$

Let $c_{d+1} = 0$.

(4) Compute $g_i = s(c_i + k_i - 2^L c_{i+1})$.

LEMMA 2.2. *The algorithm "Extraction of Coefficients" is correct and can be used to compute the coefficients of the polynomial g from the value $G = g(2^L)$, using a (log space) uniform family of Boolean circuits of depth $O(\log dL)$ and size $(dL)^{O(1)}$.*

Proof. It is easy to verify that $|g_i| \leqq 2^{L-1}$ for all i and that $g(2^L) = \sum_{i=0}^{d} g_i 2^{Li}$. Correctness then follows by Lemma 2.1. The depth and size bounds are also obvious, assuming that a binary representation of the integer G is given as input. □

For the problem "Iterated Product of Polynomials ($\mathbb{Z}$)," we consider the polynomial $g = \prod_{i=1}^{n} f_i$, where each polynomial f_i has degree at most m and has coefficients with magnitude less than 2^k, for integers $m \geqq 0$ and $k > 0$. Lemma 2.3 gives us a reasonably small upper bound for the magnitude of the coefficients of g.

LEMMA 2.3. *Let $g = \prod_{i=1}^{n} f_i \in \mathbb{Z}[x]$, for integer polynomials $f_i = \sum_{j=0}^{m} a_{ij} x^j$ with degree at most m and with coefficients a_{ij} with magnitude less than 2^k, for all i. Then $g = \sum_{i=0}^{nm} g_i x^i$ for integers $g_0, g_1, \cdots, g_{nm}$ such that $|g_i| < 2^l$ for $0 \leqq i \leqq nm$, where $l = n(k + \lceil \log_2 (m+1) \rceil)$.*

Proof. This follows immediately from the equation

$$g_i = \sum_{\substack{j_1, j_2, \cdots, j_n \geqq 0 \\ j_1 + j_2 + \cdots + j_n = i}} \prod_{h=1}^{n} a_{hj_h},$$

relating the coefficients of g to those of f, the bound $|a_{ij}| < 2^k$, and the fact that there are fewer than $(m+1)^n$ possible choices of the integers $j_1, j_2, \cdots, j_n$ for each value of i. □

We can now state the algorithm that reduces "Iterated Product of Polynomials ($\mathbb{Z}$)" to "Iterated Integer Product."

ITERATED PRODUCT OF POLYNOMIALS ($\mathbb{Z}$) VIA ITERATED INTEGER PRODUCT ALGORITHM.

Input. Binary representations of integers $n > 0$, $m \geqq 0$ and $k > 0$.
Binary representations of coefficients $a_{ij} \in \mathbb{Z}$ (for $0 \leqq j \leqq m$) of polynomials

$$f_i = a_{i0} + a_{i1} x + \cdots + a_{im} x^m \in \mathbb{Z}[x]$$

such that $|a_{ij}| < 2^k$ for $0 \leqq j \leqq m$, $1 \leqq i \leqq n$.

Output. Binary representations of coefficients $b_0, b_1, \cdots, b_{mn}$ of the product

$$g = b_0 + b_1 x + \cdots + b_{mn} x^{mn} = \prod_{i=1}^{n} f_i \in \mathbb{Z}[x].$$

(1) Compute $L = n(k + \lceil \log_2 (m+1) \rceil) + 1$.

(2) For all i, in parallel, compute

$$F_i = f_i(2^L) = a_{i0} + a_{i1} 2^L + \cdots + a_{im} 2^{Lm}.$$

(3) (*Iterated integer product.*) Compute $G = \prod_{i=1}^{n} F_i$. (Note. $G = g(2^L)$.)

(4) (*Extraction of coefficients.*) Use the value G to compute the coefficients of g.

THEOREM 2.4. *The problem "Iterated Product of Polynomials* $(\mathbb{Z})$*" is* NC^1*-reducible to "Iterated Integer Product."*

Proof. It is sufficient to note that the above algorithm is correct (by Lemmas 2.2 and 2.3) and that steps (1), (2), and (4) can be implemented using depth logarithmic, and size polynomial, in the input size. □

We reduce the remaining integer problems to "Iterated Integer Product" by reducing them to "Iterated Integer Product" and "Iterated Product of Polynomials $(\mathbb{Z})$," and applying the above theorem. It is sufficient to adapt the arithmetic reductions of Reif (see [17, §§ 2.4–2.5]) to obtain division-free (Boolean) algorithms in order to do this; we sketch the resulting methods below.

(i) *Polynomial interpolation.* Given integers $n \geqq 0$ and distinct integers $a_1, a_2, \cdots, a_n \in \mathbb{Z}$, the integer

$$d = \prod_{1 \leqq j < i \leqq n} (a_i - a_j)$$

and the polynomials

$$L_h = \left(\prod_{\substack{i=2 \\ i \neq h}}^{n} \prod_{\substack{j=1 \\ j \neq h}}^{i-1} (a_i - a_j) \right) \cdot \left(\prod_{j=1}^{h-1} (x - a_j) \right) \cdot \left(\prod_{i=h+1}^{n} (a_i - x) \right)$$

are computed using instances of "Iterated Integer Product" and "Iterated Product of Polynomials $(\mathbb{Z})$" in parallel. It is clear that, for $1 \leqq i, j \leqq n$,

$$L_i(a_j) = 0 \quad \text{if } i \neq j \quad \text{and} \quad L_i(a_i) = d.$$

Given integers $b_1, b_2, \cdots, b_n$, we set $f = \sum_{i=1}^{n} b_i L_i$; then $f(a_i) = db_i$ for $1 \leqq i \leqq n$, as desired.

(ii) *Evaluation of elementary symmetric polynomials.* It can be verified by a straightforward expansion that

$$\prod_{i=1}^{n} (x + a_i) = \sum_{m=0}^{n} \sigma_{n,m}(a_1, a_2, \cdots, a_n) x^{n-m}.$$

Thus we can compute $\sigma_{nm}(a_1, a_2, \cdots, a_n)$ by computing the product of the polynomials $(x + a_1), (x + a_2), \cdots, (x + a_n)$ and selecting the coefficient of x^{n-m} in the result.

(iii) *Division with remainder of polynomials.* Given integers $n \geqq m \geqq 0$ and polynomials

$$f = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0 \quad \text{and} \quad g = b_m x^m + b_{m-1} x^{m-1} + \cdots + b_0,$$

with $a_n, a_{n-1}, \cdots, a_0, b_m, b_{m-1}, \cdots, b_0 \in \mathbb{Z}$ and $b_m \neq 0$, we compute the ("pseudo") quotient and remainder obtained by dividing $df = b_m^{n-m+1} f$ by g, by considering the "reversals"

$$F = x^n f\left(\frac{1}{x}\right) = a_n + a_{n-1} x + \cdots + a_0 x^n \quad \text{and} \quad G = x^m g\left(\frac{1}{x}\right) = b_m + b_{m-1} x + \cdots + b_0 x^m.$$

Set $H = b_{m-1} + b_{m-2} x + \cdots + b_0 x^{m-1}$, so that $G = b_m + xH$. Solving instances of "Iterated Product of Polynomials $(\mathbb{Z})$" and "Iterated Integer Product" in parallel, we (first) compute the powers $H, H^2, H^3, \cdots, H^{n-m}$ and $b_m, b_m^2, \cdots, b_m^{n-m}$, and (then) the polynomials $Q_0, Q_1, \cdots, Q_{n-m}$ in $\mathbb{Z}[x]$ with degree at most $n - m$ such that

$$Q_j \equiv b_m^j x^j \sum_{k=0}^{n-m-j} ((-1)^k b_m^{n-m-j-k} x^k H^k) \bmod x^{n-m+1}.$$

Then, $Q_j \cdot G = Q_j \cdot (b_m + xH) \equiv b_m^{n-m+1} x^j = dx^j \bmod x^{n-m+1}$, for $0 \leqq j \leqq n-m$. Setting $Q = a_0 Q_0 + a_1 Q_1 + \cdots + a_{n-m} Q_{n-m} \in \mathbb{Z}[x]$, we find that Q has degree at most $n-m$ (since each Q_j does), and that $dF \equiv d(a_0 + a_1 x + \cdots + a_{n-m} x^{n-m}) \equiv Q \cdot G \bmod x^{n-m+1}$.

Now, setting $q = x^{n-m} Q(1/x) \in \mathbb{Z}[x]$, and $r = df - qg \in \mathbb{Z}[x]$, we find that r has degree less than m and $df = qg + r$: that is, q and r are the (pseudo) quotient and remainder we require.

Note. Reif includes this basic reduction (in a form suitable for computations over fields) as well as a variation that eliminates the need to compute most of the powers H^i and b_m^i (for $1 \leqq i \leqq n-m$) explicitly—producing circuits of smaller size for this reduction. See § 2.5 of Reif [17] for details.

THEOREM 2.5. *The problems "Polynomial Interpolation ($\mathbb{Z}$)," "Evaluation of Elementary Symmetric Polynomials ($\mathbb{Z}$)," and "Division with Remainder of Polynomials ($\mathbb{Z}$)" are all NC^1-reducible to "Iterated Integer Product."*

Proof. Each of the algorithms sketched above can be implemented using (log space) uniform families of Boolean circuits, with oracles for "Iterated Integer Product," of polynomial size and logarithmic depth. □

Now we are ready to apply the results of Reif [17] and Beame, Cook, and Hoover [1] stated below.

FACT 2.6 (Reif [17]). *The problem "Iterated Integer Product" can be solved using a* (log space) *uniform family of Boolean circuits of depth $O(\log N \log\log N)$ and size $N^{O(1)}$, for input size N.*

FACT 2.7 (Beame, Cook, and Hoover [1]). *The problem "Iterated Integer Product" can be solved using a* P*-uniform family of Boolean circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

Combining the results of Theorems 2.4–2.5 and Facts 2.6–2.7, we obtain the following.

COROLLARY 2.8. *Each of the problems stated in Theorems* 2.4 *and* 2.5 *can be solved using a uniform family of Boolean circuits of depth $O(\log N \log\log N)$ and size $N^{O(1)}$, or a* P*-uniform family of Boolean circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

The results in this section were first presented in Eberly [9]. Bini and Pan [5] have subsequently shown that the problem "Division with Remainder of Polynomials ($\mathbb{Z}$)" is reducible to the problem "Integer Division with Remainder"—also using "Interpolation via Binary Segmentation" to obtain their reduction. Since the integer problems "Integer Division with Remainder" and "Iterated Integer Product" are equivalent with respect to log space reducibility, they also establish the reduction for "Division with Remainder of Polynomials ($\mathbb{Z}$)" stated in Theorem 2.5. As they note, their more direct algorithm produces smaller (hence, more efficient) circuits than those that would be obtained using the reduction stated here; the depths of their (log space) uniform circuits and P-uniform circuits are the same as ours, to within a constant factor.

3. Boolean computations over other domains. The results of § 2 for computations of integer polynomials can be extended to computations of polynomials over more general domains. In this section we consider Boolean computations of polynomials with coefficients in a domain D, for each of the domains discussed in § 1. We assume that elements of these domains are represented as described in that section. We apply the techniques that have been described in § 2 to exhibit NC^1-reductions from computations over these domains to our "basic" problem, "Iterated Integer Product."

THEOREM 3.1. *For each of the domains $D = \mathbb{F}_p$, $\mathbb{Q}$, $\mathbb{Z}[y]$, $\mathbb{F}_p[y]$ $\mathbb{Q}[y]$, $\mathbb{F}_{p^k}$, and $\mathbb{Q}[y]/(\phi)$ (for ϕ irreducible in $\mathbb{Q}[y]$), the problems "Iterated Product of Polynomials*

(D)," "Polynomial Interpolation (D)," "Evaluation of Elementary Symmetric Polynomials (D)," and "Division with Remainder of Polynomials (D)" are all NC^1-reducible to "Iterated Integer Product."

Proof. We first reduce "Iterated Product of Polynomials (*D*)" to "Iterated Integer Product," for each of these domains *D*. We then adapt Reif's reductions (in [17]; used here to prove Theorem 2.5) to reduce the remaining problems to "Iterated Integer Product." We make repeated use of Theorems 2.4 and 2.5, and the fact that "Integer Division with Remainder" is NC^1-reducible to "Iterated Integer Product" (see Beame, Cook, and Hoover [1, Thm. 3.1 and Cor. 6.2.]).

(i) The result is straightforward for the case $D=\mathbb{F}_p$, since the elements of $\mathbb{F}_p$ are represented using integers between zero and $p-1$. We simply treat the input as a set of binary representations of integer polynomials, compute their product, and compute the residue (modulo p) of each coefficient of the result to obtain a representation of the product (in $\mathbb{F}_p[x]$) to be generated.

(ii) For the case $D=\mathbb{Q}$, we note that by computing a small number of products of integers in parallel, we can "rewrite" each input polynomial so that its coefficients have the same integer denominator: that is,

$$f_i=\frac{1}{\beta_i}(\alpha_{i0}+\alpha_{i1}x+\cdots+\alpha_{im}x^m)=\frac{1}{\beta_i}\hat{f}_i$$

for $\beta_i, \alpha_{i0}, \alpha_{i1}, \cdots, \alpha_{im}\in\mathbb{Z}$, $\beta_i>0$, and for $\hat{f}_i\in\mathbb{Z}[x]$, for $1\leqq i\leqq n$. Solving instances of "Iterated Product of Polynomials ($\mathbb{Z}$)" and "Iterated Integer Product," we compute $\beta=\prod_{i=1}^m\beta_i$ and the coefficients of $\hat{g}=\prod_{i=1}^m\hat{f}_i$ separately. We then use the binary representations of these values to obtain a representation of the polynomial $g=\beta^{-1}\hat{g}$.

(iii)-(v) For the case $D=\mathbb{Z}[y]$ we can use either of two simple methods to reduce our problem to "Iterated Product of Polynomials ($\mathbb{Z}$)." The coefficients of the product to be computed are integer polynomials in the indeterminate y; we can use "Interpolation via Binary Segmentation" to compute these coefficients by replacing y by a suitably large power of two, computing the product of the resulting univariate integer polynomials, and then extracting the coefficients (in y) as described in § 2. Alternatively, we can replace y by a suitably large power of x (x^{mn+1} is sufficient)—and again recover our coefficients after computing the product of univariate integer polynomials. We use the latter method to reduce our problems for $D=\mathbb{F}_p[y]$ and $D=\mathbb{Q}[y]$ to the problems for $D=\mathbb{F}_p$ and $D=\mathbb{Q}$, respectively.

(vi) We next consider the case $D=\mathbb{F}_{p^k}$. As stated in § 2, we are given an irreducible polynomial $\phi\in\mathbb{F}_p[y]$ with degree k, such that $\mathbb{F}_{p^k}\cong\mathbb{F}_p[y]/(\phi)$. An instance of "Iterated Product of Polynomials ($\mathbb{F}_{p^k}$)" includes the polynomial ϕ, and polynomials $\hat{f}_1,\hat{f}_2,\cdots,\hat{f}_n\in(\mathbb{F}_p[y])[x]$; the latter polynomials "represent" the polynomials $f_1=(\hat{f}_1 \bmod \phi), f_2=(\hat{f}_2 \bmod \phi),\cdots,f_n=(\hat{f}_n \bmod \phi)$ whose product we wish to compute.

Set $g=\prod_{i=1}^n f_i\in\mathbb{F}_{p^k}[x]$ and $\hat{g}=\prod_{i=1}^n\hat{f}_i\in(\mathbb{F}_p[y])[x]$; then it is clear that $g=(\hat{g} \bmod \phi)$. We compute g in two steps: We first use an instance of "Iterated Product of Polynomials ($\mathbb{F}_p[y]$)" to compute the polynomial $\hat{g}$. We then use instances of "Division with Remainder of Polynomials ($\mathbb{F}_p$)" in parallel, dividing each of the coefficients of $\hat{g}$ (for powers of x) by ϕ to obtain the coefficients of g.

We have already seen that "Iterated Product of Polynomials ($\mathbb{F}_p[y]$)" is NC^1-reducible to "Iterated Integer Product." It is clear that we can solve an instance of "Division with Remainder of Polynomials ($\mathbb{F}_p$)" by performing (pseudo) division of integer polynomials, and then computing residues (modulo p) of the resulting coefficients. Applying Theorem 2.5, and the NC^1-reduction of "Integer Division with Remainder" to "Iterated Integer Product" ([1, Thm. 3.1]), we conclude that "Division

with Remainder of Polynomials ($\mathbb{F}_p$)" is NC^1-reducible to "Iterated Integer Product." Thus "Iterated Product of Polynomials ($\mathbb{F}_{p^k}$)" is NC^1-reducible to "Iterated Integer Product," as well.

(vii) The case $D = \mathbb{Q}[y]/(\phi)$ is analogous to the case $D = \mathbb{F}_{p^k}$: we compute the desired product by solving instances of the problems "Iterated Product of Polynomials ($\mathbb{Q}[y]$)" and "Division with Remainder of Polynomials ($\mathbb{Q}$)." Again, we can reduce the last problem to "Iterated Integer Product" by extracting the denominators of coefficients (as in (ii), above) and performing pseudodivision of integer polynomials.

For each domain D stated above, we use the division-free forms of the reductions to "Iterated Product of Polynomials (D)" given by Reif [17] (and used here already to prove Theorem 2.5) to reduce our (Boolean) problems "Polynomial Interpolation (D)," "Evaluation of Elementary Symmetric Polynomials (D)," and "Division with Remainder of Polynomials (D)" to "Iterated Product of Polynomials (D)" and "Iterated Integer Product," and hence to "Iterated Integer Product" alone. We cannot apply Reif's reductions directly. We leave it to the reader to verify that the versions of the reductions given in § 2 are division-free and correct for arbitrary commutative integral domains. It is also easily checked that these "modified" reductions can be used to produce (Boolean) NC^1-reductions from each of these problems to "Iterated Integer Product" (see Eberly [9] for a more detailed presentation). □

Applying Facts 2.6 and 2.7 again, we obtain Corollary 3.2.

COROLLARY 3.2. *Each of the problems stated in Theorem* 3.1 *can be solved using a uniform family of Boolean circuits of depth* $O(\log N \log\log N)$ *and size* $N^{O(1)}$, *or a* P*-uniform family of Boolean circuits of depth* $O(\log N)$ *and size* $N^{O(1)}$, *for input size* N.

For fields $D = \mathbb{Q}$, $\mathbb{F}_p$, $\mathbb{F}_{p^k}$ and $\mathbb{Q}[y]/(\phi)$, it is clear that we do not need the denominator d included in the output for our division and interpolation problems: polynomial division and interpolation (rather than pseudodivision and our "nonstandard" interpolation problem) are well-defined problems for computations over fields.

Unfortunately, we do not obtain NC^1-reductions from these "standard" problems to "Iterated Integer Product"—for we do not have log depth circuits for division of field elements for fields $\mathbb{F}_p$, $\mathbb{F}_{p^k}$ and $\mathbb{Q}[y]/(\phi)$, using our representations.

For the field $\mathbb{F}_p = \mathbb{Z}/(p)$, and representations we are using, we must compute the binary representation of an integer s such that $su \equiv 1 \pmod p$ if we are to invert the element u. We can do this sequentially by using the extended Euclidean algorithm to compute $\gcd(u, p)$ as well as integers s and t such that $su + tp = \gcd(u, p) = 1$. However, no efficient parallel algorithm for this computation is known.

For the field extensions $\mathbb{F}_{p^k}$ and $\mathbb{Q}[y]/(\phi)$ and the representations we use, we again compute inverses sequentially by using the extended Euclidean algorithm—but for polynomials, rather than integers. This method can be implemented using Boolean circuits of depth $O(\log^2 N)$ and size $N^{O(1)}$ for input size N—using a "redundant notation" (as above) for elements of $\mathbb{F}_p$ or of $\mathbb{Q}$ (see Borodin, von zur Gathen, and Hopcroft [7], von zur Gathen [13]). Recently, the problem "Inversion in $\mathbb{F}_{p^k}$" has been reduced (with respect to log space or NC^1 computations) to "Inversion in $\mathbb{F}_p$" for arbitrary primes p (see Litow and Davida [23] and von zur Gathen [22]). However, this is not sufficient for the problem at hand.

4. Nonuniform arithmetic computations. We discuss results for two models of parallel arithmetic computation. *Arithmetic circuits* use elements of a commutative integral domain D for inputs and constants, and return elements of D as outputs. We execute one arithmetic operation over D (that is, one of $+$, $-$, $\times$, or the selection of an input or constant in D) at each node of the circuit; if D is a field, we also allow

nodes for division ($\div$). *Arithmetic-Boolean circuits* include these operations as well as a Boolean component: we include nodes for selection of Boolean constants (**T**, **F**), operations ($\wedge, \vee, \neg$), *zero tests* (using an input from D and producing a Boolean output), and *selections* (of one of two inputs from D on the basis of a third Boolean input—branching).[1] This second model of arithmetic computation is strictly stronger than the first: Fich and Tompa [10] present a problem that has an efficient parallel solution in the second model, but that has no such solution in the first. We will not define "uniform" families of arithmetic-Boolean circuits in this section; hence we call the circuit families we discuss "nonuniform."

Henceforth we consider polynomial arithmetic over a field (F). We begin with a discussion of computations for infinite fields. Bini [2] and Reif [17] have presented circuits with optimal depth (to within a constant factor) for a problem that is clearly equivalent to "Division with Remainder of Polynomials (F)," and for "Iterated Product of Polynomials (F)," respectively, for fields F containing nth primitive roots of unity for infinitely many n. Reif also notes that a variety of other problems (including those discussed here) are reducible to "Iterated Product of Polynomials (F)."

Reif [17] computes the "Iterated Product of Polynomials (F)" by implementing a discrete Fourier transform as follows. Given polynomials $f_1, f_2, \cdots, f_n$, whose product g is to be computed, he uses a discrete Fourier transform to compute the values of each of the f_i's at a set of roots of unity. He then uses multiplication in F to compute the value of g at each of these roots, and, finally, uses the inverse Fourier transform to recover the coefficients of g from this set of values. We obtain an optimal order depth algorithm for "Iterated Product of Polynomials (F)" that is correct for an arbitrary infinite field F, but that is implemented using circuits of larger size than those obtained by Reif, by replacing the roots of unity used (as evaluation points) in his method by any sufficiently large set of distinct elements of F. The resulting algorithm is stated below.

ITERATED PRODUCT OF POLYNOMIALS VIA EVALUATION-INTERPOLATION ALGORITHM.

Input. Binary representations of integers $n > 0$, and $m \geqq 0$.
Coefficients $a_{ij} \in F$ (for $0 \leqq j \leqq m$) of polynomials

$$f_i = a_{i0} + a_{i1}x + \cdots + a_{im}x^m \in F[x] \quad \text{for } 1 \leqq i \leqq n.$$

Output. Coefficients $b_0, b_1, \cdots, b_{mn} \in F$ of the product

$$g = b_0 + b_1 x + \cdots + b_{mn}x^{mn} = \prod_{i=1}^{n} f_i \in F[x].$$

Constants used. Distinct $\gamma_0, \gamma_1, \cdots, \gamma_{mn} \in F$;
the entries of the *inverse* of the Vandermonde matrix
$V(\gamma_0, \gamma_1, \cdots, \gamma_{mn})$ of order $mn+1$ defined by

$$V(\gamma_0, \gamma_1, \cdots, \gamma_{mn})_{ij} = \gamma_{i-1}^{j-1} \quad \text{for } 1 \leqq i, j \leqq mn+1.$$

(1) For all i and j, $1 \leqq i \leqq n$, $0 \leqq j \leqq mn$, compute

$$f_i(\gamma_j) = \sum_{l=0}^{m} a_{il}\gamma_j^l.$$

(2) For all j, $0 \leqq j \leqq mn$, compute $g(\gamma_j) = \prod_{i=1}^{n} f_i(\gamma_j)$.

[1] Arithmetic-Boolean circuits have been called *arithmetic networks* in some papers.

(3) Use the values computed in step (2) and the entries of $V(\gamma_0, \gamma_1, \cdots, \gamma_{mn})^{-1}$ (supplied as constants) to compute the coefficients $b_0, b_1, \cdots, b_{mn}$ such that

$$\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{mn} \end{bmatrix} = V(\gamma_0, \gamma_1, \cdots, \gamma_{mn})^{-1} \begin{bmatrix} g(\gamma_0) \\ g(\gamma_1) \\ \vdots \\ g(\gamma_{mn}) \end{bmatrix}.$$

Return these coefficients as output.

THEOREM 4.1. *Let F be an infinite field. Then the problem "Iterated Product of Polynomials (F)" can be solved using a (nonuniform) family of arithmetic circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

Proof. It is sufficient to show that the above algorithm is correct and efficient, when the field F is infinite.

It is easily checked that

$$V(\gamma_0, \gamma_1, \cdots, \gamma_{mn}) \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{mn} \end{bmatrix} = \begin{bmatrix} g(\gamma_0) \\ g(\gamma_1) \\ \vdots \\ g(\gamma_{mn}) \end{bmatrix},$$

and that $\det V(\gamma_0, \gamma_1, \cdots, \gamma_{mn}) = \prod_{i=1}^{mn} \prod_{j=0}^{i-1} (\gamma_i - \gamma_j) \neq 0$, so that the matrix $V(\gamma_0, \gamma_1, \cdots, \gamma_{mn})$ is invertible. It follows that the values $b_0, b_1, \cdots, b_{mn}$ returned by this algorithm are the coefficients we want: that is, the algorithm is correct.

The only operations used in the algorithm are the evaluation of polynomials with small degree (in step (1)); multiplication of elements in F (in step (2)); and the multiplication of a matrix by a vector (in step (3)). It is clear, then, that the algorithm can be implemented using arithmetic circuits of the stated depth and size. □

We obtain corresponding results for the problems "Polynomial Interpolation (F)," "Evaluation of Elementary Symmetric Polynomials (F)," and "Division with Remainder of Polynomials (F)," by applying Theorem 4.1 and arithmetic reductions from each of these problems to "Iterated Product of Polynomials (F)." We use the definition of "NC^1_F-reduction" stated by von zur Gathen [14], and state these reductions in a general form suitable for their use in the rest of §§ 4–5.

FACT 4.2 (Reif [17]). *Let F be an arbitrary field. Then each of the problems "Polynomial Interpolation (F)," "Evaluation of Elementary Symmetric Polynomials (F)," and "Division with Remainder of Polynomials (F)" is NC^1_F-reducible to "Iterated Product of Polynomials (F)."*

Proof. The reductions stated by Reif [17, §§ 2.4–2.5] are sufficient. We leave it as an exercise for the reader to verify that these are (log space uniform) "NC^1_F-reductions," as defined in [14]. □

Combining Theorem 4.1 and Fact 4.2, we obtain Corollary 4.3.

COROLLARY 4.3. *Let F be an infinite field. Then each of the problems "Iterated Product of Polynomials (F)," "Polynomial Interpolation (F)," "Evaluation of Elementary Symmetric Polynomials (F)," and "Division with Remainder of Polynomials (F)" can be solved using a (nonuniform) family of arithmetic circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

We now consider computations over the finite field $F = \mathbb{F}_q = \mathbb{F}_{p^k}$, for prime p and positive k. As above, we first consider the problem "Iterated Product of Polynomials (F)." Suppose m and n are as defined in the statement of this problem. If $q \geqq mn+1$ then our algorithm for infinite fields can be used directly, because F includes a suitable

set of constants. Otherwise, we must construct a field extension and perform arithmetic over the extension if we are to use this algorithm. A direct simulation, in which we simulate each operation over the extension individually using arithmetic over F, is not sufficient: the circuit depth is increased by a small multiplicative factor, so circuits of optimal depth (to within a constant factor) are not obtained. In the rest of this section we describe a slightly more complicated simulation that we use to obtain optimal order depth circuits for this problem.

We begin by describing our representation of finite fields $F = \mathbb{F}_q$ and $E = \mathbb{F}_r = \mathbb{F}_{q^l} \supseteq F$ (the extension used by our algorithm). Since all finite fields of size r are isomorphic, $E \cong F[y]/(\phi)$ for every irreducible polynomial $\phi \in F[y]$ with degree l. Hence we could represent elements of E as the residues modulo ϕ of polynomials in $F[y]$ for any such polynomial ϕ. Embedding F in E is trivial: the elements of the smaller field correspond to the residues of constant polynomials (and we convert between representations of inputs and outputs as "elements of F" and "elements of E" using constant depth). Addition and multiplication in E are also straightforward. To multiply elements of E (represented as residues of polynomials) we multiply the corresponding (small degree) polynomials in $F[y]$ and then compute the remainder (resulting from division by ϕ) to obtain a representation of the product. This last step can be implemented using circuits with size polynomial in l and depth $O(\log^2 l)$. (Note that one of our goals is to prove that polynomial division with remainder can be performed using circuits of polynomial size and logarithmic depth. We "charge" depth $\Theta(\log^2 l)$ for the division mentioned here to avoid the use of a result we have yet to prove.) If the residues y^i modulo ϕ are "precomputed," for $0 \leqq i \leqq 2l$, then the depth for subsequent multiplications in E can be reduced to $O(\log l)$. Addition in E requires "coefficient-wise" addition in $F[y]$, and can be implemented using circuits of depth one and size $O(l)$. Division and exponentiation in E are (slightly) more expensive: each can be implemented using polynomial size circuits, with depth $O(\log^2 l)$ for division and depth $O(\log n \log l)$ for computation of the nth power of an element of E. Using these costs, we find that evaluation of a polynomial of degree n with coefficients in F at an element of E can be performed using depth $O(\log n \log l)$ and polynomial size. We can compute the product of n polynomials with degree m and coefficients in $F = \mathbb{F}_q$ by implementing our algorithm over an extension $E = \mathbb{F}_{q^l}$, for $l = \lceil \log_q (mn+1) \rceil = O(\log mn)$, using circuits of polynomial size and depth $O(\log mn \log\log mn)$.

Examining the algorithm "Iterated Product of Polynomials via Evaluation–Interpolation," implemented over a fixed extension E, we see that the only steps using depth $\omega(\log mn)$ are the evaluations of products of $O(mn)$ elements of E. We can perform these computations quickly using a different representation of E. Since the multiplicative subgroup of any finite field is cyclic, there exists some nonzero $\zeta \in E$ such that

$$E = \{0\} \cup \{\zeta^i : 0 \leqq i \leqq |E| - 2 = q^l - 2\}.$$

We can represent a nonzero element α of E by its (discrete) logarithm with respect to base ζ—that is, by the unique integer i such that $0 \leqq i < q^l - 1$ and $\zeta^i = \alpha$. Using this representation, we can compute the product of n nonzero elements by adding their logarithms and dividing this sum by $q^l - 1$ (we know $\zeta^{q^l - 1} = 1$). The remainder of this division is the discrete logarithm of the product. Division in E is also straightforward, since the inverse of ζ^i is $\zeta^{q^l - 1 - i}$ if $i \neq 0$, or ζ^0 otherwise. We use circuits of polynomial size and depth $O(\log n + \log l)$ to multiply n elements of E, and circuits of depth $O(\log l)$ to divide. Unfortunately, it is not easy to add elements of E represented by

discrete logarithms, or to convert between the "standard" representation of $\alpha \in F$ and its representation by a discrete logarithm in E.

As noted above, however, we only need to perform computations over a field extension when F is very small. Furthermore, we can solve our problem using an extension $E \supseteq F$ such that E has size less than $(mn+1)^2$. Under this condition, we can combine the two representations for $E = \mathbb{F}_{q^l}$ by choosing a monic irreducible polynomial $\phi \in F[y]$ with degree l such that the element y modulo ϕ is a generator of the multiplicative subgroup of $E = F[y]/(\phi)$. That is,

$$E = \left\{ \sum_{i=0}^{l-1} c_i y^i : c_0, c_1, \cdots, c_{l-1} \in F \right\} = \{0\} \cup \{ y^i : 0 \leqq i < q^l - 1 \}.$$

We know a suitable polynomial ϕ exists: E must contain some generator of its multiplicative subgroup, and we choose ϕ to be the minimal polynomial (over F) of (any) generator. Since E is small, we can "hardwire" a table, listing the two representations of every nonzero element of E, into our circuit. We use the "small degree polynomial" representation of elements for most computations. To perform multiplications of $m > 2$ elements of E, we perform a "table lookup" to compute discrete logarithms, compute the logarithm of the product, then perform another lookup to obtain the "polynomial" representation of this product. Using this combined representation, we can evaluate polynomials of degree n over E, compute products of m elements of E, and divide elements of E, using circuits of depth $O(\log mn)$ and polynomial size. It can then be easily checked that we can implement our algorithm over a small field extension E using arithmetic-Boolean circuits of polynomial size and depth $O(\log mn)$ (with arithmetic over F at unit cost), as required. We apply Reif's reductions once again to obtain small depth circuits for the other problems in polynomial arithmetic being discussed.

THEOREM 4.4. *Let F be a finite field; then there exist nonuniform families of arithmetic-Boolean circuits of depth $O(\log N)$ and size $N^{O(1)}$ solving each of the problems "Iterated Product of Polynomials (F)," "Polynomial Interpolation (F)," "Evaluation of Elementary Symmetric Polynomials (F), and "Division with Remainder of Polynomials (F)."* □

Finally, we note again that the techniques we describe for computations over infinite fields fail us only when our ground field F is very small—specifically, when $|F| < N^c$, where N is our input size, and c is a small constant. We leave it to the reader to verify that under this condition, a "zero test" in F can be simulated by the evaluation of a fixed polynomial of small degree; "selection gates," $\wedge$, $\vee$ and $\neg$ gates can also be simulated by small arithmetic circuits. Hence, we can strengthen Theorem 4.4 by replacing "arithmetic-Boolean circuits" by "arithmetic circuits" in the statement of the result (at the cost of complicating the proof slightly). Using this observation, together with Corollary 4.3 and Theorem 4.4, we obtain Corollary 4.5.

COROLLARY 4.5. *Let F be an arbitrary field. Then each of the problems "Iterated Product of Polynomials (F)," "Polynomial Interpolation (F)," "Evaluation of Elementary Symmetric Polynomials (F)," and "Division with Remainder of Polynomials (F)" can be solved using a (nonuniform) family of arithmetic circuits (over F) of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

These results first appeared in Eberly [9]. Bini and Pan [4] have independently generalized Bini's algorithm for "Division with Remainder of Polynomials (F)" to obtain optimal order depth circuits for this problem over arbitrary infinite fields, and over each finite field. Hence they independently prove the results stated for "Division" in Corollary 4.3 and Theorem 4.4. As they note, their more direct algorithms can be

implemented using smaller (and hence more efficient) circuits than are obtained using the method described here.

5. Arithmetic computations: Adding uniformity. The circuits we have discussed in the last section have been called "nonuniform"—mainly because we have yet to define "uniform" families of arithmetic (or arithmetic-Boolean) circuits. In this section we discuss several definitions of uniformity proposed recently (von zur Gathen [14]). Our algorithms can be used (with minor modifications) to produce uniform families of arithmetic-Boolean circuits for each of these definitions.

It can be argued that a precise definition of "uniformity" for families of arithmetic circuits is not really necessary, provided that we consider algorithms that are reasonably simple, and that can be implemented using uniform families of Boolean circuits (for computations over $\mathbb{Q}$ and finite fields). It is clear that our algorithms have these properties; we will not improve our Boolean results by showing that the algorithms can be implemented using "uniform" families of arithmetic-Boolean circuits. On the other hand, it is clear that the arithmetic problems we have discussed are closely related. We make this statement more precise by adopting formal definitions of arithmetic uniformity, complexity classes, and reductions—and exhibiting reductions between the problems.

The definitions we use were proposed in von zur Gathen [14]; to our knowledge, they have not been used anywhere else. It is conceivable that they will be replaced by a different set of uniformity criteria; we regard them as provisional. They also include technical details beyond the scope of this paper. Consequently, we give a less formal presentation of uniformity. Precise statements of our results and a sketch of their proofs are given at the end of this section.

The criteria for uniformity proposed by von zur Gathen are similar to criteria for uniformity of families of Boolean circuits: families of arithmetic-Boolean circuits are considered to be "uniform" if the circuits have "descriptions" that can be computed quickly (using a deterministic Turing machine). Unfortunately, the task of defining a "circuit description" for an arithmetic-Boolean circuit over an arbitrary field is nontrivial. Encoding nodes for arithmetic operations $(+, -, \times, \div)$, Boolean operations, and connections between these nodes is not difficult. However, we must also encode arithmetic constants—elements of the field F. If F is uncountable then we cannot avoid "missing" some elements completely, or using encodings that do not identify unique field elements. We need a method for describing elements of F, and arithmetic-Boolean circuits, that is simple enough to be usable, but also general enough so that "reasonable" algorithms can be implemented using families of arithmetic-Boolean circuits that we can encode easily.

Von zur Gathen presents two encoding methods and develops definitions for "uniform" families, complexity classes, and reductions for each. One of these is quite generous: it is not difficult to show that the constants (and circuits) for our algorithms can be encoded using this method, and that the encodings can be computed using a deterministic Turing machine using space logarithmic in the number of inputs. That is, our algorithms can be implemented using "L-L-uniform" families of arithmetic-Boolean circuits.

Unfortunately, this first encoding method and definition of uniformity may be too generous. For example, we would like to be able to take (as "input") an encoding of an arithmetic-Boolean circuit over $\mathbb{Q}$ or over a finite field, and produce (as "output") an encoding of a Boolean circuit. This process is nontrivial, because it is not clear how to obtain binary representations of constants from their "encodings" efficiently.

Von zur Gathen proposes a second method for encoding circuits that is simpler, but also much more restrictive. Under this method, the only constants that can be encoded (hence the only constants used in "uniform" families of arithmetic-Boolean circuits) are the arithmetic constants 0 and 1 and the Boolean constants **T** and **F**.[2] Elements of the prime field of F can be computed from these using size linear and depth logarithmic in the lengths of their binary representations; no other constants can be supplied or computed. Now the task of defining circuit encodings, and uniformity, is straightforward. However, showing that an algorithm can be implemented using a "uniform" family of arithmetic-Boolean circuits is much more difficult.

We do not expect that this second scheme will be adopted as a criterion for uniformity—it is clearly much too restrictive. We use it here to make our statements as strong as possible: we view these results as evidence that our algorithms are uniform for *any* reasonable definition of uniformity.

In fact, we do not obtain log space constructible families of arithmetic-Boolean circuits for our algorithms when this restrictive encoding method is used. Instead, we obtain a formal reduction from our arithmetic problems to the Boolean problem "Iterated Integer Product." For each problem, we therefore have "L-uniform" (log space constructible) families of arithmetic-Boolean circuits of depth $O(\log N \log \log N)$ and size $N^{O(1)}$, and "P-uniform" (polynomial time constructible) families of arithmetic-Boolean circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N. In some sense we have reversed the usual roles of "arithmetic" and "Boolean" computations: we have shown that any improvement in the known results for a (concrete) Boolean problem yields a corresponding improvement in results for a set of (abstract) arithmetic problems.

In the rest of this section we give a precise statement of our results for "uniform" arithmetic computations and sketch their proof. For the sake of brevity we omit most of the details and leave it as an exercise for the reader to develop the full proof using this sketch and the results in von zur Gathen [14]. The reader who is not interested in these details can skip ahead to § 6.

THEOREM 5.1. *Let F be a field.*

(1) *Each of the problems "Iterated Product of Polynomials (F)," "Polynomial Interpolation (F)," "Evaluation of Elementary Symmetric Polynomials (F)," and "Division with Remainder of Polynomials (F)" can be solved using an* L-L*-uniform family of arithmetic-Boolean circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

(2) *Each of the above problems is reducible to "Iterated Integer Product." Hence each can be solved using an* L*-uniform family of arithmetic-Boolean circuits of depth $O(\log N \log \log N)$ and size $N^{O(1)}$, or a* P*-uniform family of arithmetic-Boolean circuits of depth $O(\log N)$ and size $N^{O(1)}$, for input size N.*

Proof. We consider the problem "Iterated Product of Polynomials (F)." Reif's reductions can be used to prove the results stated for the other problems.

(1) We first note that it is sufficient to consider descriptions of constants for our algorithms; the rest of the circuit descriptions can clearly be computed efficiently.

Let K be the prime field of F ($\mathbb{F}_p$ if F has characteristic $p > 0$, or $\mathbb{Q}$ otherwise). Von zur Gathen proposes that we "describe" constants in F using encodings of division free straightline programs computing polynomials with coefficients in K, together with a small amount of extra information indicating how these polynomials are to be used.

[2] In fact, von zur Gathen only allows the arithmetic constant one. However, we note that the constants 0, **T**, and **F** are easily obtained as well by using subtraction in F to obtain zero, and zero tests to obtain the Boolean constants.

Since K is countable, it is easy to define encodings of these straightline programs. A family of arithmetic-Boolean circuits is "L-L-uniform" if the circuit description (including encodings of straightline programs describing constants) for the circuit for M inputs can be computed using a deterministic Turing machine using space $O(\log M)$. We illustrate the use of straightline programs to describe constants in F using the constants required by our algorithms for "Iterated Product of Polynomials." We leave it to the reader to verify that encodings of the straightline programs described here comprise "standard encodings of descriptions of constants" for the constants we need (see von zur Gathen [14] for the definitions required).

Suppose F is infinite. For n input polynomials each with degree at most m, our algorithm requires the following constants:

(i) $N=(mn+1)$ distinct field elements $\gamma_1, \gamma_2, \cdots, \gamma_N$;

(ii) The entries of the nonsingular Vandermonde matrix $V(\gamma_1, \gamma_2, \cdots, \gamma_N)$;

(iii) The entries of $V(\gamma_1, \gamma_2, \cdots, \gamma_N)^{-1}$.

We simplify the problem of describing these constants by requiring that $\gamma_i = \gamma^{i-1}$ for some $\gamma \in F$ (so $\gamma_1 = 1$, $\gamma_2 = \gamma$, $\gamma_3 = \gamma^2$, etc.). We can use as γ any element of F that is not a root of the polynomial

$$g = \prod_{i=1}^{N-1} (y^i - 1) \in K[y],$$

for K the prime field of F.[3]

We will supply as constants the entries of adj $V(\gamma_1, \gamma_2, \cdots, \gamma_N)$ and det $V(\gamma_1, \gamma_2, \cdots, \gamma_N)$, instead of the entries of $V(\gamma_1, \gamma_2, \cdots, \gamma_N)^{-1}$; note that the latter values can be computed easily from the former. Now each of the constants γ_i and $V(\gamma_1, \gamma_2, \cdots, \gamma_N)_{st}$ (for $1 \leqq s, t \leqq N$) is a small power of γ. For $1 \leqq i \leqq N$, the entries of the ith column of adj $V(\gamma_1, \gamma_2, \cdots, \gamma_N)$ are the coefficients of the polynomial

$$(-1)^{N-i} \prod_{\substack{j=1 \\ j \neq i}}^{N} (x - \gamma_j) \prod_{\substack{k=2 \\ k \neq i}}^{N} \prod_{\substack{h=1 \\ h \neq i}}^{k-1} (\gamma_k - \gamma_h);$$

det $V(\gamma_1, \gamma_2, \cdots, \gamma_N) = \prod_{i=2}^{N} \prod_{j=1}^{i-1} (\gamma_i - \gamma_j)$. Each of these values can be computed from γ using a log space constructible straightline program of size $N^{O(1)}$. The polynomial g describing γ can also be computed using a log space constructible straightline program. Hence we have established the first part of the theorem, for infinite fields.

We next consider computations over the finite field $F = \mathbb{F}_{p^k} = \mathbb{F}_q$. If $|F| = p^k > N$ then we use the constants we have just described—and these have log space computable descriptions. Otherwise, we must supply the constants described in § 4. That is, we include the following:

(i) The coefficients of the minimal polynomial ψ (over F) of a generator γ of the multiplicative subgroup of an extension $E \supseteq F$ of size greater than N ($\psi \in F[y]$, $\psi(\gamma) = 0$, and $E = F(\gamma) = F[y]/(\psi)$).

(ii) A table of discrete logarithms (for base γ) for E: that is, the coefficients of the polynomials $(y^i \bmod \psi)$, for $0 \leqq i \leqq |E| - 2$.

(iii) Representations of the entries of the matrices $V(1, \gamma, \gamma^2, \cdots, \gamma^{N-1})$ and adj $V(1, \gamma, \gamma^2, \cdots, \gamma^{N-1})$, and of det $V(1, \gamma, \gamma^2, \cdots, \gamma^{N-1})$. Each of the elements δ of E can be represented by a vector of coefficients (in F) of a polynomial $h \in F[y]$

[3] In fact, the resulting circuit description describes a *set* of circuits, corresponding to different choices for this constant γ. The description of constants is correct because this set is nonempty, and because *any* circuit in the set can be used to solve our problem.

such that $h(\gamma) = \delta$. We use this method to represent the entries of these matrices (using constants in F).

We also include as a constant a generator α of F over $\mathbb{F}_p$. The description of the constant α is an encoding of a straightline program encoding the minimal polynomial of α over $\mathbb{F}_p$. As a description of any other constant β (in F), we use an encoding of a straightline program computing a polynomial $k \in \mathbb{F}_p[y]$ such that $k(\alpha) = \beta$.

It is actually easy to compute "descriptions" for the generator α and for the constants listed in (i) and (ii), because the fields F and E are very small. The minimal polynomial of α over $\mathbb{F}_p$ and the minimal polynomial ψ of γ over $F = \mathbb{F}_p(\alpha)$ have binary representations with length $O(\log N)$; we can use an exhaustive search to find these polynomials. Since ψ is small, we can also compute descriptions for the constants in (ii) using space $O(\log N)$ by a straightforward method. We obtain log space constructible straightline programs computing the constants in (iii) using the expressions for the determinant and entries of the adjoint of the Vandermonde matrix $V(\gamma_1, \gamma_2, \cdots, \gamma_N)$ stated in the proof of part (1) for infinite fields.

(2) As we have noted, we can only use "constants" from the prime field K of F when using this more restrictive definition of uniformity. We cannot assume K will include a suitable set of constants. Instead, we compute the value γ (described above) as an element of $G = K(\alpha_1, \alpha_2, \cdots, \alpha_M)$ where $\alpha_1, \alpha_2, \cdots, \alpha_M$ are the inputs for our circuit; or, if G is small, we choose γ from a field extension of G. This method is correct for our problems because our outputs are all rational functions of our inputs: hence we can "assume" $G = F$. Note that this assumption could not be made safely for all problems. For example, we could not make this assumption when describing circuits for factorization of polynomials over F.

While nontrivial, it can be shown that the element γ can be computed using (log space constructible) circuits of size $N^{O(1)}$ and depth $O(\log N)$. We can also compute a table of discrete logarithms for a small extension E of G at this cost, if G is small. We must also compute the determinant and adjoint of the Vandermonde matrix $V(1, \gamma, \gamma^2, \cdots, \gamma^{N-1})$; this requires the inversion of an integer Vandermonde matrix (if we are to use "uniform" circuits for the computation). While the rest of the computation can be performed using depth $O(\log N)$, this step is reducible to "Iterated Integer Product." Hence we obtain the reduction stated in the theorem. □

We obtain different bounds when using the different methods of describing constants because the first method is much more generous: we are only required to show that we can *encode* straightline programs using space $O(\log N)$. It is *not* required that these programs can be implemented using circuits wiith depth $O(\log N)$. We have algorithms for our problems that use constants whose descriptions can be computed using small space, but we do not know how to compute the constants from their descriptions using circuit depth $O(\log N)$. When using the second definition of uniformity, we must "construct" these constants as part of our circuits, and the depth of the circuits increases.

6. Extensions and open problems. We conclude with some extensions of our results and suggestions for further work.

The set of domains D over which we can perform polynomial arithmetic using small depth Boolean circuits is easily extended to include rings of polynomials in h indeterminates (for any fixed integer h) over $\mathbb{Z}$ or any of the fields $\mathbb{F}_p$, $\mathbb{Q}$, $\mathbb{F}_{p^k}$, or number fields. The methods used in § 3 can be applied to obtain these extensions.

We have not emphasized the size of the circuits we obtained. While they have size $N^{O(1)}$ for input size N, we have a substantial increase over the size of the best

sequential algorithms. With some effort, we can reduce this increase for arithmetic computations over infinite fields. Bini and Pan [3]–[5] obtain optimal order depth parallel algorithms for "Polynomial Division with Remainder" for infinite fields that have asymptotically smaller size than can be obtained by our methods. No such "optimal depth, size efficient" algorithms for computations over finite fields are known.

Von zur Gathen [14] includes a precise definition for "NC^1_F-reducibility" of arithmetic problems. Since we did not obtain optimal order depth circuits for our arithmetic problems, using the strictest definition of uniformity proposed, reductions between the problems are of some interest. If F is a field with characteristic 0 then all of the arithmetic problems (over F) listed in § 1 are NC^1_F-equivalent. For an arbitrary field F, they are all NC^1_F-reducible to "Iterated Product of Polynomials (F)." However, it is not known whether the problem "Iterated Product of Polynomials (F)" is NC^F_1-reducible to "Division with Remainder of Polynomials (F)," if F is a field with positive characteristic.

Having seen that our problems are all solvable using (polynomial time constructible) families of bounded fan-in circuits of polynomial size and logarithmic depth, it seems natural to ask whether they can also be solved using *unbounded fan-in* circuits of constant depth and polynomial size. As we show below, we can answer part of this question by applying the results of Furst, Saxe, and Sipser [12] and Smolensky [21] in a fairly direct way.

For the Boolean case, it is easy to see that the answer to this question is "no"—at least, for our integer problems. That is, these problems *cannot* be solved using unbounded fan-in Boolean circuits of constant depth and polynomial size: Furst, Saxe, and Sipser show that *parity* cannot be computed using Boolean circuits of this type and give a "constant depth, polynomial size" reduction from *parity* to the computation of the (binary representation of the) product of two n-bit integers (see [12]). It is a simple matter to show that this *multiplication* problem is "constant depth, polynomial size" reducible to each of our integer polynomial problems—so that none of them can be solved using constant depth, polynomial size Boolean circuits. It is possible that some of the Boolean computations for polynomials over finite field that are discussed in this paper can be performed using unbounded fan-in Boolean circuits of constant depth and polynomial size.

The arithmetic case is quite different. If we allow unbounded fan-in gates for both addition and multiplication, then it is easy to show that the computations for polynomials over an *infinite* field F discussed in § 4 can be performed using unbounded fan-in arithmetic circuits over F of polynomial size and constant depth: we need only implement the algorithms presented in that section. It is almost as easy to show that this is *not* the case when $F=\mathbb{F}_2$: we simply note that, given $\alpha_1, \alpha_2, \cdots, \alpha_n \in \mathbb{F}_2$, we can determine whether the majority of these values is one by examining the coefficients of $\prod_{i=1}^{n}(x-\alpha_i)$. Thus *majority* is "constant depth, polynomial size" reducible to the problem "Iterated Product of Polynomials ($\mathbb{F}_2$)." Now if $F=\mathbb{F}_2$, and we identify elements of $\mathbb{F}_2$ with the Boolean values zero and one, then an unbounded fan-in $\times$-gate is equivalent to an unbounded fan-in $\wedge$-gate, and an unbounded fan-in $+$-gate is equivalent to a *parity*- or MOD_2-gate. Hence the statement that "Iterated Product of Polynomials ($\mathbb{F}_2$)" *cannot* be solved using unbounded fan-in arithmetic (or even arithmetic-Boolean) circuits of constant depth and polynomial size is a direct result of the corollary of Theorem 2 of Smolensky (see [21]). It is conceivable, however, that "Division with Remainder of Polynomials ($\mathbb{F}_2$)," or some of our problems for polynomial arithmetic over finite fields other than $\mathbb{F}_2$, can be solved using unbounded fan-in arithmetic circuits of constant depth and polynomial size.

Finally, we ask whether the above (positive) results for computations over infinite fields by unbounded fan-in arithmetic circuits remain valid if we consider circuits with unbounded fan-in gates for addition, but with fan-in 2 gates for multiplication. Conventional wisdom, and our own intuition, would suggest that the answer is "no." However, Kung has shown that exponentiation (computing x^n given $x \in F$ and $n \in \mathbb{N}$) can be performed using constant depth, polynomial size (in n) arithmetic circuits over F, with gates of this type—provided that the field F contain an nth primitive root of unity (see [15, Algorithm 3.1]). We do not know whether this result can be generalized, either to the problems we have discussed, or to exponentiation over a more general class of ground fields.

Acknowledgments. Many thanks go to my advisor, Joachim von zur Gathen, who suggested the problem and made many contributions. I also thank Alan Borodin and Faith Fich for much helpful advice.

REFERENCES

[1] P. BEAME, S. COOK, AND H. J. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994-1003.

[2] D. BINI, *Parallel solutions of certain Toeplitz linear systems*, SIAM J. Comput., 13 (1984), pp. 268-276.

[3] D. BINI AND V. PAN, *Fast parallel polynomial division via reduction to triangular Toeplitz matrix inversion and to polynomial inversion modulo a power*, Inform. Process. Lett., 21 (1985), pp. 79-81.

[4] ———, *Fast parallel algorithms for polynomial division over arbitrary fields of constants*, Comput. Math. Appl., 12A (1986), pp. 1105-1118.

[5] ———, *Polynomial division and its computational complexity*, J. Complexity, 2 (1986), pp. 179-203.

[6] 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.

[7] A. BORODIN, J. VON ZUR GATHEN, AND J. HOPCROFT, *Fast parallel matrix and* GCD *computations*, Inform. and Control, 52 (1982), pp. 241-256.

[8] S. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2-22.

[9] W. EBERLY, *Very fast parallel matrix and polynomial arithmetic.* Tech. Report 178/85, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1985; Extended Abstract in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computing Society, Long Beach, CA, 1984, pp. 21-30.

[10] F. FICH AND M. TOMPA, *The parallel complexity of exponentiating polynomials over finite fields*, J. Assoc. Comput. Mach., 35 (1988), pp. 651-667.

[11] M. J. FISCHER AND M. S. PATERSON, *String matching and outer products*, SIAM-AMS Proc. 7, 1974, pp. 113-125.

[12] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits and the polynomial-time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13-27.

[13] J. VON ZUR GATHEN, *Parallel algorithms for algebraic problems*, SIAM J. Comput., 13 (1984), pp. 802-824.

[14] ———, *Parallel arithmetic computations: A survey*, in Proc. 12th Internat. Symposium on Mathematical Foundations of Computer Science, Bratislava, Czechoslovakia, Lecture Notes in Computer Science 233, Springer-Verlag, Berlin, New York, 1986, pp. 93-112.

[15] H. T. KUNG, *New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences*, J. Assoc. Comput. Mach., 23 (1976), pp. 252-261.

[16] V. PAN, *The bit-operation complexity of the convolution of vectors and of the* DFT, Tech. Report 80-6, Computer Science Department, State University of New York, Albany, NY, 1980.

[17] J. REIF, *Logarithmic depth circuits for algebraic functions*, SIAM J. Comput., 15 (1986), pp. 231-242.

[18] W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365-383.

[19] J. E. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.

[20] A. SCHÖNHAGE, *Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients*, in Proc. European Computer Algebra Conference, Marseille, France, Lecture Notes in Computer Science 144, Springer-Verlag, Berlin, New York, 1982, pp. 3-15.

[21] R. SMOLENSKY, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 77–82.

[22] J. VON ZUR GATHEN, *Inversion in finite fields using logarithmic depth*, J. Symb. Comput., to appear.

[23] B. E. LITOW AND G. I. DAVIDA, $O(\log(n))$ *parallel time finite field inversion*, in Proc. 3rd Aegean Workshop on Computing, Corfu, Lecture Notes in Computer Sciences 319, Springer-Verlag, Berlin, New York, 1988, pp. 74–80.

[24] V. PAN, *How to Multiply Matrices Faster*, Lecture Notes in Computer Science 179, Springer-Verlag, Berlin, New York, 1984.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 977-989, October 1989

009

TOPOLOGICAL COMPLETENESS IN AN IDEAL MODEL FOR POLYMORPHIC TYPES*

ERNST-ERICH DOBERKAT†

Abstract. The topological structure underlying the ideal model of recursive polymorphic types proposed by MacQueen, Plotkin, and Sethi are examined. It is shown that their central argument in establishing a well-defined semantical function, viz., completeness with respect to a metric obtained from the construction of their domain, is a special case of complete uniformities, which arise in a natural way from the study of closeness of ideals on domains. These uniformities are constructed and studied, and a general fixed-point theorem is derived for maps defined on these ideals.

Key words. polymorphic topological type models, domain theory, fixed points

AMS(MOS) subject classifications. 68Q55, 06B10, 54H25

1. Introduction. MacQueen, Plotkin, and Sethi discuss in [6] the semantics of types and propose σ-ideals over domains as a formalization of the naive view of types as sets of values. This is based on the following observations. Naively, one may tend to model types as sets of values: saying that the variable a has the type τ means that a always will be a member of the set modeling τ. Since types are thought to preserve structural similarity, and since this latter property should be preserved by approximations (i.e., it is closed *downward*) and by taking least upper bounds of consistent sets of values (i.e., it is closed *upward*), the formulation of a type as a set of values having all the properties of a σ-ideal is quite attractive (see [6, § 1] for a more detailed account).

In their model, self-application and recursion is possible, i.e., a type τ may be described by an expression such as $\tau = f(\tau)$, e.g., $\tau = \tau \to \sigma$. Hence, it has to be established that types of this kind do exist. This is not too difficult if the type constructor f has pleasant mathematical properties, monotonicity and continuity among them. In the case of type constructors, however, these properties cannot be guaranteed (cf., the zig-zag example in [6, pp. 98-99]). Hence, other ways of establishing the existence of a fixed point have to be found, and MacQueen, Plotkin, and Sethi propose using Banach's celebrated fixed point theorem.

This theorem is a tool that is long established in numerical mathematics to make sure that numerical iterations converge. Under proper conditions, the machinery associated with this theorem works as follows: if the convergence of an iteration has to be established, this iteration is formulated in terms of a function, mapping the search space into itself in such a way that the wanted value appears as a fixed point of the mapping. If the map can be shown to be a contraction, then convergence of the iterates to the fixed point follows from Banach's theorem. The underlying mathematical structure for the Banach theorem is a complete metric space, and the map to be considered must be a contraction, i.e., the distance between the image of two points must be by a proportionality factor smaller than the distance of these points. MacQueen,

* Received by the editors November 17, 1986; accepted for publication (in revised form) October 20, 1988. This research was funded in part by ESPRIT contract 1227 (1271) and by the Volkswagen Foundation. A preliminary version of this paper was presented at the Third Workshop on the Mathematical Foundations of Programming Language Semantics, Tulane University, New Orleans, Louisiana 70118, April 1987. This work was done while the author was on the faculty of the Computer Science Department of the University of Hildesheim.

† Department of Mathematics, Section for Computer Science/Software Engineering, University of Essen, D-4300 Essen 1, Federal Republic of Germany.

Plotkin, and Sethi establish such a complete metric on the space of all σ-ideals, which is used in that paper as a type model. This metric is introduced using an arbitrary witness function, and a closer look at the situation and at the completeness proof suggests that this scenario may indeed be a special case of a far more general one, since nature as well as properties of the witness function are rather negligible—the only interesting thing here is that the witness function exists.

In this note we demonstrate that the case considered in [6] is a special case from a topological point of view. We consider the usual generalization of metric spaces—uniform spaces—and show that the method to define a metric space may be applied to a more general one yielding complete uniform spaces.

We characterize those uniformities on the set of all σ-ideals of a domain that are generated by a rank function. Completeness was only a tool for establishing fixed points, so we discuss fixed points in greater detail, establishing from Landes's Fixed Point Theorem [5] for uniform spaces a characterization of those maps for which a unique fixed point exists. This is done first in the general uniform setting over the space of all ideals, and then a special case including the metric situation is derived from this.

The main technical innovation of this paper is to impose a uniform structure on a set of types and to establish existence of recursively defined types using a rather general fixed-point theorem. Thus, we propose replacing the metric arguments introduced by MacQueen, Plotkin, and Sethi with uniform ones, when it comes to using completeness results and in particular fixed-point arguments. Arguments from the theory of uniform spaces are usually easier to apply from a technical point of view and have the advantage of not involving the use of real numbers, since uniformities may be formulated using intrinsic properties of the objects under consideration. In addition, we expect that the results here may ease the application of fixed-point arguments without having to struggle through the technical details of metric or uniform spaces. Such fixed-point constructions seem to occur in mathematical semantics quite frequently (apart from [6], see e.g., [1]).

The present note is organized as follows: § 2 collects some notations from domain theory and from uniform spaces, and § 3 deals with Landes's fixed-point theorem. In § 4, we generalize the metric investigated by MacQueen, Plotkin, and Sethi to specific uniformities, and we study the question under which conditions the uniformity may be generated by a metric. Section 5 contains some results on fixed points, and in § 6 we indicate how the methods and results obtained may be used in the situation studied in [6].

2. Some preliminaries. For easier reference and to make the paper self-contained, we collect some notations. The standard reference for topological and uniform spaces is [4], and for domains we refer to [7].

A complete partial order (cpo) is a partial order $\leqq$ on a set D which has a smallest element $\perp$ such that each increasing sequence has its supremum in D. An element $x \in D$ is said to be *compact* if for each increasing sequence $(y_n)_{n\in\mathbb{N}}$ in D with $x \leqq \sup_{n\in\mathbb{N}} y_n$ there is an index k such that $x \leqq y_k$ holds. D is *algebraic* if there are countably many compact elements, and if for each $x \in D$ the set

$$L_x := \{a \in D;\ a \text{ is compact with } a \leqq x\}$$

is directed such that $x = \sup L_x$ holds (as usual, here we call a set L *directed* if two elements in L have an upper bound in L). Denote by A^o all compact elements in A, hence $A^o = A \cap D^o$. D is called a *domain* if it is an algebraic cpo such that each subset

bounded from above has a supremum. We fix a domain D for the rest of this paper.

A nonempty set $I \subset D$ is said to be an *ideal* if it is downward closed; hence, if $x \leqq y \in I$ implies $x \in I$ for each $x \in D$, $y \in I$; denote by $\mathscr{I}(D)$ the set of all ideals in D. An ideal is called a *σ-ideal* if it is closed with respect to taking suprema of increasing sequences; denote by $\mathscr{I}_\sigma(D)$ the set of all σ-ideals in D and by $\mathscr{I}_\sigma^*(D)$ the augmented set $\mathscr{I}_\sigma(D) \cup \{\varnothing\}$. The map $I \mapsto I^o$ is a bijection from $\mathscr{I}_\sigma(D)$ to $\mathscr{I}(D^o)$, the inverse of which is given by

$$\mathscr{I}(D^o) \ni I \mapsto \{\sup_{n\in\mathbb{N}} x_n;\ (x_n)_{n\in\mathbb{N}} \subset I \text{ is increasing}\}$$

([6], Proposition 1).

A subset $\mathscr{A}$ of the power set $\mathscr{P}(S)$ of a set S is a *filterbase* if $\varnothing \notin \mathscr{A}$, $\mathscr{A} \neq \varnothing$, and if we may find for each $A_1, A_2 \in \mathscr{A}$ an element $A \in \mathscr{A}$ with $A \subset A_1 \cap A_2$. A *filter* $\mathscr{A}$ is a filterbase that is a dual ideal, i.e., if the following implication holds:

$$A \in \mathscr{A}, \qquad A \subset B \Rightarrow B \in \mathscr{A}.$$

A filter $\mathscr{G}$ on $S \times S$ is said to be a *uniformity* if

(1) $\forall G \in \mathscr{G}$: $G^{-1} := \{(y, x);\ (x, y) \in G\} \in \mathscr{G}$ (G^{-1} is called the inverse of G).

(2) $\Delta \in \mathscr{G}$, where $\Delta := \{(x, x);\ x \in S\}$ is the diagonal.

(3) $\forall G \in \mathscr{G}\ \ \exists H \in \mathscr{G}$: $H \circ H \subset G$
(here the composition operator is defined as usual by $I \circ J := \{(x, z);$
$\exists y$: $(x, y) \in I$ and $(y, z) \in J\}$).

We then call $(S, \mathscr{G})$ a *uniform space.*

A filterbase generating a uniformity is called a *base* for this uniformity. If d is a metric on S (i.e., if $d : S \times S \to \mathbb{R}_\oplus$ with $\forall y\ \ \forall y$: $d(x, y) = d(y, x)$; $\forall x \forall y$: $d(x, y) = 0 \Leftrightarrow x = y$; $\forall x \forall y \forall z$: $d(x, z) \leqq d(x, y) + d(y, z)$), then it is easily seen that the smallest filter $\mathscr{G}_d$ on $S \times S$ that contains all the sets $\{(x, y);\ d(x, y) \leqq r\}$ is a uniformity. In a metric space, we may numerically evaluate the closeness of two points x, y by computing $d(x, y)$; in a uniform space, closeness of x and y is expressed in terms of the elements of $\mathscr{G}$: x and y are close with respect to $H \in \mathscr{G}$ if and only if $(x, y) \in H$. Thus, uniform spaces generalize metric spaces; they are, in turn, less general than topological spaces.

We will use two constructions from the theory of uniform spaces, viz., the product of two uniformities and the Hausdorff uniformity on the space $\mathscr{CL}(\mathscr{S})$ of all nonempty closed subsets of a uniform space. Both constructions will be needed later when we show that under certain topological conditions, fixed points do exist. This will be described in terms of the Hausdorff uniformity on the space $\mathscr{CL}(\mathscr{S} \times \mathscr{S})$ for a suitably chosen uniform space $\mathscr{S}$. Let (S, G) be a uniform space. First, let $W \subset \mathscr{S}^4$, then define the shuffle

$$\langle W \rangle := \{(a, c, b, d);\ (a, b, c, d) \in W\}.$$

Then the set of all shuffled rectangles

$$\mathscr{G} \boxtimes \mathscr{G} := \{\langle A \times B \rangle;\ A, B \in \mathscr{G}\}$$

defines the basis for the *product uniformity* $\mathscr{G} \otimes \mathscr{G}$ on $S \times S$ (intuitively, a pair is close if and only if both components are close to each other). If $\mathscr{G}$ has a base $\mathscr{G}_o$, $\mathscr{G}_o \boxtimes \mathscr{G}_o$ forms a base for $\mathscr{G} \otimes \mathscr{G}$. Turning to $\mathscr{CL}(S)$, we define for $A \subset S$, $U \in \mathscr{G}$ the U-neighborhood $U(A)$ of A by

$$U(A) := \{y \in S;\ (x, y) \in U \quad \text{for some } x \in A\};$$

two sets $C, D \in \mathscr{SL}(S)$ are U-close if and only if they lie in the U-neighborhoods of each other. Formally,

$$[U] := \{(C, D);\ C, D \in \mathscr{CL}(S) \quad \text{with } C \subset U(D),\ D \subset U(C)\}$$

then $\{[U];\ U \in \mathscr{G}\}$ defines the basis for a uniformity on $\mathscr{CL}(S)$, the so called *Hausdorff uniformity* (see [2, § II]).

$(S, \mathscr{G})$ is called *separated* (or *Hausdorff*) if and only if $\bigcap \mathscr{G} = \Delta$; so two distinct points can always be separated by an element of $\mathscr{G}$. The space is called *complete* if and only if each Cauchy net converges. This means the following: a *net* $(n_r)_{r \in R}$ in S is a map $R \ni r \mapsto n_r \in S$ such that R is a set that is directed by a transitive and reflexive relation (which is usually omitted in the notation; the natural numbers $\mathbb{N}$ are a directed set under the natural order; hence, each sequence is a net). The net $(n_r)_{r \in R}$ *converges to* n if and only if elements of the net are eventually close to n; hence if and only if

$$\forall U \in \mathscr{G} \quad \exists r_o \in R \quad \forall R \ni r \geqq r_o\colon (n_r, n) \in U$$

and it is a *Cauchy net* if and only if

$$\forall U \in \mathscr{G} \quad \exists r_o \in R \quad \forall r, s \geqq r_o\colon (n_r, n_s) \in U,$$

hence if and only if the elements of the net are eventually close to each other.

It is not too difficult to see that $(S \times S, \mathscr{G} \otimes \mathscr{G})$ is a Hausdorff space, if $(S, \mathscr{G})$ is one, and that this space is complete if $(S, \mathscr{G})$ is. In addition, $\mathscr{CL}(S)$ with the Hausdorff uniformity is a separated space whenever $(S, \mathscr{G})$ is [2, Thm. II-12, p. 45].

Finally, if $A, B \subset S$, define the symmetric difference $A \triangle B$ of A and B by $(A \cup B) - (A \cap B)$. It is rather easy to see that $(\mathscr{P}(S), \triangle)$ is an Abelian group, and that for all $A, B, C \subset S$

$$A \triangle B \subset (A \triangle C) \cup (C \triangle B)$$

as well as

$$(A \triangle B) \cap C = (A \cap C) \triangle (B \cap C)$$

hold (see, e.g., [3, p. 6]).

3. Fixed points. Since we will study fixed points over uniform structures on the set of all σ-ideals of a domain as a generalization of [6], we need an extension of the fixed-point arguments of the Banach type. Recently, Thomas Landes gave a suitable generalization of Banach's celebrated theorem that fits into the framework considered here and that will be of considerable use in the arguments below. Since this result has not been published yet, and in order to make the present paper self-contained, we give Landes's result in greater detail and sketch its proof (for technical reasons the arrangement is a bit different from the one given by Landes; however, the proofs are entirely due to Landes).

For this section fix a uniform space $(S, \mathscr{G})$ with a complete and separated uniformity $\mathscr{G}$ having $\mathscr{G}_o$ as a base. We begin with some definitions.

DEFINITION 3.1. A map $F\colon \mathscr{G}_o \to \mathscr{G}_o$ is said to be an *attractor* if and only if the following condition holds

(A-1) $$\forall U \in \mathscr{G}_o\colon F(U) \subset U.$$

(A-2) $$\forall U \in \mathscr{G}_o \quad \exists N = N(U) \in \mathbb{N} \quad \forall k \in \mathbb{N}\colon F^N(U) \circ \cdots \circ F^{N+k}(U) \subset U.$$

This notion of an attractor simulates contractions in a metric space in the following sense: suppose (X, d) is a complete metric space, and $f\colon X \to X$ has a contracting

property, viz., there is $0<\beta<1$ such that $d(f(x),f(y))\leqq\beta d(x,y)$ always holds. Taking the set of balls $\{K_r(x);\ r>0,\ x\in X\}$ (with $K_r(x):=\{y;\ d(x,y)<r\}$), it is easy to see that the map $K_r(x)\mapsto f[K_r(x)]$ furnishes an attractor.

DEFINITION 3.2. A map $f: M\to M$ is called a *contraction* if

(A-3) there exists an attractor F_f with the property that for all $U\in\mathcal{G}_o$ the following holds: $(f(x),f(y))\in F_f(U)$, whenever $(x,y)\in U$.

(A-4) $$\forall x\in M\quad \forall U\in\mathcal{G}_o\quad \exists m\in\mathbb{N}\colon (f^m(x),f^{m+1}(x))\in U.$$

If in addition the condition (A-5) holds, then we call f a *strong contraction*:

(A-5) $$\forall x\neq y\quad \exists U\in\mathcal{G}_o\quad \exists m\in\mathbb{N}\colon (x,y)\notin U \text{ and } (f^m(x),f^m(y))\in U.$$

The fixed-point theorem due to Landes then reads as follows:

THEOREM 3.3. (1) *Any contraction f has a fixed point x^*, and $x^*=\lim_{n\to\infty}f^n(x)$ holds, where $x\in S$ is arbitrary.*

(2) *If f is a strong contraction, then x^* is the unique fixed point.*

Proof. (1) Uniqueness is easily established for a strong contraction, once we know that x^* exists.

(2) Fix $x\in S$, then it is enough to show that the sequence $(f^n(x))_{n\in\mathbb{N}}$ is a Cauchy sequence; completeness will then establish a limit for this sequence, which in turn is easily seen to be a fixed point. Given $U\in\mathcal{G}_o$, we may find $m\in\mathbb{N}$ such that

$$(f^m(x),f^{m+1}(x))\in U.$$

Now let $k\geqq 1$, $n\geqq N+m$, where N is chosen according to condition (A-2) for the attractor F_f, then

$$(f^n(x),f^{n+k}(x))\in F_f^N(U)\circ\cdots\circ F_f^{N+k}(U)\subset U.$$

This establishes the theorem. □

We will need to extend the notion of a contraction to several dimensions, and for the sake of demonstrating techniques we will focus on the case of two dimensions. We call a map $f: M\times M\to M$ a (*strong*) *contraction* if and only if for each $s,t\in M$ the maps $f_s:=\lambda t\cdot f(s,t)$ and $f^t:=\lambda s\cdot f(s,t)$ are (strong) contractions, where the attractors belonging to F_{f_s} and F_{f^t}, respectively, are uniformly equal, i.e., $\forall s,t\in M\colon F_{f_s}=F_{f^t}$.

4. Constructing uniformities. When defining a uniformity $\mathcal{G}$, one has to specify which elements are considered to be close to each other. If, in addition, closeness can be specified in numeric terms, i.e., in terms of a distance function, a metric space is defined.

Let $r: D^o\to\mathbb{N}$ be a rank function on the compact elements of a domain; then the closeness $c(x,y)$ for $x,y\in\mathcal{I}_\sigma(D)$ is defined by

$$c(x,y):=\begin{cases}\infty, & \text{if } x=y\\ \min\{r(k);\ k\in x^o\triangle y^o\}, & \text{otherwise.}\end{cases}$$

Hence, x and y are the closer the later one realizes that there is a witness $x^o\triangle y^o$, since $x=y$ iff $x^o=y^o$ iff $x^o\triangle y^o=\varnothing$.

The distance associated with the rank function r is then defined by

$$d(x,y):=2^{-c(x,y)}.$$

This makes $(\mathcal{I}_\sigma(D),d)$ a complete (ultra-) metric space.

The use of $c(x, y)$ suggests that in the general situation the closer the two ideals, the less distinguishable their compact elements are, i.e., the smaller $x^o \triangle y^o$ is. Taking an arbitrary set $\mathscr{A} \subset \mathscr{D}^o$ as a yardstick, we say that the ideals x and y are *$\mathscr{A}$-close* if the difference of their compact elements is witnessed by $\mathscr{A}$. This is stated formally in Definition 4.1.

DEFINITION 4.1. For $A \subset D^o$, define

$$G_A := \{(x, y);\ x, y \in \mathscr{I}_\sigma(D),\ x^o \triangle y^o \subset A\}.$$

Thus two σ-ideals are close to each other with respect to A if the difference of the trace they leave on the compact elements is contained in A. Now we will demonstrate that this constitutes a basis for a uniformity on $\mathscr{I}_\sigma(D)$.

LEMMA 4.2. *Let $\mathscr{A} \subset P(D^o)$ be a filter base, then we have the following properties*:

(a) $\mathscr{G}_{\mathscr{A}} := \{G_A;\ A \in \mathscr{A}\}$ *is the base for a uniformity $\mathscr{G}^*_{\mathscr{A}}$ (thus $\mathscr{G}^*_{\mathscr{A}}$ is the smallest uniformity containing $\mathscr{G}_{\mathscr{A}}$)*;

(b) *if $\mathscr{B}$ is another filter base with $\mathscr{A} \subset \mathscr{B}$, then $\mathscr{G}^*_{\mathscr{A}} \subset \mathscr{G}^*_{\mathscr{B}}$*;

(c) *if $\mathscr{F}(\mathscr{A})$ denotes the filter generated by $\mathscr{A}$, then $\mathscr{G}^*_{\mathscr{F}(\mathscr{A})} = \mathscr{G}^*_{\mathscr{A}}$*;

(d) *$\mathscr{G}^*_{\mathscr{A}}$ is Hausdorff, provided that $\bigcap \mathscr{A} = \varnothing$.*

Proof. (a) It is enough to demonstrate that we can find for any $A \in \mathscr{A}$ an element $B \subset \mathscr{A}$ with $G_B \circ G_B \subset G_A$. In fact, we will show that B may be chosen as A itself. Now, let $(x, y) \in G_A \circ G_A$, then we know that we can find z with $x^o \triangle z^o \subset A$ and $z^o \triangle y^o \subset A$. But because

$$x^o \triangle y^o \subset (x^o \triangle z^o) \cup (z^o \triangle y^o) \subset A,$$

we see that $(x, y) \in G_A$. The other properties of a uniformity are straightforward.

(b) Property (b) follows from the observation that $C \in \mathscr{G}^*_{\mathscr{A}}$ if and only if $G \subset C$ for some $G \in \mathscr{G}_{\mathscr{A}}$.

(c) Property (c) is established similarly: Since $F \in \mathscr{F}(\mathscr{A})$ iff $A \subset F$ for some $A \in \mathscr{A}$, and since $B \mapsto G_B$ is an increasing map, the assertion is immediate.

(d) Let $(x, y) \in \bigcap \mathscr{G}^*_{\mathscr{A}}$ be a pair of ideals, then $x^o \triangle y^o \subset \bigcap \mathscr{A}$; so we may infer that $(x, y) \in \bigcap \mathscr{G}^*_{\mathscr{A}}$ if and only if $x = y$, provided $\bigcap \mathscr{A} = \varnothing$. But this means that in the latter case $\bigcap \mathscr{G}^*_{\mathscr{A}}$ reduces to the diagonal. □

The metric defined in [6] falls rather naturally into the realm of the uniformity defined above. As indicated, let d be derived from a rank function r, then

$$d(x, y) < \varepsilon$$

iff

$$r(k) \geqq \left\lceil \log_2 \frac{1}{\varepsilon} \right\rceil$$

for each $k \in x^o \triangle y^o$, provided the σ-ideals x and y are different. It is then easily seen that the metric uniformity $\mathscr{G}_r$ coincides with $\mathscr{G}^*_{\mathscr{A}}$ where

$$\mathscr{A} := \{A_n;\ n \in \mathbb{N} \quad \text{and } A_n \neq \varnothing\}$$

with

$$A_n := \{r \geqq n\} := \{t \in D^o;\ r(t) \geqq n\}.$$

We will return to the question under which conditions $\mathscr{G}^*_{\mathscr{A}}$ equals $\mathscr{G}_r$ for some r in due course.

We are interested in the completeness of $\mathscr{G}^*_{\mathscr{A}}$. Recall that a uniform space is complete if each Cauchy net has a limit. We are going to represent the limit in terms

of the following set-theoretical construction. Let $(A_r)_{r\in R}$ be a net of sets, then

$$\liminf_{r\in R} A_r := \bigcup_{r\in R} \bigcap_{r_o \geqq r} A_{r_o}$$

is said to be the *limit inferior* of this net and is the set of all elements which are eventually in all A_r, and

$$\limsup_{r\in R} A_r := \bigcap_{r\in R} \bigcup_{r_o \geqq r} A_{r_o}$$

is called the *limit superior* of this net and is the set of all elements which are frequently in A_r.

It is rather easy to verify the following technical lemma.

LEMMA 4.3. *Let $(x_r)_{r\in R}$ be a net of ideals in D^o, then we have the following*:

(a) $\liminf_{r\in R} x_r$ *and* $\limsup_{r\in R} x_r$ *are both ideals in* D^o;

(b) $\liminf_{r\in R} x_r \subset \limsup_{r\in R} x_r$;

(c) *for every* $s \in R$, *the following inclusions hold*

$$\liminf_{s\leqq r\in R} x_r \subset \liminf_{r\in R} x_r$$

and

$$\limsup_{s\leqq r\in R} x_r \supset \limsup_{r\in R} x_r.$$

This allows the representation of a limit in purely set-theoretical terms, provided one knows that it exists. Recall that a sequence in a Hausdorff space has at most one limit.

PROPOSITION 4.4. *Let $(x_r)_{r\in R} \subset \mathcal{I}_\sigma(D)$ be a net with $\mathcal{G}^*_{\mathcal{A}} - \lim_{r\in R} x_r = x \in \mathcal{I}_\sigma(D)$, and assume that $\mathcal{G}^*_{\mathcal{A}}$ is Hausdorff. Then we have*

$$x^o = \liminf_{r\in R} x_r^o = \limsup_{r\in R} x_r^o.$$

Proof. (0) $\mathcal{G}^*_{\mathcal{A}} - \lim_{r\in R} x_r = x$ means that we can find for every $A \in \mathcal{A}$ an index $r_o = r_o(A)$ such that $x_r^o \triangle x^o \subset A$ holds for every $r \geqq r_o$.

(1) Fix $A \in \mathcal{A}$ and select r_o for A according to part 0. If $r \geqq r_o$, we have $x^o \triangle x_r^o \subset A$, or, equivalently, $x^o - A \subset x_r^o - A$. Consequently,

$$x^o - A \subset \liminf_{r\in R} x_r^o - A.$$

Because $\mathcal{G}^*_{\mathcal{A}}$ is Hausdorff, hence $\bigcap \mathcal{A} = \varnothing$, we may infer

$$E \subset F$$

from

$$\forall A \in \mathcal{A}\colon E - A \subset F - A.$$

Thus we may conclude that

$$x^o \subset \liminf_{r\in R} x_r^o.$$

Similarly, we see that

$$\limsup_{r\in R} x_r^o \subset x^o.$$

Now the conclusion follows from 4.3(b). □

From 4.4, we infer completeness for the uniformity.

PROPOSITION 4.5. *If $\mathcal{G}^*_{\mathcal{A}}$ is Hausdorff, then it is complete.*

Proof. (0) We show that each Cauchy net $(x_r)_{x\in R}$ of σ-ideals converges with respect to $\mathcal{G}^*_A$. Fix for the moment $A\in\mathcal{A}$, then we find an index $r_o\in R$ such that $x^o_r \triangle x^o_s \subset A$ whenever $r, s \geqq r_o$.

(1) From the latter inclusion we see that

$$\forall s, t \geqq r_o\colon x^o_s - A \subset x^o_t - A,$$

hence we have for any $s \geqq r_o$ the inclusion

$$x^o_s - A \subset \liminf_{s\leqq r\in R} x^o_r - A \subset \liminf_{r\in R} x^o_r - A.$$

Similarly, we see that

$$x^o_s - A \supset \limsup_{r\in R} - A.$$

Now let $x\in\mathcal{I}_\sigma(D)$ be determined by the equation

$$x^o = \liminf_{r\in R} x^o_r,$$

then we see that $x^o_s - A = x^o - A$, whenever $s \geqq r_o$. □

Let us illustrate Proposition 4.5 and our constructions with an example. It is well known that $D := \mathcal{P}(\mathbb{N})$ is a domain, using set inclusion as the order relation and that the compact elements are just the finite subsets. Count D^o using the bijection

$$\varphi : a \mapsto \sum_{t\in a} 2^t.$$

Putting

$$\mathcal{A} := \{\{\varphi \geqq n\};\ n\in\mathbb{N}\}$$

we obtain a complete Hausdorff uniformity $\mathcal{G}^*_{\mathcal{A}}$ on $\mathcal{I}^*_\sigma(D)$. Now define

$$x_n := \{a\in D;\ k\geqq n \quad \text{holds for all } k\in a\},$$

then x_n is easily seen to be a σ-ideal.

CLAIM. *$(x_n)_{n\in\mathbb{N}}$ is a Cauchy sequence.*

Proof. Given $N\in\mathbb{N}$, select k with $2^k > N$. If $s, t\in\mathbb{N}$, let

$$a \in x^o_{k+s} \triangle x^o_{k+s+t} \subset x^o_{k+s},$$

thus

$$\varphi(a) = \sum_{m\in a} 2^m > \operatorname{card}(a) 2^{k+s} > N,$$

so $(x_{k+s}, x_{k+s+t}) \in G_{\{\varphi\geqq N\}}$. Since s and t have been chosen arbitrarily, the above claim is established. □

Consequence. $\mathcal{G}^*_{\mathcal{A}} - \lim_{n\to\infty} x_n = \varnothing$.

This has been the reason for including the empty set into the set of σ-ideals that we wanted to consider. Usually, one considers only the set of nonempty σ-ideals; this is justified by the fact that the bottom element in a domain constitutes an ideal.

Let us remark that φ may serve as a ranking function, so that on one hand we have the uniform space $(\mathcal{I}_\sigma(\mathcal{P}(\mathbb{N})), \mathcal{G}^*_{\mathcal{A}})$, and on the other we have the metric space $(\mathcal{I}_\sigma(\mathcal{P}(\mathbb{N})), d)$ with the metric d as defined by MacQueen, Plotkin, and Sethi. It is not

difficult to see that the uniformity defined by the metric is just the same as $\mathcal{G}^*_{\mathcal{A}}$. We will denote the uniformity that is generated from the metric obtained from the rank function r by $\mathcal{G}_r$. Now we will discuss conditions under which the uniformity $\mathcal{G}^*_{\mathcal{A}}$ is actually generated by a rank function. Since a rank function has the property that its uniformity is generated by some countable sets, it is worthwhile to investigate countability conditions.

If $\mathcal{B}$ is a filter, then $\mathcal{B}$ is said to be *countably based* if there exists a countable filter base $\mathcal{A}$ such that $\mathcal{B}$ equals $\mathcal{F}(\mathcal{A})$. It is the countably based filters which constitute the special case of metrizable uniformities. We return to the case of a general D for an investigation of this question.

PROPOSITION 4.6. *Let $\mathcal{A}$ be a filter base with $\cap\, \mathcal{A} = \varnothing$. There exists a rank function $r: D^o \to \mathbb{N}$ with $\mathcal{G}^*_{\mathcal{A}} = \mathcal{G}^*_r$ if and only if $\mathcal{F}(\mathcal{A})$ is countably based.*

Proof. (1) If $\mathcal{G}^*_{\mathcal{A}} = \mathcal{G}^*_r$ for some rank function r, then

$$\{A_n;\ n \in \mathbb{N},\ A_n \neq \varnothing\}$$

with

$$A_n := \{r \geqq n\}$$

constitutes a countable base for $\mathcal{F}(\mathcal{A})$.

(2) If $\mathcal{F}(\mathcal{A})$ has a countable base

$$\mathcal{B} = \{B_t;\ t \in \mathbb{N}\},$$

we may assume without loss of generality that the B_t form a descending chain

$$B_1 \supset B_2 \supset \cdots$$

(otherwise, force this condition without losing the properties of $\mathcal{B}$). Let for $v \in D^o$ the rank function be defined by

$$r(v) := \operatorname{card}(\{t;\ v \in B_t\}).$$

Then r is a finite function, because $\cap\, \mathcal{B} = \varnothing$, and $B_t = \{r \geqq t\}$ is easily seen. This implies the desired equality $\mathcal{G}^*_{\mathcal{A}} = \mathcal{G}_r$. □

5. Fixed points. Completeness is only a necessary condition for establishing the existence of a fixed point for a map of a metric or a uniform space into itself. The technical condition used in [6] was that the maps in question are contractions, so that the distance between the image of two points is bounded by a constant factor times the distance between the points themselves; the constant being strictly smaller than unity. This is shown to have the effect that successive applications of the map will eventually result in closer and closer points, thus yielding a Cauchy sequence that must converge to the fixed point.

A similar strategy is employed in the case of a complete uniform space. Since here numerical measurements of distances are not possible, one has to use other techniques in establishing the convergence of the iterates of the map in question. This has been discussed in § 3, and we will show how to utilize this result in the context of σ-ideals over a domain.

It will be convenient to consider a general situation first. Let M be a complete uniform space with uniformity $\mathcal{G}$ and let $\mathcal{G}_o$ be a base for $\mathcal{G}$. Assume that $\varphi: M \to M$ is a contraction, and define for $V \in \mathcal{G}_o$

$$R_m(\varphi, V) := \operatorname{closure}((\varphi^m \times \varphi^{m+1})[V]),$$

where

$$(\varphi^m \times \varphi^{m+1})[V] := \{(\varphi^m(a), \varphi^{m+1}(b));\ (a, b) \in V\}$$

is the image of the basic neighborhood $V \subset M \times M$ under the map $(\varphi^m \times \varphi^{m+1})$, the closure being topological in the product space $M \times M$, where M is endowed with the topology canonically generated by the uniformity that is based on $\mathcal{G}_o$. Thus $(x, y) \in M \times M$ is in $R_m(\varphi, V)$ if and only if there exists a net $((a_n, b_n)_{n \in R}) \subset V$ such that

$$(x, y) = \lim_{n \in R} (\varphi^m \times \varphi^{m+1})(a_n, b_n).$$

$R_m(\varphi, \cdot)$ maps basic neighborhoods into closed and nonempty subsets of the Cartesian product. A useful condition for a fixed point may be derived from this observation: if applying two subsequent iterations of φ to an arbitrary pair in V renders the images more and more indistinguishable, or, topologically speaking, if $R_m(\varphi, V)$ is absorbed eventually by the diagonal, then condition (A-4) establishing a fixed point is satisfied.

This condition is intuitively rather clear, and in order to formulate it we need the topological framework which is sketched in § 2.

PROPOSITION 5.1. *If* $\lim_{m \to \infty} R_m(\varphi, V)$ *exists for each* $V \in \mathcal{G}_o$ *and is a subset of the diagonal, then* φ *has a fixed point.*

Proof. 0. According to Theorem 3.3 it is enough to show that given $x \in M$, and $U_1 \in \mathcal{G}_o$, there exists $m \in \mathbb{N}$ such that

$$(\varphi^m(x), \varphi^{m+1}(x)) \in U_1.$$

Since $\mathcal{G}_o$ is a base for $\mathcal{G}$, there exists for U_1 a neighborhood $U \in \mathcal{G}_o$ that is symmetric (i.e., for which $U^{-1} = U$), holds, such that $U \circ U \subset U_1$, and for x there exists a basic neighborhood V with the property that $(x, \varphi(x)) \in V$. Consequently,

$$\forall k \in \mathbb{N}: (\varphi^k(x), \varphi^{k+1}(x)) \in R_k(\varphi, V)$$

holds.

(1) Let $\Delta^* := \lim_{n \to \infty} R_n(\varphi, V)$ be the limit with respect to the Hausdorff uniformity. Since the sequence $(R_n(\varphi, V))_{n \in \mathbb{N}}$ converges, we know that there exists an index n_o such that

$$\forall n \geqq n_o: R_n(\varphi, V) \subset \langle U \times U \rangle(\Delta^*).$$

Thus $(z, z, \varphi^n(x), \varphi^{n+1}(x)) \in \langle U \times U \rangle$ for a suitably chosen $z \in M$ and all such n. This implies the desired property. □

The condition given above is a bit strong and not very practical. First, the proof shows that one need not insist on the convergence of the closures of $(\varphi^m \times \varphi^{m+1})[V]$. This has been done to assure a proper topological formulation of the sufficient condition (otherwise, one would have to have resorted to a topology on the space of all subsets of $M \times M$, a space that is too large to be topologized properly). Second, the convergence is not easily checked, so the condition is not really useful, although it applies to the situation in question as well.

Let us return to the space $\mathcal{I}_\sigma(D)$ of all σ-ideals of the domain D (note that now we can do without the empty set). Assume that we have a contraction φ defined on $\mathcal{I}_\sigma(D)$ and that furthermore a rank function $r: D^o \to \mathbb{N}$ is defining the uniformity. Let $\mathcal{A}$ be the base determined by r, then $\mathcal{G}^*_{\mathcal{A}}$ is Hausdorff. Define for each subset $A \subset D$ its weight

$$r^*(A) := \min \{r(d);\ d \in A \cap D^o\}$$

(we adopt the convention that $r^*(\varnothing) = +\infty$).

The next lemma will be an auxiliary statement establishing the existence of a fixed point. Intuitively, it describes what happens when the map φ makes the difference of

the images of two ideals harder to discern than the ideals themselves. Call a map $\varphi : \mathcal{J}_\sigma(D) \to \mathcal{J}_\sigma(D)$ *discerning* if $r^*(\varphi(x) \triangle \varphi(y)) > r^*(x \triangle y)$ holds whenever $x \neq y$.

LEMMA 5.2. *If φ is discerning, then we can find for every $A, B \in \mathcal{A}$ an index m_o such that*

$$R_m(\varphi, G_B) \subset G_A$$

holds whenever $m \geqq m_o$.

Proof. (0) We show first that

$$T := \min \{r^*(a_1 \triangle a_2); (a_1, a_2) \in (\mathrm{id} \times \varphi)[G_B]\}$$

is a finite quantity. Assume on the contrary that $T = +\infty$, then $a_1 = a_2$, whenever $(a_1, a_2) \in (\mathrm{id} \times \varphi)[G_B]$. But this means that $G_B = \{(x, x)\}$ for some $x \in \mathcal{J}_\sigma(D)$. But since the diagonal on $\mathcal{J}_\sigma(D)$ is a subset of G_B by definition, there is only one σ-ideal x on D, thus the domain is trivial, and φ is not discerning. Hence we have arrived at a contradiction.

(1) Now an inductive argument will show that

$$(+) \qquad \forall (a_1, a_2) \in (\varphi^m \times \varphi^{m+1})[G_B]: r^*(a_1 \triangle a_2) \geqq T + m.$$

This is trivial for $m = 0$ by construction of T. Now assume that $(+)$ holds for m, and consider $(a_1, a_2) \in (\varphi^{m+1} \times \varphi^{m+2})[G_B]$, thus $a_p = \varphi(b_p)$ for some $(b_1, b_2) \in (\varphi^m \times \varphi^{m+1})[G_B]$, consequently

$$\begin{aligned} r^*(a_1 \triangle a_2) &= r^*(\varphi(b_1) \triangle \varphi(b_2)) \\ &> r^*(b_1 \triangle b_2) \\ &\geqq T + m. \end{aligned}$$

(2) In $(+)$ we may substitute $R_m(\varphi, G_B)$ for $(\varphi^m \times \varphi^{m+1})[G_B]$, since the former is the topological closure of the latter, and we can find for each element in $R_m(\varphi, G_B)$ a net in $(\varphi^m \times \varphi^{m+1})[G_B]$ converging to it. Hence Proposition 4.4 actually justifies the substitution.

(3) Since we may assume that $A = \{r \geqq t\}$ for some suitably chosen value t, Lemma 5.2 now follows from relation $(+)$. □

We are now in a position allowing us to formulate a fixed-point theorem adequate for the considered situation.

PROPOSITION 5.3. *Let $\varphi : \mathcal{J}_\sigma(D) \to \mathcal{J}_\sigma(D)$ be discerning. Then φ has a fixed point.*

Proof. (1) We infer from Lemma 5.2 that for any $x \in \mathcal{J}_\sigma(D)$ and for any $B \in \mathcal{A}$ there exists $m \in \mathbb{N}$ with

$$(\varphi^m(x), \varphi^{m+1}(x)) \in G_B,$$

thus condition (A-4) is satisfied. To establish the existence of a unique fixed point, it has to be shown that the condition (A-5) is satisfied, too. For this, let x and y be σ-ideals. Because $\mathcal{G}^*_{\mathcal{A}}$ is Hausdorff, we can find $C \in \mathcal{A}$ so that G_C does not contain the pair (x, y), and we may find $B \in \mathcal{A}$ with $(\varphi(x), y) \in G_B$. From Lemma 5.2 we see that eventually $R_m(\varphi, G_B) \subset G_C$ holds, so we find m such that

$$(\varphi^{m+1}(x), \varphi^{m+1}(y)) = (\varphi^m \times \varphi^{m+1})(\varphi(x), y) \in G_C,$$

thus condition (A-5) indeed holds.

(2) It remains to be shown that φ is a quasi contraction. Define for $U \in \mathcal{A}$ the map $F : \mathcal{A} \to \mathcal{A}$ by setting

$$F(U) := \{(\varphi(x), \varphi(y)); (x, y) \in U\}$$

then by construction $(\varphi(x), \varphi(y)) \in F(U)$ holds whenever $(x, y) \in U$. We have, however, to establish the properties (A-1) as well as (A-2). Since φ is discerning, we have for $A := \{r \geqq n\}$ the property

$$x^o \triangle y^o \subset A \quad \text{implies } \varphi(x)^o \triangle \varphi(y)^o \subset A,$$

which in turn establishes (A-1). In order to establish (A-2), we first observe that

$$F(U_1 \circ U_2) = F(U_1) \circ F(U_2);$$

hence F is a homomorphism with respect to composing relations. From this, (A-2) is easily inferred, since from what we have shown it is plain that

$$U^N \subset U$$

holds for each $U \in \mathscr{A}$. □

6. Discussion. So far we have demonstrated that the metric completeness result and the application of Banach's fixed-point theorem in [6] are special cases of a situation that may more generally be described by uniformities and by a fixed-point theorem due to Landes. The metric defined in [6] was defined using a rank function, and in turn the value of this function is determined by the construction of the special kind of domain that MacQueen, Plotkin, and Sethi use. This domain is constructed using a limiting process [6, § 2.3] in which an increasing sequence of approximating domains is built up. The rank of an element is then its height in that sequence.

The results proposed here suggest that one does not need to make use of this limiting process, but that rather intrinsic properties of the uniformity are sufficient for the constructions employed in [6] as far as the semantics of type expressions are concerned. We want to demonstrate this with a simple example; rather than going through the formalism of specifying a language with recursive types, we show how one of the central specific results in [6] may be obtained using the framework sketched in this note. For this, we assume that the domain D under consideration satisfies the domain equation

$$D = Y + D \times D$$

for some domain Y, where the equality actually is an order isomorphism. Thus $D^o = Y^o + D^o \times D^o$ holds, and we may identify $(s, t) \in \mathscr{I}_\sigma(D) \times \mathscr{I}_\sigma(D)$ with the σ-ideal $s \times t \subset D \times D$. We will consider the map induced by this identification; hence, the map

$$p: \begin{cases} \mathscr{I}_\sigma(D) \times \mathscr{I}_\sigma(D) \to \mathscr{I}_\sigma(D \times D) \\ (s, t) \qquad\qquad\quad \mapsto s \times t, \end{cases}$$

and show that this is a strong contraction for a wide class of uniformities $\mathscr{G}^*_{\mathscr{A}}$ on $\mathscr{I}_\sigma(D)$.

DEFINITION 6.1. Let $A \subset \mathscr{P}(D^o)$ be a filter basis, then

(a) $\mathscr{A}$ is *p-closed* iff $\forall A \in \mathscr{A}\ \forall t \in \mathscr{I}(D^o)$: $A \times t \cup t \times A \subset A$;

(b) $\mathscr{A}$ is *absorbing* iff $\forall s, t \in \mathscr{I}(D^o)\ \forall A \in \mathscr{A} \exists m \in \mathbb{N}$: $s^m \times t \cup t \times s^m \subset A$;

(c) $\mathscr{A}$ is *suitable* iff $\mathscr{A}$ is p-closed as well as absorbing.

Suppose for the moment that $\mathscr{A}$ is determined by a ranking function r which has the additional property that

$$r(a, b) > \max\{r(a), r(b)\}$$

holds, whenever $a, b \in D$; this is the case, e.g., in [6]. Hence,

$$\mathscr{A} = \{\{r \geqq n\}; \{r \geqq n\} \neq \varnothing, n \in \mathbb{N}\}$$

holds. It is immediate that $\mathscr{A}$ is suitable.

Now fix for the general case an absorbing filterbase $\mathscr{A}$. We will show now that p is a strong contraction for the uniformity $\mathscr{G}^*_{\mathscr{A}}$, where the attractor on $\mathscr{G}_{\mathscr{A}}$ is the identity. For the sake of not duplicating arguments, we will restrict our attention to $p_s = \lambda t \cdot p(s, t)$ for some arbitrarily chosen $s \in \mathscr{I}_\sigma(D)$. Since the conditions (A-1) and (A-2) are obvious for the identity on $\mathscr{G}_{\mathscr{A}}$, and since (A-3) holds because $\mathscr{A}$ is suitable, we will have to focus on the conditions (A-4) and (A-5). Since $\mathscr{A}$ is absorbing, we may find for $A \in \mathscr{A}$, $t \in \mathscr{I}_\sigma(D)$ an exponent m such that

$$(s^o)^m \times t^o \subset A.$$

Consequently,

$$(s^o)^m \times (s^o \triangle s^o \times t^o) \subset A,$$

which is equivalent to

$$(p_s^m(t), p_s^{m+1}(t)) \in G_A.$$

(Here we have used the identity

$$P \times (Q \triangle R) = (P \times Q) \triangle (P \times R),$$

which is easily established using the indicator function 1_M of a set M:

$$1_M(g) := \textit{if } g \in M \quad \textit{then } 1 \textit{ else } 0 \textit{ fi}$$

by observing the identities

$$1_{M \times N}(g, h) = 1_M(g) \cdot 1_N(h)$$

and

$$1_{M \triangle N}(g) = |1_M(g) - 1_M(h)|.$$

The argumentation above shows that p_s satisfies (A-4), and the identity

$$((s^o)^m \times t_1^o) \triangle ((s^o)^m \times t_2^o) = (s^o)^m \times (t_1^o \triangle t_2^o)$$

immediately yields condition (A-5).

Acknowledgment. Benno Fuchssteiner, Thomas Landes, and Marcel Erné provided some insight into fixed points. The referees' comments helped to clarify the presentation and to remove some obscure points.

REFERENCES

[1] J. W. DE BAKKER AND J. N. KOK, *Towards a uniform topological treatment of streams and functions on streams*, in Proc. 12th International Conference on Automata, Languages, and Programming, W. Brauer, ed., Lecture Notes in Computer Science 194, Springer-Verlag, Berlin, New York, 1985, pp. 140–148.

[2] C. CASTAING AND M. VALADIER, *Convex analysis and measurable multifunctions*, Lecture Notes in Mathematics 580, Springer-Verlag, Berlin, New York, 1977.

[3] E. HEWITT AND K. STROMBERG, *Real and Abstract Analysis*, Springer-Verlag, Berlin, New York, 1969.

[4] J. L. KELLY, *General Topology*, Van Nostrand Reinhold, New York, 1955.

[5] TH. LANDES, *The Banach fixed point principle in uniform spaces*, preprint, Department of Economics, University of Paderborn, September, 1986.

[6] D. MACQUEEN, G. PLOTKIN, AND R. SETHI, *An ideal model for recursive polymorphic types*, Inform. and Control, 71 (1986), pp. 95–130.

[7] J. STOY, *Denotational Semantics*, MIT Press, Cambridge, MA, 1977.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 990-1012, October 1989

010

THE DISTRIBUTED FIRING SQUAD PROBLEM*

BRIAN A. COAN†, DANNY DOLEV‡, CYNTHIA DWORK§¶, AND LARRY STOCKMEYER§

Abstract. The distributed firing squad problem is defined in the context of a synchronous distributed system where the correct processors operate in lock-step synchrony but do not share a global clock. If one or more correct processors receive a command to start a firing squad synchronization, then at some future time all correct processors must "fire" (formally, enter a special state) at exactly the same step. For various fault models, upper and lower bounds are proved on the number of faulty processors that can be tolerated and on the number of rounds of communication required between the reception of the start command and firing. For example, if a firing squad protocol is resilient to t fail-stop faults, then at least $t+1$ rounds are necessary and sufficient. For the case of Byzantine faults with authentication where the faulty processors can take steps in between the synchronous steps of the correct processors, the firing squad problem can be solved in $t+5$ rounds, provided that $n>3t$, where n is the number of processors and t is the number of faults, and the problem cannot be solved at all if $n \leqq 3t$. Moreover, in the case that $n \leqq 3t$, the impossibility of a firing squad protocol holds even for a weaker "timing fault model" where all processors generate messages correctly according to the protocol, but the faulty processors can affect the system by slightly slowing down or speeding up messages.

Key words. firing squad problem, Byzantine generals problem, synchronization, coordination, fault tolerance, distributed computing

AMS(MOS) subject classifications. 68M10, 68M15, 68Q

1. Introduction.

Many fault-tolerant distributed algorithms assume a *synchronous system*, in which processing is divided into synchronous unison "steps" separated by rounds of message exchange (see, e.g., [11], [20], [25]). A message sent at step s from a correct processor p to a correct processor q is received by q at step $s+1$. This assumption is motivated by the impossibility results of [15] and [8], which show that if the system is asynchronous then there is no protocol for distributed agreement tolerant to even one benign processor failure. There are various ways to maintain synchronous steps in an unreliable distributed system. For example, one can have hardware that sends a periodic signal to all processors. Another common assumption is that all processors begin the algorithm simultaneously, i.e., at the same step. Typically, however, an algorithm is executed in response to a request from some specific processor that may in turn be responding to some external request. If the given processor is correct then all correct processors learn of the request simultaneously, so they can indeed begin the algorithm in unison. However, if the processor is faulty then the correct processors may learn of the request at different steps.

* Received by the editors July 23, 1986; accepted for publication (in revised form) October 17, 1988. A preliminary and abridged version of this paper appeared in the Proceedings of the 17th ACM Symposium on Theory of Computing, Providence, Rhode Island, 1985.

† Bell Communications Research, MRE 2P-252, 445 South St., Morristown, New Jersey 07960. This work was performed in part while the author was at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts. It was supported by the National Science Foundation under grant DCR-8302391, by the U.S. Army Research Office under contracts DAAG29-79-C-0155 and DAAG29-84-K-0058, and by the Advanced Research Projects Agency of the Department of Defense under contract N00014-83-K-0125.

‡ IBM Research Division, K53/802, 650 Harry Road, San Jose, California 95120, and Computer Science Department, Hebrew University, Jerusalem, Israel. This work was performed in part while the author was a Batsheva de Rothschild Fellow.

§ IBM Research Division, K53/802, 650 Harry Road, San Jose, California 95120.

¶ This work was performed in part while the author was a Bantrell Postdoctoral Fellow at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts.

In this paper we justify the design assumption of simultaneous starts. Specifically, we provide algorithms to solve the associated synchronization problem that we call the *distributed firing squad* problem. An algorithm for the distributed firing squad problem has two properties:

(1) If any correct processor receives a message to start a firing synchronization, then at some future time all correct processors will "fire" (formally, enter a special state); and

(2) The correct processors all fire at exactly the same step.

Our principal results, which are summarized in this Introduction, are for the case in which the processors, although operating in lock-step, are not assumed to share a common view of the current "global" time. We also present a simpler algorithm for the case in which such a common view is assumed.

The two complexity measures we study are *fault tolerance*, the maximum number of faulty processors that can be tolerated, and *time*, the maximum number of rounds of message exchange taken by the algorithm, starting with the step at which some correct processor receives a message to start a firing synchronization and ending with the step at which all correct processors fire. We are also interested in the *communication complexity* of an algorithm, that is, the total number of message bits sent by correct processors, but only to the extent of distinguishing polynomial from exponential communication complexity. Below, n denotes the number of processors in the system; t denotes the maximum number of faults that can be tolerated by a particular algorithm, and any such algorithm is said to be *t-resilient.*

In the case of fail-stop faults (the most benign type of fault usually studied, in which a faulty processor follows its algorithm correctly but simply stops at some point), it is easy to find a t-resilient distributed firing squad algorithm for any number $t \leqq n$ of faults that halts in $t+1$ rounds. This was observed independently by Burns and Lynch [3]. By reducing the Weak Byzantine Agreement (WBA) problem to the distributed firing squad problem, we can use a lower bound of Lamport and Fischer [18] on the time complexity of the WBA problem to show that any t-resilient algorithm for the distributed firing squad problem requires $t+1$ rounds for fail-stop faults, and therefore also for more malicious types of faults. Thus, the situation for fail-stop faults is well understood. Burns and Lynch [4] give a distributed firing squad algorithm for the case of Byzantine faults without authentication (the most serious type of fault usually studied, where faulty processors can exhibit arbitrary behavior); we say more about this case below. The main results in this paper concern Byzantine faults with authentication. Byzantine processors can exhibit arbitrary behavior, but we assume that every processor can sign messages in such a way that the signature of a correct processor cannot be forged by a faulty processor (see, e.g., [11]), and the signatures are common knowledge.

In trying to determine the maximum fault tolerance of the distributed firing squad problem in the authenticated Byzantine case, we found it necessary to distinguish between several types of faulty behavior, since these distinctions affect the fault tolerance. In *rushing*, a Byzantine faulty processor can receive, process, and resend messages "between" the synchronous steps of other processors. Figure 1(a) shows a normal communication round involving three correct processors A, B, and C, with A sending messages to B and C. Fig. 1(b) shows a similar round in which processor C is faulty, takes a step between the steps of the correct A and B, computes its response to the message it received from A, and then "rushes" this response to B in the same round. A special case of rushing is the *timing fault* model, where faulty processors never fail and always follow their algorithms correctly, but may take steps at irregular

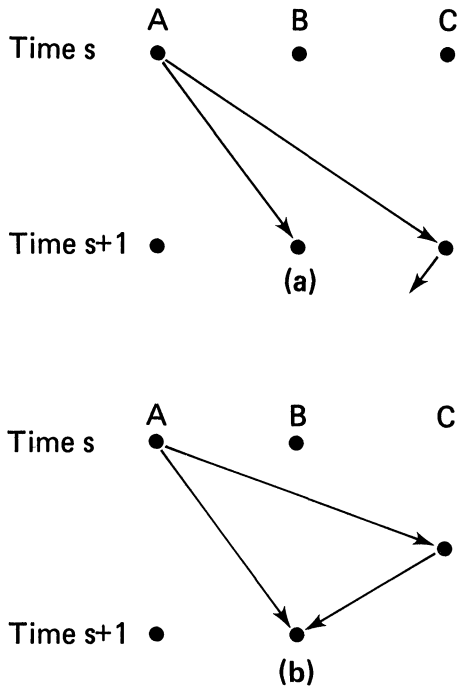


FIG. 1. (a) *A communication round with three correct processors. For simplicity, only the messages sent by A at time s are shown.* (b) *The processor C rushes.*

times and may experience slight delays or accelerations in communicating with the other processors. Rushing and timing faults are realistic types of faults whenever there is sufficient uncertainty in message transmission time. The length of a communication round must be chosen as large as the maximum possible transmission time between correct processors, but if a message happens to be delivered to a faulty processor in time less than this maximum, the faulty processor has the opportunity to rush.

We must also distinguish the case where faulty processors can sign messages using the signature functions of other faulty processors, which we call *collusion*, and the case where a faulty processor has only its own signature function. Collusion is unlikely to occur as a result of a random failure, but it could occur if the faulty processors were controlled by a malevolent intelligence that allowed faulty processors to share signature keys.

Table 1 summarizes our results and the results of [4] for the different fault models. Each entry gives $n_{\min}$, the smallest number n ($n \geqq 2$) of processors for which there exists a t-resilient distributed firing squad algorithm ($t \geqq 1$). Unless otherwise indicated, all algorithms require at most $t+c$ rounds, where $c \leqq 5$ is a constant independent of n and t, and the total number of bits of communication sent by correct processors is polynomial in n. In proving lower bounds on the minimum n, we make the usual assumption that the receiver of a message knows the identity of the sender; however, this assumption is not needed by our algorithms because the necessary deductions about the sender's identity can be made in the fault models we consider.

There are several interesting things to note about these results. First we should emphasize that the lower bound $n \geqq 3t+1$ holds for the timing fault model in which *all processors follow the algorithm correctly.* The only way faulty processors can affect the system is by taking steps at irregular times and unknowingly delaying and speeding up certain messages by small amounts. Second, even though our bounds on the

TABLE 1

Fault	$n_{\min}$ for t-resiliency	Remarks
fail-stop	t	1
Byzantine with authentication		
no rushing, no collusion	t	1
collusion, no rushing	$5t/3 < n_{\min} \leqq 2t+1$	2
rushing, no collusion	$3t+1$	
rushing and collusion	$3t+1$	
timing faults	$3t+1$	
Byzantine without authentication	$3t+1$	3

Remarks. (1) *Due independently to Burns and Lynch* [3]. (2) *Lower bound* $5t/3 < n_{\min}$ *proved only for* $t \geqq 3$; *algorithm with* $2t+1$ *processors takes* $2t+1$ *rounds.* (3) *Algorithm due to Burns and Lynch* [4]; *algorithm uses either exponential communication or more than* $t+O(1)$ *rounds. Except as noted in* 2 *and* 3, *running time of all algorithms is* $t+c$, $c \leqq 5$, *and communication complexity is polynomial in n.*

minimum n in the case of collusion but no rushing are presently not tight, the bounds are sufficient to show that collusion does decrease fault tolerance when compared to the case of no collusion and no rushing, and rushing alone admits less fault tolerance than collusion alone. The distinction thus shown between these three fault models is (to us) an unexpected result of this work.

Burns and Lynch [4] solve the distributed firing squad problem in the unauthenticated Byzantine case essentially by adapting an agreement protocol. Since all known unauthenticated agreement protocols either use exponential communication, use more than $t+O(1)$ rounds, or require $n > 8t$ [2], [5], [9], [10], [20], [24], their distributed firing squad solution has the same property. Recently, Moses and Waarts [24] have devised a new unauthenticated agreement protocol; together with the work of Burns and Lynch, this gives an unauthenticated distributed firing squad protocol using polynomial communication, $n > 8t$ processors, and the optimal time $t+1$. By using signatures, we achieve polynomial communication, time $t+5$, and the maximum resiliency $t = \lfloor (n-1)/3 \rfloor$. Our lower bounds $n > 3t$ ($n > 5t/3$) for rushing (collusion) suggest that the approach of directly adapting agreement protocols to the distributed firing squad problem will not work in the authenticated case, since there are authenticated agreement protocols tolerant to any number of failures [11]. For the same reason, the distributed firing squad problem seems to be different from the clock synchronization problem studied in [16], [19], [21], [26], where the object is to bring the clocks of correct processors "close" together: in the authenticated Byzantine case, there is a clock synchronization algorithm tolerant to any number of failures [16]. Our problem is also different from the version of the firing squad problem that was proposed in the late 1950s [23]. That version of the problem was interesting because the processors were finite state machines which were connected in a linear array so each processor could count only to some fixed constant independent of n; however, faults were not considered. In our version of the problem, the difficulty arises not from limitations on the processors or communication network (we assume a completely connected system of powerful processors) but rather from the possibility of processor and timing faults.

In § 2 we give definitions. Section 3 contains results (firing squad algorithm and lower bound) for the case of no collusion and no rushing, § 4 gives results for rushing and timing faults, and § 5 considers the case of collusion but no rushing. In § 6 we mention some related results, such as the application of firing squad ideas to the

problem of Byzantine Agreement in the case that the processors do not all start at the same round, and results concerning the distributed firing squad problem where the processors share a common view of global time.

2. Definitions. For simplicity we give definitions for a single occurrence of a firing squad synchronization. Let $p_1, p_2, \cdots, p_n$ denote the processors in the system. For technical reasons we introduce another "processor" w, whose only purpose is to start a synchronization; w does not receive messages from the p_i's. In reality w might be another process running within one of the p_i. Formally a distributed firing squad algorithm is specified by an infinite set of messages M and for each processor p_i an infinite set of states Q_i, a state transition function σ_i, and a sending function β_i, where

$$\sigma_i : Q_i \times M^{n+1} \to Q_i,$$
$$\beta_i : Q_i \times M^{n+1} \to M^n.$$

The inputs to σ_i and β_i are the current state and an $(n+1)$-tuple of received messages, one from each processor $p_1, \cdots, p_n, w$. The function σ_i gives the new state, and β_i gives an n-tuple of messages $(m_1, \cdots, m_n)$ such that m_j is sent to p_j for each j. There are special messages $\varnothing$, the null message, and "Awake", the awake message, which is sent by w to start a synchronization. For each i there are states q_0 and q_f in Q_i, the *quiescent state* and the *firing state*, respectively, where $q_0 \neq q_f$. In addition,

$$\sigma_i(q_0, \varnothing, \cdots, \varnothing) = q_0,$$
$$\beta_i(q_0, \varnothing, \cdots, \varnothing) = (\varnothing, \cdots, \varnothing).$$

We introduce the concept of *global time* as an expositional convenience. The individual processors have no knowledge of global time. We assume that processors take steps at global times specified by nonnegative real numbers. A *run* is specified by giving, for each processor $p_1, \cdots, p_n$, w (including both correct and faulty processors), a list of nonnegative real numbers that specifies the times at which the processor takes steps. A message sent from a processor p to a processor q at time s is received by q at time s', where s' is the smallest $s' > s$ such that q takes a step at s'. (If q receives more than one message from some p at some step, then the message sent at the latest time is used by the transition functions; since this occurs only if either p or q is faulty, this convention is not critical.) Whenever w takes a step, it sends the awake message to some (possibly empty) subset of the p_i's and it sends the null message to the rest.

A processor p_i is *correct* in a run R if

(1) p_i takes its first step at time 0 in state q_0 receiving messages $(\varnothing, \varnothing, \cdots, \varnothing)$, and thereafter takes steps at successive integer times 1, 2, 3, $\cdots$;

(2) p_i executes its algorithm (transition functions) correctly; and

(3) if authentication is assumed (see below), then no other processor p_j signs any message using the signature function of p_i.

A run R is *active* if some correct processor receives a non-null message at some step; define $awake(R)$ to be the earliest such time (necessarily integer). If a correct p_i receives a non-null message for the first time at time s, we say that p_i *awakens* at time s. Define $fire_i(R)$ to be the time of the first step in R during which p_i makes a transition into state q_f (undefined if p_i does not enter q_f).

A distributed firing squad algorithm is *t-resilient* with respect to a given type of faulty behavior if for any active run R in which at most t of the processors $p_1, \cdots, p_n$ are faulty and in which the faulty processors conform to the given type of faulty behavior, there is a (necessarily integer) time $fire(R) \geqq awake(R)$ such that $fire_i(R) = fire(R)$ for all i such that p_i is correct in R. The *time complexity* of the algorithm is

the maximum of $fire(R) - awake(R)$ over all such runs R. (Note that w is not counted among the t faulty processors no matter how it behaves.)

Since the definitions above consider a processor to be faulty throughout a run if it fails at any time in the run, the definitions would seem to allow a scenario S where some processor p_i first fails after time $fire(R)$; thus, the definitions do not require p_i to fire with the other processors even though it is actually "correct" throughout execution of the algorithm. It is sufficient to note, however, that any such scenario can be modified to a scenario S', where p_i is correct throughout the entire (infinite) run, and all processors behave exactly as they do in S from time 0 through time $fire(R)$. In the modified S', p_i violates the definition of correctness. We find it convenient, both for definitions and for proofs of correctness, to consider a processor to be either correct or faulty throughout an entire run rather than to define the first time when a faulty processor actually exhibits faulty behavior.

We now define various types of faulty behavior. A faulty processor p_i is *fail-stop* if it operates as a correct processor up to some time s, at time s some nonempty subset of the messages p_i is supposed to send are replaced by null messages, and for all subsequent steps p_i sends only null messages. A processor is *Byzantine* if its behavior can deviate in any way from the behavior specified by its transition functions. A special case of Byzantine faultiness is Byzantine faults with authentication, where each processor p can sign messages using a private signature function E_p in such a way that the signature of a correct processor cannot be forged by any other processor. In this case, if processor q receives a message $E_p(m)$ from a third processor r, then q knows that if p is correct then p actually sent the message $E_p(m)$ at some previous time. We also let E_i denote the signature function of processor p_i. A Byzantine faulty processor *rushes* if it takes some step at a noninteger time. In this case, messages to and from faulty processors may take less than one round to be delivered. Faulty processors p and q *collude* if p signs a message using the signature function of q. In this case q is considered faulty, even though the messages it sends may be correct.

Finally we define the *timing fault model.* Runs in this model have the following properties:

(1) all processors execute the algorithm correctly;

(2) correct processors take steps at times $0, 1, 2, \cdots$;

(3) faulty processors take steps at times $\frac{1}{2}, \frac{3}{2}, \frac{5}{2}, \cdots$;

(4) messages between two correct processors or between two faulty processors take one unit of time to be delivered; and

(5) messages between a correct and a faulty processor take either $\frac{1}{2}$ unit of time or $\frac{3}{2}$ units of time to be delivered.

It is not hard to see that the timing fault model is a special case of authenticated Byzantine faultiness with rushing (but no collusion). The model with rushing can simulate a delivery time of $\frac{3}{2}$ simply by having a Byzantine processor either delay sending or delay receiving the message. For example, if in the timing fault model the faulty processor p sends a message m at time $\frac{3}{2}$ which the correct q should receive at time 3, then in the model with rushing the (now Byzantine) p simply holds m and sends it to q at time $\frac{5}{2}$. Therefore, giving an algorithm for the model with rushing yields a result for both models, as does proving a lower bound on n for the timing fault model.

The *communication complexity* of an algorithm is the maximum total number of message bits sent by correct processors, between the time when a correct processor first awakens and the time when all correct processors fire. The communication complexity is expressed as a function of n. Of course, there is no way to bound the number of bits sent by Byzantine faulty processors. For each of our algorithms it is

easy to see that exceedingly long messages sent by faulty processors never cause a correct processor to send exceedingly long messages. This is true because each correct processor has, at each step of the algorithm, a bound on the length of messages it expects to receive. Once the length of a message exceeds that bound, the message can be ignored.

As the firing squad problem is defined above, any single Byzantine faulty processor can initiate a firing synchronization. One way to limit the ability of faulty processors to start extraneous synchronizations in models with authentication is to have w sign the awake message with its own unforgeable signature. A correct processor considers as null any received message that does not contain w's signature. The necessary changes to our algorithms are trivial (in algorithms that count the number of signatures on a message, the signature of w is not counted). This still allows a faulty w to start extraneous synchronizations, but this is unavoidable in any model where we allow a single processor or an external agent to start a synchronization.

Processors as defined above are deterministic. Coan and Dwork [6] have studied probabilistic protocols for firing squad synchronization and have found that probabilistic protocols are essentially no better than deterministic ones for the firing squad problem.

3. No rushing and no collusion.

3.1. Upper bound. We begin with a simple algorithm that tolerates any number of fail-stop or authenticated Byzantine faults. It does not tolerate rushing, timing faults, or collusion. This algorithm was discovered independently by Burns and Lynch [3]. The basic idea is that since any processor, faulty or otherwise, can add at most one signature per round, we can use the number of signatures on a message as a clock, giving a lower bound on the time elapsed since the protocol was initiated. A correct processor fires as soon as it knows that at least $t+1$ rounds have elapsed. The details of the algorithm and its proof of correctness are similar to those of the Dolev–Strong algorithm for authenticated Byzantine agreement [11].

THEOREM 3.1. *In the model with authenticated Byzantine failures (but no rushing or collusion) there is a t-resilient distributed firing squad algorithm for any number $n \geqq t$ of processors. The algorithm has time complexity of $t+1$ rounds, and it uses an amount of communication polynomial in n.*

Proof. Each processor p participating in the protocol has a private clock c_p that is completely under the control of p. Initially, $c_p = -1$. A message m is *proper* if it has the form

$$m = E_{i_1}(E_{i_2}(\cdots E_{i_k}(\text{Awake})\cdots))$$

where E_{i_j} is the signature function of p_{i_j}, and the k signatures are by distinct processors. The *length* of the proper message m, denoted $|m|$, is the number of signatures appearing in m (i.e., k above). The awake message has length 0. A proper message m is *acceptable* to p if and only if $|m| > c_p$. A proper message m is *new* to p if and only if p's signature does not appear in m.

Upon first awakening, p sets its clock to $|m|$, where m is any acceptable message of maximum length received by p. If $c_p = t+1$, then p fires; otherwise, p signs m and broadcasts the result $E_p(m)$. If all messages received by p in this first step are unacceptable, then p sets its clock to 0 and broadcasts $E_p(\text{Awake})$.

At each subsequent step, if p receives any acceptable message, p arbitrarily chooses one such message m of maximum length and sets c_p to $|m|$. If $c_p \geqq t+1$, then p fires; if $c_p < t+1$ and m is new to p, then p signs m and broadcasts the result $E_p(m)$. If no acceptable message is received, p increments c_p by one. Again, if $c_p \geqq t+1$ then p fires.

This completes the description of the protocol for a correct p.

LEMMA 3.1.1. *For all k with $k \geqq 0$ and for any message m, at least k rounds are required to add k distinct signatures to m.*

Proof. Each processor knows at most one signature function and there is no rushing. □

LEMMA 3.1.2. *In any execution of the protocol resulting in a firing, all correct processors fire simultaneously.*

Proof. Let r be the earliest step at which some correct processor fires, and let p be a correct processor that fires at step r. We will show that for all correct processors q, $c_q \geqq t+1$ by the end of step r, so q fires at r.

There are two cases to consider, according to whether or not p receives an acceptable message at step r. If p receives an acceptable message m at step r, then m has at least $t+1$ signatures. Without loss of generality, let $m = E_k(E_{k-1}(\cdots E_1(\text{Awake})\cdots))$, where $k = |m| \geqq t+1$. Clearly, if p_k is correct then all correct processors receive m at r, and so all correct clocks have value at least $t+1$ by the end of step r. If p_k is faulty, then let p_i be the last correct processor whose signature appears in a substring $m_i = E_i(\cdots E_1(\text{Awake})\cdots)$ of m, and let $s-1$ be the step at which m_i is sent by p_i. By Lemma 3.1.1 at most one signature can be added to m_i in each step, so $r - s \geqq |m| - |m_i|$. Furthermore, since p_i is correct, the clocks of all correct processors have value at least $|m_i|$ by the end of step s. Since the clock of a correct processor is incremented by at least 1 at each step, by the end of step r the clocks of all correct processors will have value at least

$$|m_i| + (r-s) \geqq |m_i| + (|m| - |m_i|) = |m| \geqq t+1$$

so all correct processors will fire at step r.

In the second case where p receives no acceptable message at step r, let i be such that step $r-i$ is the last step at which p broadcasts. (Since this happens when p first awakens, i is well defined.) Let $E_p(m)$ be the message broadcast by p at $r-i$. Since this message has length $|m|+1$ and since all correct processors receive this message at step $r-i+1$, by the end of step $r-i+1$ all correct processors have clock value at least $|m|+1$. Therefore, the clocks of all correct processors have value at least $|m|+i$ by the end of step r. We now want to argue that p increments its clock by exactly 1 at each step j with $r-i+1 \leqq j \leqq r$. Suppose otherwise that p increments its clock by more than 1 at step j, and let m' be the acceptable message received at step j which causes this increment. Since p receives no acceptable message at step r, we must have $j < r$. If m' is not new to p at step j, then using Lemma 3.1.1 as in the preceding paragraph, it is easy to show that m' could not cause p to increment its clock by more than 1. If m' is new to p, then by definition of the algorithm p would broadcast at step j, contradicting the choice of i. Therefore, such a j and m' cannot exist. Having bounded p's clock increment at each step j with $r-i+1 \leqq j \leqq r$, it follows that $c_p = |m|+i$ at the end of step r. Since p fires at r, we have $|m|+i \geqq t+1$. Thus all correct processors fire at step r. □

LEMMA 3.1.3. *Let R be any active run and let s be the earliest time when some correct processor p awakens in R. Then p fires at time $s+t+1$ or earlier.*

Proof. Upon awakening, p sets its clock to some value $c_p \geqq 0$. At every step c_p is incremented by at least one, and p fires when $c_p \geqq t+1$. □

It is now easy to complete the proof of Theorem 3.1. By Lemma 3.1.3, if any correct processor awakens there is a firing, and by Lemma 3.1.2 all correct processors fire simultaneously. Lemma 3.1.3 also implies that the time complexity of the algorithm

is at most $t+1$ rounds. Furthermore, at each step each correct processor broadcasts at most one message. Since this message is proper, it requires only polynomial in n bits. Thus the communication required is polynomial in n. □

Obviously, the algorithm of Theorem 3.1 also works for fail-stop faults: instead of signing a message m with the signature function E_p, the processor p simply attaches its name to m.

3.2. Lower bound. We now show that the time complexity of this algorithm is optimal by reducing the Weak Byzantine Agreement problem (WBA) [17] to the distributed firing squad problem. Optimality follows from the fact that WBA requires at least $t+1$ rounds [18].

In the WBA problem, all processors start the algorithm at the same global time (say, time 0), and each processor has a binary initial value. By maintaining a counter, all correct processors have a common notion of global time. A protocol solves WBA if (1) every correct processor eventually reaches a decision; (2) no two correct processors reach different decisions; and (3) if all initial values are the same, say v, and there are no failures, then v is the value decided. The following result shows that WBA reduces to distributed firing squad at no cost in running time.

THEOREM 3.2. *Let A be a distributed firing squad algorithm that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that requires k rounds between awakening and firing in the execution in which all the processors awaken simultaneously and no failure occurs (note that k is unique since the system is completely deterministic in this case). Then there exists an algorithm for* WBA *that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that always halts in k rounds.*

Proof. Consider an instance of WBA in which processor p_i has initial value v_i. If $v_i = 0$, then p_i begins simulating A at time 0. That is, p_i acts as if it received the awake message from w and null messages from the rest. If $v_i = 1$, then p_i begins simulating A at time 1. That is, p_i sends null messages during the first round and acts as though it received the awake message from w at time 1 (p_i could receive non-null messages from other processors at time 1 in this case if other processors had initial value 0). If the simulation of A causes p_i to fire at time k or earlier, then p_i decides 0 at time k; otherwise, p_i decides 1 at time k.

Correctness of A immediately implies that all correct processors decide on the same value, since either all correct processors simulate a firing at a time $\leqq k$ or none do. If all processors begin with value 0 and there are no failures, then by choice of k each processor will simulate a firing at time k, so the decision will be 0. However, if all begin with 1 and there are no failures, then all processors will simulate a firing at time $k+1$, so the decision will be 1. □

COROLLARY 3.3. (1) *Let $t \leqq n-2$. Any distributed firing squad algorithm resilient to t fail-stop faults requires at least $t+1$ rounds. Moreover, this is true even if the order in which processors are sent to in a round is fixed a priori. It is also true even in all executions in which all processors are correct.*

(2) *Any distributed firing squad algorithm resilient to t unauthenticated Byzantine faults requires at least $3t+1$ processors.*

Proof. The proof is immediate from the preceding theorem and the corresponding bounds for WBA [13], [14], [17], [18], [22]. □

Remark. To close the gap between Theorem 3.1 and Corollary 3.3(1) for $n-1 \leqq t \leqq n$, note that if we take $t = n-2$ in the algorithm of Theorem 3.1, then the algorithm is in fact n-resilient. If there is only one correct processor p, then p will fire within $t+1$

$(=n-1)$ steps after awakening, since c_p is incremented by at least 1 at each step. If there are no correct processors, then there is nothing to prove.

In the next section the lower bound of Corollary 3.3(2) is strengthened to hold in the model with authentication and rushing.

4. Rushing and timing faults.

4.1. Upper bound. This section contains tight bounds on fault tolerance for timing faults and authenticated Byzantine faults with rushing. The following result gives the principal algorithm of the paper.

THEOREM 4.1. *In the model with Byzantine failures and authentication, in which faulty processors can both rush and collude, there is a t-resilient distributed firing squad algorithm requiring* $t+5$ *rounds,* $n \geq 3t+1$ *processors, and communication polynomial in n.*

Before giving the details of the algorithm and its proof of correctness, we begin with an informal discussion of the principal ideas. Our protocol is composed of a set of identical subprotocols executed independently and in parallel. A processor initiates a subprotocol by broadcasting its signature. Let p be an arbitrary (possibly faulty) initiator, and consider a set of processors all receiving p's signature at the same step. In some sense these processors are synchronized, in that they share a common idea of when they first heard from p, although no processor in the set knows which other processors are in the set. If the set of synchronized processors is sufficiently large, then *because they are synchronized* these processors can run an agreement protocol similar to the Dolev and Strong protocol [11] that assumes synchronous start. The processors are essentially agreeing on the members of the set. If the agreed upon set is sufficiently large, then correct processors will order a firing. In particular, a correct processor in the set orders a firing only if there are at least $n-t \geq 2t+1$ processors in the agreed upon set. Of these, at least $t+1$ are correct, synchronized processors. Thus a correct processor orders a firing only if at least $t+1$ correct processors do so simultaneously. Now, consider a processor q receiving at least $t+1$ commands to fire. Since q knows at least one of these messages is from a correct processor, it knows at least $t+1$ are. Thus q knows that every processor receives at least $t+1$ commands to fire, and therefore that every processor knows every processor has received these commands, and so on. In short, it becomes *common knowledge* that every processor has received $t+1$ commands to fire, so it is safe to fire.

Proof of Theorem 4.1. As stated above, the protocol is composed of a set of identical subprotocols executed independently and in parallel. Specifically, as each correct processor awakens it initiates a *core protocol.* If the core protocol is successfully completed, then the correct processors fire upon completion. An execution of the core protocol initiated by a correct processor will complete successfully, unless a firing occurs earlier due to the completion of a different execution of the core protocol. An execution of the core protocol initiated by a faulty processor may not cause a firing, but if it does then all correct processors fire simultaneously. (Thus, it would be sufficient to have any $t+1$ processors initiate the core protocol.) In the following, if a correct processor receives the same message at different times, all receptions but the first are ignored; this prevents a faulty processor from doing any damage by taking a message that was broadcast by a correct processor and resending it at a later time.

A processor p initiates a core protocol by broadcasting its signature, $E_p(p)$. Each processor (including p itself) that receives $E_p(p)$ signs it and broadcasts it. Each processor q then attempts to form a *core for p,* that is, a list of the form

$$\langle E_{i_1}(E_p(p)), \cdots, E_{i_k}(E_p(p))\rangle$$

where $k \geqq n-t$ and each of the k copies of $E_p(p)$ is signed by a distinct processor. The signatures $E_{i_1}, \cdots, E_{i_k}$ *belong* to the core, and the core *contains* these signatures. Intuitively, a core is a set of processors which claim to have received $E_p(p)$ at the same time.

A *notarized core for p* is a list of the form

$$\langle E_{i_1}(C_1), \cdots, E_{i_k}(C_k)\rangle$$

where $k \geqq n-t$ and the C's are (possibly different) cores for p, each signed by a distinct processor.

We now describe how a processor q participates in the core protocol initiated by p. (Processor p participates in the core protocol initiated by itself.) Let s be the global time when q receives $E_p(p)$. Then q tries to form a core for p at time $s+1$. This is done by looking for a set of messages $\{E_{i_1}(E_p(p)), \cdots, E_{i_k}(E_p(p))\}$ received at time $s+1$ where $k \geqq n-t$ and where each of the k copies of $E_p(p)$ is signed by a distinct processor. This is the only time at which q tries to form a core for p, and q includes in the core only messages received at time $s+1$. If q forms a core then q includes in the core all messages of the form signature$(E_p(p))$ received at time $s+1$. If a core is formed, q signs it and broadcasts it. Processor q also tries to form a notarized core for p at time $s+2$. This is the only time when q tries to form a notarized core for p. A notarized core, if formed, contains all messages of the form signature(core for p) received at time $s+2$. If a notarized core N is formed by q at this step, then N is considered to have been "received" at this step. Starting with the second step after $E_p(p)$ was received, each correct processor q does the following (regardless of whether or not q formed a core for p or a notarized core for p).

If q receives message m, q checks if m is *acceptable* in the following sense:

(1) $m = E_{j_1}(E_{j_2}(\cdots E_{j_k}(N)\cdots))$, where N is a notarized core for p and each of the k signatures ($k \geqq 0$) is distinct (m is said to have *length* k, denoted $|m|$);

(2) q's signature belongs to at least $n-2t$ of the cores in N (we say that q *supports* N); and

(3) q first received $E_p(p)$ $k+2$ steps back (this condition implies that at any given step of q, messages of only one particular length are acceptable).

An acceptable message m as in (1) is *new* to q if none of the signatures $E_{j_1}, E_{j_2}, \cdots, E_{j_k}$ is by q. If q finds one or more messages of length k that are new and acceptable, q chooses one such message m arbitrarily and broadcasts $E_q(m)$, ignoring the rest. Finally, if q receives an acceptable message m of length $t+1$, then q signs and broadcasts "fire$_p$". A correct processor fires at step f if and only if at step f it receives at least $t+1$ commands "fire$_p$" signed by different processors.

Lemmas 4.1.1–4.1.4 show that the core protocol causes a firing if the initiator is correct. For these lemmas, let p be a correct processor initiating a core protocol at time r.

LEMMA 4.1.1. *At time $r+2$ all correct processors can form a core for p containing the signature of every correct processor, and at time $r+3$ all correct processors can form a notarized core for p.*

Proof. Since p is correct, all correct processors receive $E_p(p)$ at time $r+1$. All correct processors q broadcast $E_q(E_p(p))$ at time $r+1$, and these messages are received at time $r+2$. Since there are at least $n-t$ correct processors, every processor receives at least $n-t$ messages of the form $E_q(E_p(p))$ signed by distinct processors. Thus all correct processors can form a core at time $r+2$. Furthermore, since a correct processor puts all messages $E_i(E_p(p))$ received into the core, for every correct processor q the message $E_q(E_p(p))$ appears in the cores formed by the correct processors.

A similar argument shows that every correct processor can form a notarized core for p at time $r+3$. □

LEMMA 4.1.2. *Let N be any notarized core for p. Then every correct processor q supports N.*

Proof. A notarized core contains at least $n-t$ cores, each signed by distinct processors, so at least $n-2t$ of the cores contained in N were formed by correct processors. By Lemma 4.1.1 all these $n-2t$ cores contain the signature of every correct processor. □

LEMMA 4.1.3. *For all i, $0 \leqq i \leqq t$, at time $r+3+i$ at least one correct processor receives a new acceptable message of length i.*

Proof. The proof is by induction on i.

Basis $i=0$. By the previous two lemmas every correct processor forms a notarized core which it supports at time $r+3$. By convention, this notarized core is a new acceptable message "received" at time $r+3$.

Assume the lemma is true inductively for $i-1$ ($i \geqq 1$). Thus at time $r+3+(i-1) = r+i+2$ some correct processor receives a new acceptable message of length $i-1$. It signs this message and broadcasts the resulting message m of length i with notarized core N. The message m is received at time $r+i+3$ by all correct processors. Since there are $n-t \geqq 2t+1$ correct processors, at most t of which have signed m, and since by Lemma 4.1.2 every correct processor supports N, m is acceptable to some correct processor that has not yet signed it, so the induction holds. □

LEMMA 4.1.4. *If a correct processor p initiates a core protocol at time r, then the core protocol runs to completion and the correct processors fire at time $r+t+5$.*

Proof. By Lemma 4.1.3, at time $r+3+t$ at least one correct processor receives a new acceptable message m. Thus by time $r+4+t$ every correct processor receives an acceptable message of length $t+1$, so all correct processors broadcast "fire$_p$". Since there are at least $n-t > t+1$ correct processors, every processor receives at least $t+1$ commands to fire at time $r+5+t$, so a firing will indeed take place at time $r+5+t$. □

We now show that for an arbitrary initiator p, the core protocol never causes two correct processors to fire at different times. Let p be a possibly faulty processor initiating a core protocol. If S is a set of processors, we say that *S forms a core for p* if any processor in S forms a core for p. A *group* is a maximal set of correct processors receiving $E_p(p)$ at the same time. Let G be a group and let s be the time at which the members of G receive $E_p(p)$. Let H be the set of correct processors not in G.

LEMMA 4.1.5. *If G forms a core for p, then H does not form a core for p.*

Proof. First we observe that if G forms a core, then the core contains no signatures of processors in H. Similarly, no signature of a processor in G is contained in a core formed by any processor in H.

If G forms a core for p, then there exists some g in G that received at least $n-t$ messages of the form $E_q(E_p(p))$ at time $s+1$. Since none of those messages were sent by processors in H, we have $|H| \leqq t$. Thus even if the processors in H form a group and t faulty processors cooperate in helping H to form a core, the total number of cooperating processors is $2t < n-t$, so H cannot form a core. □

LEMMA 4.1.6. *If G forms a core for p and if any processor forms a notarized core N for p, then every processor in G supports N.*

Proof. Every notarized core N contains at least $n-t$ cores, at least $n-2t$ of which were formed by correct processors. Since no processor in H forms a core at least $n-2t$ of the cores in N were formed by processors in G and therefore contain all the signatures of all the processors in G. □

LEMMA 4.1.7. *Let N be a notarized core for p. If some g in G supports N then*
(1) *all processors in G support N, and*
(2) *no processor in H supports N.*

Proof. We will show that if g belongs to $n-2t$ of the cores in N, then G forms a core for p. It follows by Lemma 4.1.6 that every processor in G supports N. This will give us (1). Furthermore, by Lemma 4.1.5 if G forms a core, then H does not. Since neither processors in G nor in H form cores containing signatures of processors in H, the only cores that contain processors in H are formed by faulty processors. Thus, there can be at most $t < n-2t$ of them, so we have (2).

It remains to show that if g belongs to at least $n-2t > t$ of the cores in N, then G forms a core. This is immediate from the fact that no processor in H forms a core containing elements of G. Thus, if g appears in more than t cores, at least one of these was formed by some processor in G. □

LEMMA 4.1.8. *If any processor in G ever finds a message acceptable, then G contains at least $n-2t$ processors.*

Proof. Let m be acceptable to some g in G and let N be the notarized core of m. Of the $n-2t \geqq t+1$ cores in N containing g, at least one is signed by a correct processor. Let q be such a correct processor and let C be the core in N signed by q; i.e., $E_q(C)$ has the form

$$E_q(C) = E_q(\langle \cdots, E_g(E_p(p)), \cdots \rangle).$$

Of the $n-t$ processors whose signatures belong to C, at least $n-2t$ are correct. These $n-2t$ correct processors (one of which is g) all wrote to q at the same time, indicating that they received $E_p(p)$ at that time. Since no processor in H received $E_p(p)$ at the same time as g, no processor in H belongs to C. Since the correct processors are in either G or H, it follows that G contains at least $n-2t$ processors. □

LEMMA 4.1.9. *Let m be a message that is new and acceptable to processor g in group G at time z. Then $E_g(m)$ is acceptable to all processors in G at time $z+1$.*

Proof. Let N be the notarized core of m. One of the conditions of acceptability is that g supports N. By Lemma 4.1.7, every processor in G supports N. By condition (3) of acceptability, g first received $E_p(p)$ at time $z-|m|-2$, as did all other processors in G (by definition of a group), so every processor in G first received $E_p(p)$ at time $(z+1)-|E_g(m)|-2$. Thus every processor in G finds $E_g(m)$ acceptable at time $z+1$. □

LEMMA 4.1.10. *Let f be the earliest time at which some correct processor q fires (as a result of the core protocol initiated by p). Then all correct processors fire at time f.*

Proof. Since q fires only if it simultaneously receives at least $t+1$ messages "fire$_p$", some correct processor g sent "fire$_p$" at time $f-1$. Therefore, g received an acceptable message m of length $t+1$ at time $f-1$. Let G be the group of g. Without loss of generality, let

$$m = E_{t+1}(E_t(\cdots E_1(N)\cdots)).$$

Let $c = p_j$ be a correct processor among the $t+1$ processors that signed N. Let

$$m' = E_{j-1}(\cdots E_1(N)\cdots).$$

Since c finds m' acceptable, c supports N. Since g finds m acceptable, g supports N. It follows from Lemma 4.1.7(2) that c belongs to G. Let z be the time when c receives m'. By Lemma 4.1.9, all processors in G find $E_c(m')$ acceptable at time $z+1$. Furthermore, by Lemma 4.1.8, G contains at least $n-2t \geqq t+1$ processors, so there will be

some processor in G that has not yet signed N, provided $|E_c(m')| \leqq t$. By repeated application of Lemmas 4.1.9 and 4.1.8, all processors in G receive an acceptable message of length $t+1$ at time $f-1$, so they all broadcast "fire$_p$" at time $f-1$. Recall that G contains at least $t+1$ processors. It follows from the definition of the core protocol that all correct processors fire at time f. □

The proof of Theorem 4.1 follows directly from Lemmas 4.1.4 and 4.1.10. It is clear from the definition of the protocol that the number of bits of communication is polynomial in n. □

4.2. Lower bound. We next give a matching lower bound, $n \geqq 3t+1$, for the timing fault model. As noted in § 2, the lower bound of Theorem 4.2 holds also for the fault model of Theorem 4.1 (even without collusion).

THEOREM 4.2. *In the timing fault model there is a t-resilient distributed firing squad algorithm only if* $n \geqq 3t+1$.

Proof. Consider first the proof that there is no algorithm for $t=1$ and $n=3$. We consider four scenarios with three processors, A, B, and C, in each. Processor C is faulty in Scenarios 1 and 4, B is faulty in Scenario 2, and A is faulty in Scenario 3. It is possible to fix the wake-up times and the message transmission times (see Fig. 2) so that the following lemmas hold.

LEMMA 4.2.1. *If A fires at time z in Scenario* 1, *then A fires at time $z+1$ in Scenario* 4.

Proof. This follows since Scenarios 1 and 4 are identical except that all processors wake up exactly one time unit later in Scenario 4. □

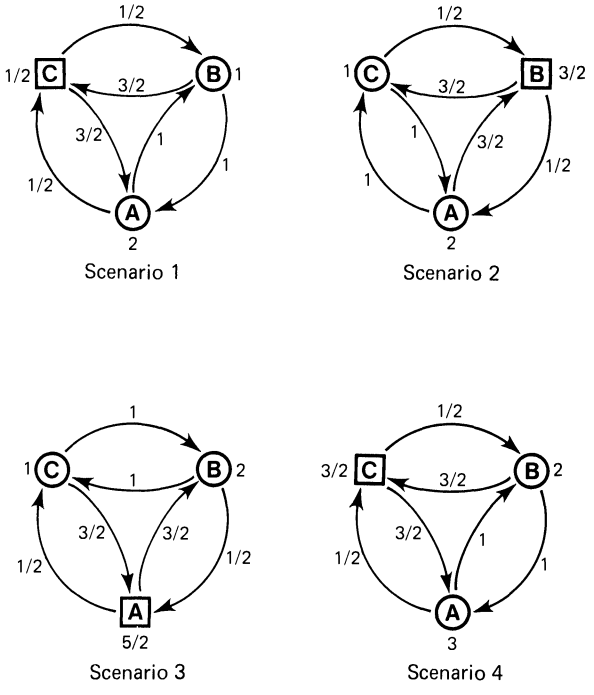


FIG. 2. *The scenarios used to prove Theorem* 4.2. *The number on the edge directed from processor X to processor Y is the message transmission time from X to Y. The number written next to processor X is the time when processor X wakes up. Correct (faulty) processors are drawn as circles (squares).*

For the next lemma it is convenient to introduce the "local step number" of a processor. A processor executes its first local step at the time it wakes up, and the local step number is incremented by one at each subsequent step. For example, in Scenario 1 in Fig. 2, A is executing its first step at global time 2, whereas B is executing its second step at global time 2. Letting p denote a processor, two scenarios are *p-equivalent* if the message history of p, i.e., messages received and messages sent at each local step of p, are the same in the two scenarios. Two scenarios are *strongly p-equivalent* if they are p-equivalent and p wakes up at the same global time in both scenarios.

LEMMA 4.2.2. *For $i=1,2,3$, Scenarios i and $i+1$ are strongly p-equivalent where p is the processor that is correct in both scenarios.*

Proof. By inspection of the scenarios in Fig. 2, one can easily verify that the following two facts hold for all four scenarios and for all integers $s \geqq 1$:

(1) for all messages sent from A to B, from B to C, or from A to C, the message sent at local step s of the sender is received at local step $s+2$ of the receiver; and

(2) for all messages sent from B to A, from C to B, or from C to A, the message sent at local step s of the sender is received at local step s of the receiver.

It follows easily from these facts (formally by induction on the local step number) that any two scenarios are p-equivalent where p is any of the three processors. The lemma then follows immediately from the choice of the wake-up times. □

These lemmas easily give a contradiction. Say that A fires at time z in Scenario 1. By strong A-equivalence of Scenarios 1 and 2, A fires at time z in Scenario 2. Since A and C are correct in Scenario 2, C also fires at z in Scenario 2. By a similar argument, B fires at z in Scenario 3, and A fires at z in Scenario 4, which contradicts Lemma 4.2.1.

The impossibility proof for general n and t with $n \leqq 3t$ is done as usual (cf. [25]) by replacing each processor by a group of at least one and at most t processors. The intragroup transmission times are all 1. The intergroup transmission times and the wake-up times are chosen as in Fig. 2. This completes the proof of Theorem 4.2. □

5. Collusion.

5.1. Upper bound. In this section we examine the distributed firing squad problem in the authenticated Byzantine model, in which faulty processors may share signature functions but they cannot rush messages.

THEOREM 5.1. *In the model with Byzantine failures and authentication where faulty processors can collude but cannot rush, there exists a t-resilient distributed firing squad algorithm requiring $n \geqq 2t+1$ processors, $2t+1$ rounds, and an amount of communication polynomial in n.*

We describe the main ideas informally before giving the formal proof. As in the other protocols, correct processors attempt to build messages signed by several processors and to use the length of these messages to synchronize. Since faulty processors can add several signatures at a given step, we wish to obtain a sort of "notarization" for each signature in a string of signatures guaranteeing that a specific amount of time was spent adding the signature.

In the straightforward approach, a processor p requests notarization of a signed message $E_p(m)$ by broadcasting $E_p(m)$. Then all processors attempt to obtain at least $t+1$ acknowledgments of the form $E_q(E_p(m))$. The list

$$m' = \langle E_{q_1}(E_p(m)), \cdots, E_{q_{t+1}}(E_p(m)) \rangle$$

is the notarization of $E_p(m)$. If the length of a message is the number of notarizations

it has undergone, then a message of length k requires exactly $2k$ steps to be constructed, even if the k signers of the message are faulty. Although conceptually simple, this approach leads to an algorithm with communication complexity exponential in t, since each notarization increases the size of a message by at least the factor $t+1$. Our algorithm uses the idea of notarization with an implementation that is harder to prove correct but that requires communication only polynomial in n. The basic idea is as follows. Suppose that p has received a notarized message m in the form of at least $t+1$ signatures of m by distinct processors. Then p "requests support" of $E_p(m)$ by broadcasting $E_p(m)$ together with a "proof" that m was notarized. This proof consists of $t+1$ signatures of m by distinct processors. Any correct processor q receiving $E_p(m)$, together with such a proof, "supports" $E_p(m)$ by signing $E_p(m)$ and broadcasting the result. The key fact is that the proof that m was notarized can be thrown away by q at this point, so message length does not grow exponentially. The idea of notarization and its implementation below is similar to the fault-tolerant distributed clocks described in [1], [12]. Similar ideas were also used in [27].

Proof of Theorem 5.1. We first define certain types of messages. A *proper* message has the form

$$E_{i_1}(E_{i_2}(\cdots E_{i_k}(p_{i_k})\cdots))$$

where the k signatures are by distinct processors. Such a message is called an α_k*-message.*

At various times in the protocol, processors may request support for a message m. A processor p *supports* m by sending a *support message* of the form $E_p(\text{support } m)$. We let $S(m)$ denote a support message for m.

A *proof* of a message m is a list of $t+1$ support messages for m, each signed by a distinct processor. We let $P(m)$ denote a proof for message m.

A processor p *requests support* for a message $E_p(m)$ by broadcasting a message of the form $\langle E_p(m), P(m)\rangle$. We let $R(E_p(m))$ denote a request for support of $E_p(m)$. When $E_p(m)$ has the form α_k, $k>1$, we call this request an R_k*-message.* An R_1-message has the form $\langle E_p(p), \lambda\rangle$, where λ denotes the empty string.

We now describe the protocol for a correct processor p. At every step p may issue both support messages and requests for support. In particular, after receiving at each step, p does the following.

(1) p chooses the maximum i such that p can form an R_i-message, $R(E_p(m)) = \langle E_p(m), P(m)\rangle$, where $E_p(m)$ is proper and p could not form an R_i-message at any previous step. If such an i exists then p broadcasts an R_i-message. If it can construct several syntactically distinct R_i-messages, then it arbitrarily chooses one to broadcast.

(2) For each processor q, p chooses the maximum j such that p receives an R_j-message $\langle E_q(m), P(m)\rangle$ from q. If $j<t+1$, then p broadcasts the corresponding support message $E_p(\text{support } E_q(m))$.

(3) If p receives an R_j-message for some $j\geqq t+1$, then p fires.

Viewing the number of signatures on a message as a clock, the two key lemmas state that the faulty processors cannot increment the clock faster than by 1 within two steps (Lemma 5.1.1) and that the correct processors can increment the clock at least that quickly (Lemma 5.1.2).

LEMMA 5.1.1. *Let $R(m)$ be an R_i-message and let s be the earliest time at which some correct processor sends $R(m)$. Let s' be the earliest time at which some (possibly faulty) processor q sends $R(m')$, where m' is an α_j-message of the form $E_b(E_c(\cdots(m))$. Then $s'\geqq s+2(j-i)$.*

Proof. The proof is by induction on $j-i$.

Basis $j-i=0$. Since $R(m)$ is sent by a correct processor, m is of the form $E_p(m'')$ for some correct processor p. Since $m'=m$ in this case, and since q cannot forge p's signature, q cannot construct $R(m')$ before time s.

Assume the lemma inductively for $j-i \leqq k$. Let $j=i+k+1$. Let $m''=E_c(\cdots(m))$ be such that $E_b(m'')=m'$. By the inductive hypothesis $R(m'')$ can be constructed no sooner than step $s+2(j-i-1)$.

If q constructs $R(m')$ at s', then q constructs $P(m'')$ at s', so q receives $t+1$ $S(m'')$ messages no later than time s'. Thus some correct processor received $R(m'')$ at $s'-1$, so $R(m'')$ was sent not later than $s'-2$. We therefore have $s'-2 \geqq s+2(j-i-1)$ by the inductive hypothesis, so $s' \geqq s+2(j-i)$ and the induction holds. □

LEMMA 5.1.2. *Fix a time s. Let i be the maximum i such that a correct processor broadcasts an R_i-message at time s, and let p be such a processor. If $i<t+1$, then by time $s+2$ some correct processor q forms and broadcasts an R_j-message for some $j \geqq i+1$.*

Proof. Without loss of generality, we may assume s is the first time at which p can construct an R_i-message. In this case p broadcasts $R(E_p(m))=\langle E_p(m), P(m)\rangle$ at s ($\langle E_p(m), \lambda\rangle$ if $i=1$), where $E_p(m)$ is an α_i-message, and every processor receives $R(E_p(m))$ at time $s+1$. Since this is the only request for support sent by p at s, every correct processor responds by broadcasting $S(E_p(m))$ at $s+1$. Thus all processors receive $n-t \geqq t+1$ $S(E_p(m))$ messages at $s+2$, each signed by a distinct processor, so at time $s+2$ every correct processor can construct a proof for $E_p(m)$. By signing $E_p(m)$ to obtain an α_{i+1}-message and combining this with the proof for $E_p(m)$, any processor that has not already signed $E_p(m)$ can construct an R_{i+1}-message. Because $i<t+1$, there is at least one such correct processor, say, q. If q has already broadcast an R_j-message for some $j \geqq i+1$, then the lemma holds trivially. Otherwise, by Step 1 of the protocol and the fact that it can construct an R_{i+1}-message, q will broadcast an R_j-message for some $j \geqq i+1$ at time $s+2$. □

LEMMA 5.1.3. *If any correct processor awakens, then every correct processor eventually fires. Moreover, if s is the earliest time when some correct processor awakens, then every correct processor fires by time $s+2t+1$.*

Proof. Upon awakening, each correct processor forms and broadcasts an R_j-message, for some $j \geqq 1$, since the proof of an R_1-message is the empty string. Let p be the first correct processor to awaken, and let s be the time at which it awakens. By t applications of Lemma 5.1.2, some correct processor constructs and broadcasts an R_j-message for some $j \geqq t+1$ by time $s+2t$. Thus every correct processor fires by $s+2t+1$. □

LEMMA 5.1.4. *In any execution of the protocol resulting in a firing, all correct processors fire simultaneously.*

Proof. Let p be the first correct processor to fire and let f be the time at which p fires. If p fires at f then, for some $k \geqq t+1$, p receives an R_k-message at f. Let $R(m')$ be such a message. Without loss of generality let $m'=E_k(\cdots E_2(E_1(p_1)))$. Let i be the maximum i, $1 \leqq i \leqq k$, such that p_i is correct. (Since there are at most t faulty processors, some such p_i exists.) If $p_i=p_k$, then all processors receive $R(m')$ at f so all fire simultaneously at f.

If $i<k$, then at some round $f'<f$, p_i broadcast an R_i-message. This message was received by all correct processors no later than round $f'+1 \leqq f$. If $i \geqq t+1$, then by Step 3 of the protocol all correct processors fire at $f'+1$. Since f is the first round at which any correct processor fires, we have $f'+1=f$, and all correct processors fire simultaneously.

We now consider the case $i<t+1$. Let s be the time at which p_i broadcasts $R(m)$, where $m=E_i(\cdots(E_1(p_1)))$. Since p_i is correct, s is the earliest time at which $R(m)$ is

sent. By Lemma 5.1.1, no processor can construct $R(m')$ before step $s+2(t+1-i)$, so $f \geqq s+2(t+1-i)+1$. By $t+1-i$ applications of Lemma 5.1.2, some correct processor constructs and broadcasts an R_j-message for some $j \geqq t+1$ by time $s+2(t+1-i)$, so every correct processor receives such an R_j-message by time $s+2(t+1-i)+1$. Thus all correct processors fire by time $s+2(t+1-i)+1$. Since p is the first correct processor to fire and p fires at f, we have $f \leqq s+2(t+1-i)+1$. Thus all correct processors fire simultaneously at f. □

It is now easy to complete the proof of Theorem 5.1. By Lemmas 5.1.3 and 5.1.4, the algorithm is t-resilient. By Lemma 5.1.3, every correct processor fires within $2t+1$ rounds after the first correct processor awakens. Finally, at each step a correct processor broadcasts at most one request for support and n support messages, so at each step the number of bits sent by any correct processor is polynomial in n. Since there are n processors and $O(n)$ steps, the total amount of communication is polynomial in n. □

5.2. Lower bound.

THEOREM 5.2. *In the fault model of Theorem* 5.1 (*Byzantine faults with authentication, collusion, but no rushing*), *if* $t \geqq 3$, *there is a t-resilient distributed firing squad algorithm only if* $n \geqq \lfloor 5t/3 \rfloor + 1$.

Proof. The general outline of the proof is similar to the proof of Theorem 4.2. Consider the impossibility proof for $t=3$ and $n=5$. We consider six scenarios, with three faulty and two correct processors in each. Figure 3 shows the message transmission

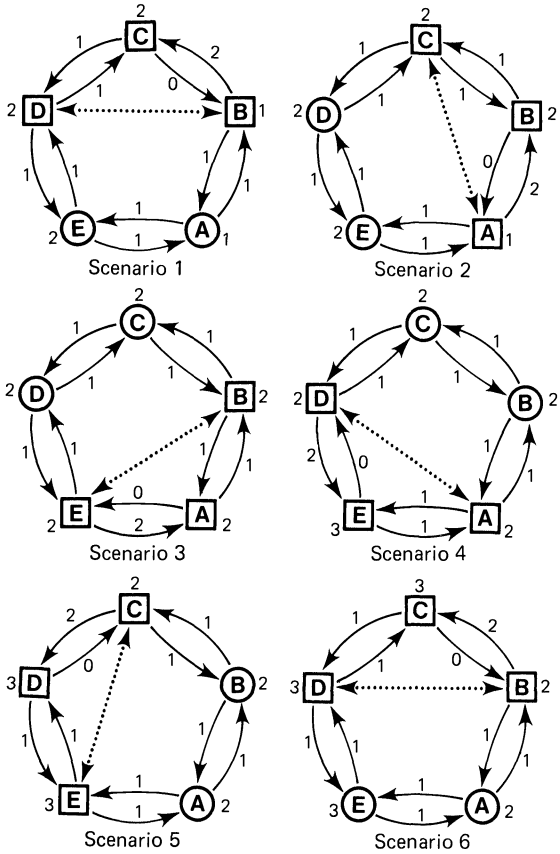


FIG. 3. *The scenarios used to prove Theorem* 5.2.

times, wake-up times, and which processors are faulty in each scenario. A link that is not drawn in these scenarios means that the faulty processor at one end of the link does not communicate along that link; i.e., no messages are sent along that link by the faulty processor, and messages received along that link are ignored. The link drawn as a dotted line is used only by faulty processors. Therefore, in each scenario the network is essentially a ring from the point of view of a correct processor. In each scenario, two of the faulty processors simulate a "timing fault" where messages in one direction take time 2 and messages in the other direction take time 0. The only nonobvious part is simulating a transmission time of zero. To see how this is done focus, for example, on Scenario 1, where messages from C to B take zero time. Note that D is also faulty in Scenario 1. Whenever D takes a step at some time x in which it should send the message m to C, it sends m to B also. At time $x+1$, B has enough information to do the processing that C would do at time $x+1$ to find the message m' that C should send to B at time $x+1$ (the ability of B to sign messages with C's signature is necessary here). But B has m' during the step it is executing at time $x+1$, thus simulating the transmission of m' from C to B in zero time. Message transmission time of 0 is simulated similarly in the other scenarios.

By following the proof of Theorem 4.2, it is straightforward to show that analogues to Lemmas 4.2.1 and 4.2.2 hold. (Formally, in defining equivalence of scenarios, only messages sent along the ring links are included in message histories; the messages sent over dotted links are not included.)

The proof for general n and t is done by replacing each processor by a group of at most $\lfloor t/3 \rfloor$ or at most $\lfloor t/3 \rfloor + 1$ processors in such a way that the total number of faulty processors never exceeds t in any of the scenarios. □

Remark. Regarding the condition $t \geqq 3$ in Theorem 5.2, by using the assumption that the receiver of a message knows the identity of the sender, it is not hard to find a 2-resilient distributed firing squad algorithm for any number $n \geqq 2$ of processors. Briefly, the algorithm is a modification of the algorithm used to prove Theorem 3.1 in the case of no collusion and no rushing. In the modified algorithm, whenever a processor p sends a message m to a processor q, p attaches a header to m that says "the next signer of this message should be processor q". When checking acceptability of a proper message

$$m = E_{i_1}(E_{i_2}(\cdots E_{i_k}(\text{Awake})\cdots))$$

p also checks that m was received from processor p_{i_1} and that each signature in m matches the header of the message being signed. This effectively prevents two faulty processors from adding two signatures in one step, even if they collude. Given this observation, the correctness proof is identical to that of Theorem 3.1, and details are left to the reader. (Note, however, that the modification does not prevent three faulty processors from adding three signatures in two steps, so this method does not generalize to $t \geqq 3$.)

Remark. We can close the fault-tolerance gap between Theorems 5.1 and 5.2 by adding the requirement that each correct processor must broadcast one message at each step (formally, if $(m_1, m_2, \cdots, m_n)$ is in the range of some sending function β_i, then $m_1 = m_2 = \cdots = m_n$). Note that the algorithm of Theorem 5.1 meets this requirement. With this requirement, the proof of the lower bound, Theorem 5.2, can be done with a ring of four processors, two of which are faulty, thus improving the lower bound to $n \geqq 2t+1$. This can be done because the broadcast condition prevents the correct processors from hiding from the faulty processors signed text that these faulty processors would otherwise have to forge. Details are left to the reader. (Note that this

result applies to communication systems like the Ethernet, in which eavesdropping cannot be avoided.)

6. Related results.

6.1. Byzantine agreement with nonunison start. Suppose we want to solve authenticated Byzantine agreement when the correct processors do not all awaken at the same time and the faulty processors can rush. For simplicity, we consider the version of the Byzantine agreement problem as in [11], where the protocol is initiated by a single "sender" processor p that wants to send a "value" v to all the processors. If the sender p is correct, then p sends $E_p(v)$ to all processors at exactly the same step; in this case, we require that all correct processors eventually decide that v was the value sent by p. If the sender p is faulty, it can initially send different values to different processors, and it can send values at different times; in this case, we require that if any correct processor decides on a value u, then all correct processors must decide on u. The second type of faulty behavior, initiating the protocol at different times with respect to different processors, is not considered in Dolev and Strong [11], and their efficient ($t+1$ round) algorithm does not work in this case. An obvious solution would be to first run the firing squad algorithm of Theorem 4.1 to synchronize the processors and then run the Dolev-Strong algorithm for a total time of $2t+6$. This time can be improved to $t+5$ by modifying the algorithm of Theorem 4.1 to solve agreement directly.

THEOREM 6.1. *In the model with Byzantine failures and authentication, in which faulty processors can both rush and collude, there is a t-resilient protocol for Byzantine agreement with nonunison start, requiring $t+5$ rounds, $n \geqq 3t+1$ processors, and communication polynomial in n.*

Proof. The algorithm of Theorem 4.1 is modified by associating a value with every core and with every notarized core. Let h be a function that maps a set of values to a single value as follows: $h(\{v\}) = v$; if S is not a singleton set, then $h(S) = 0$. The value associated with the core

$$\langle E_{i_1}(E_p(v_1)), \cdots, E_{i_k}(E_p(v_k))\rangle$$

is $h(\{v_1, \cdots, v_k\})$. The value of the notarized core

$$\langle E_{i_1}(C_1), \cdots, E_{i_k}(C_k)\rangle$$

is h applied to the set containing the values of the cores $C_1, \cdots, C_k$. Each processor q remembers the set V_q of values of notarized cores that it has seen in acceptable messages. In addition to the previous algorithm for signing and forwarding acceptable messages, whenever q receives an acceptable message m containing the notarized core N, and if the value of N is not currently in V_q, then that value is added to V_q and q signs m and broadcasts the result $E_q(m)$. At the point where q receives an acceptable message of length $t+1$, q signs and broadcasts "decide$_p$ $h(V_q)$". A processor decides v if it receives, at the same step, at least $t+1$ messages "decide$_p$ v" signed by different processors. The correctness proof is very similar to the correctness proof given in § 4; only Lemma 4.1.10 requires modification. Details are left to the reader. □

A similar modification to the algorithm solves the version of the agreement problem where each processor begins the algorithm with an initial value.

A drawback in modifying a distributed firing squad algorithm to solve agreement is that it requires $n > 3t$. By adapting an algorithm of Cristian, Aghili, Strong, and Dolev [7] to the model used in this paper, there is a completely different solution, not using firing squad ideas, which tolerates any number $t \leqq n$ of faults but which takes

$2t+2$ rounds. It is an open question whether arbitrary fault tolerance and time $t+O(1)$ can be obtained simultaneously.

6.2. Distributed firing squad with a global clock. At this point, one might suspect that the difficulty of the distributed firing squad problem is due to the processors having no common notion of global time. We show now, however, that the problem does not become trivial, even with a common clock, although for one fault model the problem does become easier. Informally, a common clock means that at each integer time s, all correct processors taking their unison step at time s know that it is time s. (More formally, the state set Q_i of p_i is partitioned into sets $Q_{i,j}$ for integer $j \geqq 0$, and all transitions from a state in $Q_{i,j}$ must go to states in $Q_{i,j+1}$. Each set $Q_{i,j}$ contains a copy $q_{0,j}$ of the quiescent state. The initial state of p_i is $q_{0,0}$. If p_i is in state $q_{0,j}$ and receives only null messages, then p_i next enters state $q_{0,j+1}$.)

Let the *clocked distributed firing squad* problem be defined like the distributed firing squad problem, but in the model with a common clock. We first give a reduction similar to that of Theorem 3.2. As corollaries of this reduction, clocked distributed firing squad still requires $t+1$ rounds for fail-stop faults, and $n \geqq 3t+1$ is needed in the unauthenticated Byzantine case.

THEOREM 6.2. *Let A be an algorithm for clocked distributed firing squad that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that has time complexity of k rounds. Then there exists an algorithm for* WBA *that is t-resilient to fail-stop faults (respectively, unauthenticated Byzantine faults) and that always halts in k rounds.*

Proof. Given A, define the function f on the natural numbers as follows. For a given integer $r \geqq 0$, consider the run of A in which all processors wake up at (common) time r and there are no faults; then $f(r)$ is the (common) time when all processors fire. Since $f(r) \geqq r$ for all r, there must be a time s such that $f(s+1) > f(s)$.

Now consider an instance of WBA in which processor p_i has initial value v_i. If $v_i = 0$, then p_i begins simulating A as though it were time s. That is, at time 0 of the WBA algorithm, p_i acts as though it were in state $q_{0,s}$ receiving the awake message from w and null messages from the rest. If $v_i = 1$, then p_i waits one step during the WBA algorithm and begins simulating A as though it were awakened at time $s+1$. (In general, time i in the WBA algorithm corresponds to time $s+i$ in the simulation of A.) Let $m = f(s) - s$. If the simulation of A causes p_i to fire within m steps after the beginning of the WBA algorithm, then p_i decides 0 at time m; otherwise, p_i decides 1 at time m. Since A has time complexity k, we have $m \leqq k$. The correctness proof for this WBA algorithm is very similar to the proof of Theorem 3.2 and is left to the reader. □

The next result concerns the case of Byzantine faults with authentication and rushing and shows that the clocked version of the problem is easier for this fault model; specifically, the fault-tolerance improves to any $t \leqq n$, and the time is optimal.

THEOREM 6.3. *In the model with Byzantine failures and authentication, in which faulty processors can both rush and collude, there is a t-resilient clocked distributed firing squad protocol requiring $t+1$ rounds, $n \geqq t$ processors, and communication polynomial in n.*

Proof. In this algorithm, a *proper* message has the form

$$m = E_{i_1}(E_{i_2}(\cdots E_{i_k}(\text{“fire at } c\text{”})\cdots))$$

where c is a natural number modulo $t+1$, where $k \geqq 1$, and where the k signatures are by distinct processors; the *length* of m is k and the *content* of m is c. Such a

message is *acceptable* at common time r if and only if $r \equiv c + k \bmod (t+1)$. This message is *new* to p if p's signature does not appear in m.

A processor p that is awakened by the Awake message at common time s computes $c = s \bmod (t+1)$ and broadcasts E_p("fire at c"). If any processor q receives one or more new acceptable messages with content c at some time, q arbitrarily chooses one, say m, and broadcasts $E_q(m)$. A processor q fires at common time z if q has received, at time z or earlier, an acceptable message m with content c, where $c \equiv z \bmod (t+1)$ such that message m has not caused q to fire at any time earlier than z.

It is clear that if some correct processor awakens at common time s, then all correct processors fire on or before common time $s+t+1$. To argue that all correct processors fire together, we note that if m is new and acceptable to some correct p at time r, then $E_p(m)$ is acceptable to all correct processors at time $r+1$. Let z be the earliest time when some correct processor fires, and let p be a correct processor that fires at z. Therefore, p received an acceptable message m with content c, where $c \equiv z \bmod (t+1)$. If m was received before time z, then all correct processors receive an acceptable message with content c on or before time z, because p must have broadcast such a message before time z. If m is received by p at time z, then $z \equiv c + k \bmod (t+1)$ where k is the length of m, because m is acceptable at time z. Since $c \equiv z \bmod (t+1)$ and $k \geqq 1$, it follows that $k \geqq t+1$. So m must contain the signatures of $t+1$ processors, at least one of which is correct, and again it is easy to argue that all correct processors received an acceptable message with content c by common time z. □

Acknowledgment. We are grateful to Nancy Lynch for saving an extra round in our reduction of the WBA problem mentioned in § 3.2.

REFERENCES

[1] C. ATTIYA, D. DOLEV, AND J. GIL, *Asynchronous Byzantine consensus*, Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 119–133.

[2] A. BAR-NOY AND D. DOLEV, *Families of consensus algorithms*, Proc. Aegean Workshop on Computing, Greece, 1988, pp. 380–390.

[3] J. E. BURNS AND N. A. LYNCH, personal communication, 1984.

[4] ———, *The Byzantine firing squad problem*, in Advances in Computing Research: Parallel and Distributed Computing, Vol. 4, JAI Press Inc., Greenwich, CT, 1987, pp. 147–161.

[5] B. A. COAN, *A communication-efficient canonical form for fault-tolerant distributed protocols*, Proc. 5th ACM Symposium on Principles of Distributed Computing, 1986, pp. 63–72.

[6] B. A. COAN AND C. DWORK, *Simultaneity is harder than agreement*, Proc. 5th IEEE Symposium on Reliability in Distributed Software and Database Systems, 1986, pp. 141–150.

[7] F. CRISTIAN, H. AGHILI, R. STRONG, AND D. DOLEV, *Atomic broadcast: From simple message diffusion to Byzantine agreement*, Proc. 15th International Conference on Fault Tolerant Computing, 1985, pp. 1–7.

[8] D. DOLEV, C. DWORK, AND L. STOCKMEYER, *On the minimal synchronism needed for distributed consensus*, J. Assoc. Comput. Mach., 34 (1987), pp. 77–97.

[9] D. DOLEV, M. J. FISCHER, R. FOWLER, N. A. LYNCH, AND H. R. STRONG, *Efficient Byzantine agreement without authentication*, Inform. and Control, 52 (1982), pp. 257–274.

[10] D. DOLEV, R. REISCHUK, AND H. R. STRONG, *Eventual is earlier than immediate*, Proc. 23rd IEEE Symposium on Foundations of Computer Science, 1982, pp. 196–203.

[11] D. DOLEV AND H. R. STRONG, *Authenticated algorithms for Byzantine agreement*, SIAM J. Comput., 12 (1983), pp. 656–666.

[12] C. DWORK, N. LYNCH, AND L. STOCKMEYER, *Consensus in the presence of partial synchrony*, J. Assoc. Comput. Mach., 35 (1988), pp. 288–323.

[13] C. DWORK AND Y. MOSES, *Knowledge and common knowledge in Byzantine environments* I: *Crash failures*, Proc. Conference on Theoretical Aspects of Reasoning About Knowledge, Morgan-Kaufmann, Los Altos, CA, 1986, pp. 149–170.

[14] M. J. FISCHER, N. A. LYNCH, AND M. MERRITT, *Easy impossibility proofs for distributed consensus problems*, Distributed Computing, 1 (1986), pp. 26-39.

[15] M. J. FISCHER, N. A. LYNCH, AND M. PATERSON, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374-382.

[16] J. HALPERN, B. SIMONS, H. R. STRONG, AND D. DOLEV, *Fault-tolerant clock synchronization*, Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 89-102.

[17] L. LAMPORT, *The weak Byzantine generals problem*, J. Assoc. Comput. Mach., 30 (1983), pp. 668-676.

[18] L. LAMPORT AND M. J. FISCHER, *Byzantine generals and transaction commit protocols*, Tech. Report Op. 62, SRI International, Menlo Park, CA, 1982.

[19] L. LAMPORT AND P. M. MELLIAR-SMITH, *Synchronizing clocks in the presence of faults*, J. Assoc. Comput. Mach., 32 (1985), pp. 52-78.

[20] L. LAMPORT, R. SHOSTAK, AND M. PEASE, *The Byzantine generals problem*, ACM Trans. Programming Languages and Systems, 4 (1982), pp. 382-401.

[21] J. LUNDELIUS WELCH AND N. LYNCH, *A new fault-tolerant algorithm for clock synchronization*, Inform. and Comput., 77 (1988), pp. 1-36.

[22] M. MERRITT, personal communication, 1984.

[23] E. F. MOORE, *The firing squad synchronization problem*, in Sequential Machines, Selected Papers, E. F. Moore, ed., Addison-Wesley, Reading, MA, 1964.

[24] Y. MOSES AND O. WAARTS, *Coordinated traversal: $(t+1)$-round Byzantine agreement in polynomial time*, Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 246-255.

[25] M. PEASE, R. SHOSTAK, AND L. LAMPORT, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228-234.

[26] T. K. SRIKANTH AND S. TOUEG, *Optimal clock synchronization*, J. Assoc. Comput. Mach., 34 (1987), pp. 626-645.

[27] ———, *Byzantine agreement made simple: Simulating authentication without signatures*, Distributed Computing, 2 (1987), pp. 80-94.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 1013–1036, October 1989

011

FASTER SCALING ALGORITHMS FOR NETWORK PROBLEMS*

HAROLD N. GABOW† AND ROBERT E. TARJAN‡

Abstract. This paper presents algorithms for the assignment problem, the transportation problem, and the minimum-cost flow problem of operations research. The algorithms find a minimum-cost solution, yet run in time close to the best-known bounds for the corresponding problems without costs. For example, the assignment problem (equivalently, minimum-cost matching in a bipartite graph) can be solved in $O(\sqrt{n}m \log(nN))$ time, where n, m, and N denote the number of vertices, number of edges, and largest magnitude of a cost; costs are assumed to be integral. The algorithms work by scaling. As in the work of Goldberg and Tarjan, in each scaled problem an approximate optimum solution is found, rather than an exact optimum.

Key words. graph theory, networks, assignment problem, matching, scaling

AMS(MOS) subject classifications. 68Q20, 68Q25, 68R10, 05C70

1. Introduction.

Many problems in operations research involve minimizing a cost function defined on a bipartite or directed graph. A simple but fundamental example is the assignment problem. This paper gives algorithms for such problems that run almost as fast as the best-known algorithms for the corresponding problems without costs. For the assignment problem, the corresponding problem without costs is maximum cardinality bipartite matching.

The results are achieved by scaling the costs. This requires the costs to be integral-valued. Further, for the algorithms to be efficient, costs should be polynomially bounded in the number of vertices, i.e., at most $n^{O(1)}$. These requirements are satisfied by a large number of problems in both theoretical and practical applications.

Table 1 summarizes the results of the paper. The parameters describing the input are specified in the caption and defined more precisely below. The first column gives the problem and the best-known strongly polynomial time bound. Such a bound comes from an algorithm with running time independent of the size of the numbers (assuming the uniform cost model of computation [AHU]). The second column gives the time bounds achieved in this paper by scaling. The table shows that significant speedups can be achieved through scaling. Further, it will be seen that the scaling algorithms are simple to program. Now we discuss the specific results.

The assignment problem is to find a minimum-cost perfect matching in a bipartite graph. The strongly polynomial algorithm is the Hungarian algorithm [K55], [K56] implemented with Fibonacci heaps [FT]. This algorithm can be improved significantly when all costs are zero. Then the problem amounts to finding a perfect matching in a bipartite graph. The best-known cardinality matching algorithm, due to Hopcroft and Karp, runs in time $O(\sqrt{n}m)$ [HK]. The new time bound for the assignment problem is

*Received by the editors August 18, 1987; accepted for publication (in revised form) November 4, 1988.

†Department of Computer Science, University of Colorado, Boulder, Colorado 80309. The research of this author was supported in part by National Science Foundation grant DCR-851191 and AT&T Bell Laboratories.

‡Computer Science Department, Princeton University, Princeton, New Jersey 08544 and AT&T Bell Laboratories, Murray Hill, New Jersey 07974. The research of this author was supported in part by National Science Foundation grant DCR-8605962 and Office of Naval Research contract N00014-87-K-0467.

just a factor of $\log(nN)$ more than this. The algorithm is similar to the Hopcroft–Karp cardinality algorithm and appears simple enough to be useful in practice.

TABLE 1
Bounds for network problems.

Strongly Polynomial Bound	New Scaling Bound
Assignment problem $O(n(m + n\log n))$ [FT]	$O(\sqrt{n}m\log(nN))$
Shortest paths (single-source, directed graph, possibly negative lengths) $O(nm)$ [Bel]	$O(\sqrt{n}m\log(nN))$
Minimum cost degree-constrained subgraph of a bipartite multigraph $O(U(m + n\log n))$ [FT], [G83]	$O(\min\{\sqrt{U}, n^{2/3}M^{1/3}\}\overline{m}\log(nN))$
Transportation problem (uncapacitated or capacitated) $O(\min\{U, n\log U\}(m + n\log n))$ [FT], [EK], [L]	$O((\min\{\sqrt{U}, n\}m + U\log U)\log(nN))$
Minimum cost flow $O(n^2(m + n\log n)\log n)$ [GalT]	$O(nm\log n\log(nN)\log M)$ convex cost functions allowed $O(n(m + n\log n)\log M)$ lower bounds only

Parameters: n = number of vertices; m = number of edges; $\overline{m}$ = number of edges counting multiplicities; U = total degree constraints; N = maximum cost magnitude; and M = maximum flow capacity or lower bound, or edge multiplicity.

The new algorithm improves the scaling algorithm of [G85], which runs in time $O(n^{3/4}m\log N)$. The improvement comes from a different scaling method. The algorithms of [G85] compute an optimum solution at each of $\log N$ scales. The new method computes an approximate optimum at each of $\log(nN)$ scales; using $\log n$ extra scales ensures that the last approximate optimum is exact. The appropriate definition of approximate optimum is due to Tardos [Tard] and independently to Bertsekas [Ber79], Ber86]. The new approach to scaling was recently discovered by Goldberg and Tarjan for the minimum-cost flow problem [Go], [GoT87a], [GoT87b]. Their minimum-cost flow algorithm solves the assignment problem in time $O(nm\log(nN))$, which this paper improves. Bertsekas [Ber87] gives an algorithm for the assignment problem that

also runs in sequential time $O(nm \log(nN))$ and has a distributed asynchronous implementation.

The assignment algorithm extends to other network problems. This paper presents extensions to problems on bipartite graphs and directed graphs. For algorithms on general graphs (and bidirected graphs) and other extensions, see § 4. Throughout this paper all undirected graphs are bipartite (we usually mention this explicitly).

Variants of minimum-cost perfect bipartite matching (such as minimum-cost bipartite matching) can be done in the same time bound. The linear programming dual variables for perfect bipartite matching can be found from the algorithm. This gives a solution to the shortest path problem when negative edge lengths are allowed. The table entry for the degree-constrained subgraph problem is just a factor of $\log(nN)$ more than the bound of [ET] for the corresponding problem without costs, namely, the problem of maximum flow in a 0-1 network. These bounds improve [G85] in a manner analogous to the assignment problem.

The table entry for the transportation problem is a good bound when total supply and demand (U) is small. The key fact for this bound is the low total augmenting path length for the assignment algorithm; this fact generalizes the bounds of [ET] for cardinality matching and 0-1 network flow. The entry for minimum-cost flow is a double scaling algorithm — it scales edge capacities, and at each scale solves a small transportation problem by the above cost-scaling algorithm. This algorithm is not as good asymptotically as the recent bound of Goldberg and Tarjan, $O(nm \log(n^2/m) \log(nN))$ [GoT87b]. The latter is just a factor of $\log(nN)$ more than the best bound for maximum value flow [GoT86]. The double scaling algorithm may be more useful in practice, however, since it requires fewer data structures. The double scaling algorithm generalizes to find a minimum-cost integral flow when the cost of each edge is an arbitrary convex function of its flow. The time bound is unchanged, as long as the cost for a given flow value can be computed in $O(1)$ time. The last bound of the table improves the previous one for problems where edges have a lower bound on the flow and infinite capacity (more generally, $O(n)$ finite capacities are allowed). Such problems arise as covering problems. We illustrate how this bound leads to an efficient strongly polynomial bound for the directed Chinese postman problem.

Section 2 presents the matching algorithm and its analysis, including facts used in the generalizations. Section 3 presents the extensions to more general network flow problems. Section 4 gives some concluding remarks. This section closes with definitions from graph theory; more thorough treatments are in [L], [PS], [Tarj].

We use interval notation for sets of integers: for integers i and j, define $[i..j] = \{k | k \text{ is an integer}, i \leq j \leq k\}$, $[i..j) = \{k | k \text{ is an integer}, i \leq j < k\}$, etc. The *symmetric difference* of sets S and T is denoted by $S \oplus T$. The function $\log n$ denotes logarithm to the base two.

For a graph G, $V(G)$ and $E(G)$ denote the vertex set and edge set, respectively. The given graph G is bipartite and has bipartition V_0, V_1 (so $V(G)$ is the disjoint union of V_0 and V_1, and any edge joins V_0 to V_1). The given graph G has m edges; in § 2, $n = |V_0| = |V_1|$ (we assume without loss of generality that the two sets of the bipartition have equal cardinality); in § 3, $n = |V(G)|$. If H is a subgraph of G, an *H-edge* is an edge in H and a *non-H-edge* is not in H. When an auxiliary graph G' is constructed from the given graph G, *G-edge* refers to an edge of G' that represents an edge of G. We use this term without explicit comment only when the representation is obvious (i.e., $vw \in E(G)$ is represented by $v'w'$, where v' and w' are obvious representatives of v and w). We say *path P ends with edge vw* if vw is at an end of P and further, v is an endpoint of P.

A *matching* in a graph is a set of vertex-disjoint edges. Thus a vertex v is in at most one matched edge vv'; v' is the *mate* of v. A *free* vertex has no mate. A *maximum cardinality matching* has the greatest number of edges possible; a *perfect matching* has no free vertices (and is clearly maximum cardinality). An *alternating path* (*cycle*) for a matching is a simple path (cycle) whose edges are alternately matched and unmatched. An *augmenting path* P is an alternating path joining two distinct free vertices. *Augmenting* the matching along P means enlarging the matching M to $M \oplus P$, thus giving a matching with one more edge. Suppose each edge e has a numeric *cost* $c(e)$; in this paper costs are integers in $[-N..N]$, unless stated otherwise. The cost $c(S)$ of a set of edges S is the sum of the individual edge costs. A *minimum* (*maximum*) *perfect matching* is a perfect matching of smallest (largest) possible cost. The *assignment problem* is to find a minimum perfect matching in a bipartite graph. More generally, a *minimum-cost maximum cardinality matching* is a matching that has the greatest number of edges possible, and subject to that restriction has minimum cost possible. (The phrase "minimum-cost maximum cardinality set" can be interpreted ambiguously. In this paper it refers to a set that has maximum cardinality subject to any other restrictions that have been mentioned, and among such sets has minimum cost possible.) A *minimum-cost matching* is a matching of minimum cost (its cardinality can be any value, including zero).

A *multigraph* has a set of edges $E(G)$, where each edge e has an integral *multiplicity* $u(e)$ (i.e., there are $u(e)$ parallel copies of e). The size parameter m is the number of edges, $m = |E(G)|$; $\overline{m}$ counts multiplicities, i.e., $\overline{m} = \sum\{u(e) | e \in E(G)\}$; M is the maximum edge multiplicity. (In a graph $M = 1$.) When each vertex v has associated nonnegative integers $\ell(v)$ and $u(v)$, a *degree-constrained subgraph* (DCS) is a subgraph such that each vertex has degree in $[\ell(v)..u(v)]$. It is convenient to use both set notation and functional notation for a DCS. Thus we use a capital letter D to denote a DCS, and the corresponding lowercase letter d to denote two functions defined by D: for an edge e, $d(e)$ denotes the multiplicity of e in D, and for a vertex v, $d(v)$ denotes the degree of v in D, i.e., $d(v) = \sum\{d(vw) | vw \in E(G)\}$. Hence $d(e) \leq u(e)$ and $\ell(v) \leq d(v) \leq u(v)$. The *deficiency* of DCS D at vertex v is $\phi(v, D) = u(v) - d(v)$. In a *perfect* DCS each deficiency is zero. The size of the DCS is measured by $U = \sum\{u(v) | v \in V\}$ (so U is twice the number of edges in a perfect DCS). When edges e have costs, the usual assumption is that each copy of e has the same cost, denoted $c(e)$. When this assumption fails, we use cost functions, defined in the text. Other definitions for DCS — e.g., *minimum perfect* DCS, *minimum-cost maximum cardinality* DCS, etc., follow by analogy with matching.

The *transportation problem* is to find a minimum-cost perfect DCS in a bipartite multigraph in which all edges have infinite multiplicity; alternatively, if M is the maximum degree constraint, all multiplicities are M. If some multiplicities are less than M, the problem is a *capacitated transportation problem.* The usual definition of the transportation problem allows nonnegative real-valued degree constraints and edge multiplicities (both given multiplicities and those in the solution). This paper deals with the integral case of this problem. Note that if the given degree constraints and multiplicities are rational, they can be scaled up to integers. Also note that no loss of generality results from the constraint in this paper that the solution to the transportation problem has integral multiplicities — such an optimum solution always exists when the given degree constraints and multiplicities are integral [L]. Finally note that in our terminology the minimum perfect DCS problem is the same as the capacitated transportation problem.

2. Matching and extensions. Section 2.1 presents our algorithm to find a minimum perfect matching in a bipartite graph. Section 2.2 gives extensions to other versions of matching, some facts about the algorithm needed in § 3, and our shortest path algorithm. In this section n denotes the number of vertices in each vertex set V_0, V_1 of the given bipartite graph.

2.1. The assignment algorithm. For convenience assume that the given graph G has a perfect matching (the algorithm can detect graphs not having a perfect matching, as indicated below).

The plan for the algorithm is to combine the Hungarian algorithm for weighted matching with the Hopcroft–Karp algorithm for cardinality matching. Recall that the Hungarian algorithm always chooses an augmenting path of smallest net cost. The Hopcroft–Karp algorithm always chooses an augmenting path of shortest length. Both of these rules can be approximated simultaneously *if* the costs are small integers. Arbitrary costs can be replaced by small integers by scaling. Thus our algorithm scales the costs. At each scale it computes a perfect matching. The computation is efficient because it is similar to the Hopcroft–Karp algorithm; the matching is close to optimum because the computation is similar to the Hungarian algorithm. Now we give the details.

Each scale of the algorithm finds a close-to-minimum matching, defined as follows. Every vertex v has a *dual variable* $y(v)$. A 1-*feasible matching* consists of a matching M and dual variables $y(v)$ such that for any edge vw,

$$y(v) + y(w) \le c(vw) + 1,$$
$$y(v) + y(w) = c(vw), \quad \text{for } vw \in M.$$

A 1-*optimal matching* is a perfect matching that is 1-feasible. If the $+1$ term is omitted from the first inequality, these are the usual complementary slackness conditions for a minimum perfect matching [L], [PS]. The following result is due to Bertsekas [Ber79], [Ber86].

LEMMA 2.1. *Let M be a 1-optimal matching.*

(a) *Any perfect matching P has $c(P) \ge c(M) - n$.*

(b) *If some integer k, $k > n$, divides each cost $c(e)$, then M is a minimum perfect matching.*

Proof. Part (a) follows because

$$c(M) = \sum \{c(e) | e \in M\} = \sum \{y(v) | v \in V(G)\} \le c(P) + n.$$

Part (b) follows from (a) and the fact that any matching has cost a multiple of k. □

This lemma is the basis for the *main routine* of the algorithm, which does the scaling. The routine starts by computing a new cost $\bar{c}(e)$ for each edge e, equal to $n+1$ times the given cost. Consider each $\bar{c}(e)$ to be a signed binary number $\pm b_1 b_2 \cdots b_k$ having $k = \lfloor \log(n+1)N \rfloor + 1$ bits. The routine maintains a variable $c(e)$ for each edge e, equal to its cost in the current scale. The routine initializes each $c(e)$ to 0 and each dual $y(v)$ to 0. Then it executes the following loop for index s going from 1 to k:

Step 1. For each edge e, $c(e) \leftarrow 2c(e) +$ (signed bit b_s of $\bar{c}(e)$). For each vertex v, $y(v) \leftarrow 2y(v) - 1$.

Step 2. Call the *scale_match* routine to find a 1-optimal matching. □

Lemma 2.1(b) shows that the routine halts with a minimum perfect matching. Each iteration of the loop is called a *scale*. We give a *scale_match* routine that runs in $O(\sqrt{n}m)$ time. Since there are $O(\log(nN))$ scales, this achieves the desired time bound.

It is most natural to work with small costs. The *scale_match* routine transforms costs to achieve this. Specifically, *scale_match* changes the cost of each edge vw to $c(vw) - y(v) - y(w)$; then it calls the *match* routine on these costs to find a 1-optimal matching M with duals $y'(v)$; then it adds $y'(v)$ to each dual $y(v)$ ($y(v)$ is the dual value before the call to *match*).

Clearly, M with the new duals is a 1-optimal matching for cost function c. Further, since Step 1 of the main routine changes costs and duals so that the empty matching is 1-feasible, the costs input to *match* are integers -1 or larger. If vw is an edge in the 1-optimal matching found in the previous scale, then after Step 1, $y(v) + y(w) \geq c(vw) - 3$. Hence vw costs at most three in the costs for *match*. Thus there is a perfect matching of cost at most $3n$. (This is true even in the first scale). We will show that if every edge costs at least -1 and a minimum perfect matching costs $O(n)$, *match* finds a 1-optimal matching in $O(\sqrt{n}m)$ time. This gives the desired time bound.

Note that the transformation done by *scale_match* is for conceptual convenience only. An actual implementation would not transform costs; rather *match* would work directly on the untransformed costs.

The *cost-length* of an edge e with respect to a matching M is

$$cl(e) = c(e) + (\textbf{if } e \notin M \textbf{ then } 1 \textbf{ else } 0).$$

The *net cost-length* of a set of edges S with respect to M is

$$cl(S) = \sum\{cl(e) | e \in S - M\} \; - \sum\{cl(e) | e \in S \cap M\}.$$

This quantity equals the net cost of S (with respect to M) plus the number of unmatched edges in S. Hence an augmenting path with smallest net cost-length approximates both the smallest net cost augmenting path and the shortest augmenting path; this is in keeping with our plan for the algorithm.

An edge vw is *eligible* if $y(v) + y(w) = cl(vw)$, i.e., the 1-feasibility constraint for vw holds with equality. (Note that a matched edge is always eligible.) It follows from the analysis below that an augmenting path of eligible edges has the smallest possible net cost-length. Hence the algorithm augments along paths of eligible edges. If no such path exists, it adjusts the duals to create one. The details are as follows.

Assume the costs given to *match* are integers that are at least -1, and there is a perfect matching costing at most an. (In the scaling algorithm $a = 3$.)

procedure *match*.
Initialize all duals $y(v)$ to 0 and matching M to $\emptyset$. Then repeat the following steps until Step 1 halts with the desired matching:

Step 1. Find a maximal set $\mathcal{A}$ of vertex-disjoint augmenting paths of eligible edges. For each path $P \in \mathcal{A}$, augment the matching along P, and for each vertex $w \in V_1 \cap P$, decrease $y(w)$ by 1. (This makes the new matching 1-feasible.) If the new matching M is perfect, halt.

Step 2. Do a Hungarian search to adjust the duals (maintaining 1-feasibility) and find an augmenting path of eligible edges. □

We now give the details of Steps 1 and 2 that are needed to analyze *match*. (A full description of these steps is given below, in the paragraphs preceding the statement of Theorem 2.1. The reader may prefer to examine these details now, although this is not necessary.) Both steps can be implemented in $O(m)$ time. Step 2 is a Hungarian search (essentially Dijkstra's shortest path algorithm, see e.g., [L], [PS]). The search does a number of *dual adjustments*. Each dual adjustment calculates a positive integer δ and increases or decreases various dual values by δ, so as to preserve 1-feasibility and eventually create an augmenting path of eligible edges. (The dual adjustment is defined more precisely below.) At any point in the algorithm define

F = the set of free vertices in V_0;

Δ = the sum of all dual adjustment quantities δ in all Hungarian searches so far.

The Hungarian search maintains the duals so that any free vertex $v \in F$ has $y(v) = \Delta$ and any free vertex $v \in V_1$ has $y(v) = 0$.

To analyze the *match* routine, first observe that it is correct: The changes to the matching (in Step 1) and to the duals (in Steps 1–2) keep M a 1-feasible matching. If M is not perfect but G has a perfect matching, the Hungarian search creates an augmenting path of eligible edges. Hence the algorithm eventually halts with M a 1-optimal matching, as desired. (Note that if G does not have a perfect matching, this is eventually detected in Step 2.)

To analyze the run time, consider any point in the execution of *match*. Let M be the current matching, and define F and Δ as above. Let M^* be a minimum perfect matching. For any set of edges S let $cl(S)$ denote net cost-length with respect to M.

$M^* \oplus M$ consists of an augmenting path P_v for each $v \in F$, plus alternating cycles C_w. Thus

$$n + c(M^*) - c(M) \geq cl(M^* \oplus M) = \sum_{v \in F} cl(P_v) + \sum_{w} cl(C_w). \tag{1}$$

To estimate the right-hand side, consider an alternating path P from $u \in V_0$ to $m \in V_0$, where u is on an unmatched edge of P and m is on a matched edge of P (m stands for "matched"; no confusion should result from the double usage of m). Then

$$y(u) \leq y(m) + cl(P). \tag{2}$$

This follows since for edges $uv \notin M$ and $vm \in M$, $y(u) + y(v) \leq cl(uv)$ and $y(v) + y(m) = cl(vm)$, so $y(u) \leq y(m) + cl(uv) - cl(vm)$. Inequality (2) implies that any alternating cycle C has $cl(C) \geq 0$. It also implies that any augmenting path P_v from some $v \in F$ to some free vertex $t \in V_1$ has $y(v) + y(t) \leq cl(P_v)$. Recall that the Hungarian search keeps $y(v) = \Delta$ and $y(t) = 0$. Hence $\Delta \leq cl(P_v)$, and the right-hand side of (1) is at least $|F|\Delta$.

By assumption on the input to *match*, $c(M^*) \leq an$ and $c(M) \geq -n$. Hence the left-hand side of (1) is at most bn for $b = a + 2$. Thus we have shown

$$|F|\Delta \leq bn. \tag{3}$$

This implies there are $O(\sqrt{n})$ iterations of the loop of *match*. To see this, note that each execution of Step 1 (except possibly the first) augments along at least one path because of the preceding Hungarian search. Hence at most $\sqrt{bn} + 1$ iterations start with $|F| \leq \sqrt{bn}$. From (3), $|F| \geq \sqrt{bn}$ implies $\Delta \leq \sqrt{bn}$. The next paragraph

shows that each Hungarian search increases Δ by at least one. This implies that at most $\sqrt{bn}+1$ iterations start with $\Delta \leq \sqrt{bn}$, giving the desired bound.

Now we show that a Hungarian search $\mathcal{S}$ increases Δ by at least one. It suffices to show that $\mathcal{S}$ does a dual adjustment (since any dual adjustment quantity δ is a positive integer). Search $\mathcal{S}$ does a dual adjustment unless there is an augmenting path P of eligible edges when it starts. Clearly, P intersects some augmenting path found in Step 1. It is easy to see that P contains an unmatched edge vw, with w but not v in an augmenting path of Step 1, and $w \in V_1$. But vw is ineligible after the Step 1 decreases $y(w)$. So P does not exist, and $\mathcal{S}$ does a dual adjustment.

In summary, *match* does $O(\sqrt{n})$ iterations. Each iteration takes time $O(m)$, giving the desired time bound $O(\sqrt{n}m)$.

It remains to give the details of Steps 1 and 2. Step 1 finds the augmenting paths P by depth-first search. To do this, it marks every vertex reached in the search. It initializes a path P to a free unmarked vertex of V_0. To grow P, it scans an eligible edge xy from the last vertex x of P (x will always be in V_0). If y is marked, the next eligible edge from x is scanned; if none exists, the last two edges of P (one matched and one unmatched) are deleted from P; if P has no edges, another path is initialized. If y is free, another augmenting path has been found; in this case y is marked, the path is added to $\mathcal{A}$, and the next path is initialized. The remaining possibility is that y is matched to a vertex z. In this case y and z are marked; edges xy, yz are added to P; and the search is continued from z.

It is clear that this search uses $O(m)$ time. To show that it halts with $\mathcal{A}$ maximal, first observe that for any marked vertex $x \in V_0 - V(\mathcal{A})$, every eligible edge xy has y marked and matched, or y in $V(\mathcal{A})$. (Note that $V(\mathcal{A})$ is the set of vertices in paths of $\mathcal{A}$.) Hence an easy induction shows that an alternating path of eligible edges, starting at a free vertex of V_0 and vertex-disjoint from $\mathcal{A}$, has all its V_0 vertices marked and is not augmenting.

Step 2 is the Hungarian search. It grows a forest $\mathcal{F}$ of eligible edges, from roots F. An eligible edge vw with $v \in V_0 \cap \mathcal{F}$ and $w \notin \mathcal{F}$ is added to $\mathcal{F}$ whenever possible. If w is free, $\mathcal{F}$ contains an augmenting path of eligible edges. Otherwise, the matched edge ww' is added to $\mathcal{F}$. Eventually either $\mathcal{F}$ contains an augmenting path or $\mathcal{F}$ cannot be enlarged.

In the latter case a *dual adjustment* is done. It changes duals in a way that preserves 1-feasibility and allows $\mathcal{F}$ to be enlarged, as follows. It computes the dual adjustment quantity

$$\delta = \min\{cl(vw) - y(v) - y(w) | v \in V_0 \cap \mathcal{F}, w \notin \mathcal{F}\}.$$

Each $v \in \mathcal{F}$ gets $y(v)$ increased by $\delta \times$ (**if** $v \in V_0$ **then** 1 **else** -1). It is easy to see that this achieves the goal of the dual adjustment (an edge vw achieving the above minimum becomes eligible and so can be added to $\mathcal{F}$).

After the dual adjustment, the search continues by enlarging $\mathcal{F}$. Eventually $\mathcal{F}$ contains the desired augmenting path of eligible edges and the Hungarian search halts.

Note that, as claimed above, at any point in the algorithm a free vertex v has $y(v) = \Delta$ if $v \in F$ (since every dual adjustment increases $y(v)$) and $y(v) = 0$ if $v \in V_1$ (no dual adjustment changes $y(v)$).

A Hungarian search can be implemented in $O(m)$ time. This depends on two observations. First, the proper data structure allows a dual adjustment to change all duals $y(v)$ in $O(1)$ time total. Specifically the algorithm keeps track of Δ (defined above). When a vertex v is added to $\mathcal{F}$, its current dual value and the current value

of Δ are saved as $y^0(v)$ and $\Delta(v)$, respectively. Then at any time the current value of $y(v)$ can be calculated as

$$y^0(v) + (\Delta - \Delta(v)) \times (\textbf{if } v \in V_0 \textbf{ then } 1 \textbf{ else } -1).$$

Hence the dual adjustment is accomplished by simply increasing the value of Δ.

The second observation is how to compute δ in a dual adjustment. The usual implementation of a Hungarian search does this with a priority queue that introduces a logarithmic factor into the time bound (e.g., [FT]). This can be avoided when, as in our case, costs are small integers (this was observed in [D], [W] for Dijkstra's shortest path algorithm). The details are as follows. The next value of δ is the amount that the next value of Δ increases from its current value. Hence it suffices to calculate the next value of Δ. The next value of Δ is the smallest possible value such that some edge vw with $v \in V_0 \cap \mathcal{F}$ and $w \notin \mathcal{F}$ becomes eligible (when duals are adjusted by δ). Thus the next value of Δ equals

$$\min\{cl(vw) - y^0(v) - y(w) + \Delta(v) | v \in V_0 \cap \mathcal{F}, w \notin \mathcal{F}\}.$$

Since any Hungarian search has $|F| \geq 1$, inequality (3) implies $\Delta \leq bn$. The algorithm maintains an array $Q[1..bn]$. Each entry $Q[r]$ points to a list of edges vw that can make $\Delta = r$, i.e., $v \in V_0 \cap \mathcal{F}$, $w \notin \mathcal{F}$, and $r = cl(vw) - y^0(v) - y(w) + \Delta(v)$. The algorithm scans down Q and chooses the next value of Δ as the smallest value r with $Q[r]$ nonempty. This gives the next value of δ, and the newly eligible edges, as desired. The total overhead for scanning is $O(n)$, since Q has bn entries. (Note that an edge vw with $v \in V_0 \cap \mathcal{F}$, $w \notin \mathcal{F}$ does not get entered in this data structure if $cl(vw) - y^0(v) - y(w) + \Delta(v) > bn$.)

Only one detail of the derivation remains: We have assumed that the dual values $y(v)$ do not grow too large, so that arithmetic operations use $O(1)$ time. To justify this, we show that each $y(v)$ has magnitude $O(n^2N)$. It suffices to do this for $v \in V_0$. Define Y_s as $\max\{|y(v)||v \in V_0\}$ after the sth scale. Then $Y_0 = 0$ and $Y_{s+1} \leq 2Y_s + bn - 1$ (since $\Delta \leq bn$). Thus $Y_k \leq (2^k - 1)(bn - 1) = O(n^2N)$, as desired. Note that the input uses a word size of at least $\max\{\log N, \log n\}$ bits. Hence at worst the algorithm uses triple-word integers for the dual variables.

THEOREM 2.1. *A minimum perfect matching in a bipartite graph can be found in* $O(\sqrt{n}m \log(nN))$ *time and* $O(m)$ *space.* □

A heuristic that may speed up the algorithm in practice is to prune the graph at the start of each scale. Specifically, *scale_match* can delete any edge whose new cost is $6n$ or more. In proof, recall that in the costs computed by *scale_match* there is a perfect matching M costing at most $3n$; taking into account the low-order bits of cost that are not included in the current scale, the true cost of M is less than $4n$. In the costs computed by *scale_match* every edge costs at least -1; again taking into account low order bits, the true cost is more than -2. Hence a matching containing an edge of new cost $6n$ or more has true cost more than $4n$ and so is not minimum.

2.2. Extensions of the assignment algorithm. The bounds of Theorem 2.1 also apply to finding a minimum-cost matching. To see this, let G be the given graph. Form $\overline{G}$ by taking two copies of G; for each $v \in V(G)$ join the two copies of v by a cost zero edge. Then $\overline{G}$ is bipartite, and a minimum perfect matching in $\overline{G}$ gives a minimum-cost matching in G.

A similar result holds for minimum-cost maximum cardinality matching. The construction is the same except that the edges joining two copies of v cost nN. The

problem of finding a minimum-cost matching of given cardinality can also be solved in the same bounds; it is most convenient to use Theorem 3.2 below.

Returning to perfect matching, several properties of *match* are needed for § 3. Define

$$A = \text{ the total length of all augmenting paths found by } match.$$

We first derive a bound on A. Let P_i denote the ith augmenting path found by *match*. Let ℓ_i be its length, measured as its number of unmatched edges; let Δ_i denote the value of Δ when P_i is found; let M_i be the matching after augmenting along P_i. Recall that in the Hopcroft–Karp algorithm, for some constant c, $\ell_i \le cn/(n-i+1)$. Thus the total augmenting path length is $\sum_{i=1}^{n} \ell_i = O(n \log n)$ [ET]. In *match*, ℓ_i does not have a similar bound. However, it is bounded in an amortized sense, as follows.

LEMMA 2.2. *For any* $k \in [1..n]$, $c(M_k) + \sum_{i=1}^{k} \ell_i = \sum_{i=1}^{k} \Delta_i$.

Proof. A calculation similar to (2) shows that for any i, $\Delta_i = cl(P_i)$. It is easy to see that $cl(P_i) = \ell_i + c(M_i) - c(M_{i-1})$ (assume $c(M_0) = 0$). Summing these relations gives the lemma. □

COROLLARY 2.1. $A = O(n \log n)$.

Proof. Since $|M_k| = k$, the entry conditions for *match* imply $c(M_k) \ge -k$. Hence $A \le n + \sum_{i=1}^{n} \Delta_i$. By (3), $\Delta_i \le bn/(n-i+1)$. Summing these inequalities gives the lemma. □

The second property shows that the depth-first search of Step 1 never encounters a cycle. A similar property for network flows is used in [GoT87a].

LEMMA 2.3. *In* match *there is never an alternating cycle of eligible edges.*

Proof. Initially there are no matched edges, so there are no alternating cycles of eligible edges. In a Hungarian search, whenever the duals of a matched edge vw are changed, $w \in V_1$ gets $y(w)$ decreased. Hence any edge joining w to a vertex not in the search forest $\mathcal{F}$ is ineligible. This implies that the Hungarian search does not create an alternating cycle of eligible edges. Similar reasoning applies when an augment creates a new matched edge and changes duals. □

Some applications of matching require the optimal linear programming dual variables. The dual variables $y(v)$ are *optimal* if there is a perfect matching M such that every edge vw has $y(v) + y(w) \le c(vw)$, with equality for every $vw \in M$. (This implies that M is a minimum perfect matching.) Such duals exist for any bipartite graph having a perfect matching [L], [PS]. The scaling algorithm halts with duals that are 1-optimal but not necessarily optimal. Optimal duals can be found as follows.

Let G^+ be G with an additional vertex $s \in V_0$ and an edge sv for each $v \in V_1$. Extend the given cost function c to G^+ by defining $c(sv)$ as an arbitrary integer; the cost function used by the matching algorithm extends to G^+ by its definition, $\bar{c} = (n+1)c$. To specify a cost function on G^+, we write $G^+; c$ or $G^+; \bar{c}$. Let M be a minimum perfect matching in G; for vertex v, let v' denote its mate, i.e., $vv' \in M$. For $v \in V_0$, let M_v be a minimum perfect matching in $G^+ - v; c$. (Such a matching exists, for instance, $M - vv' + sv'$.) Set

$$y(v) = \textbf{if } v \in V_0 \textbf{ then } -c(M_v) \textbf{ else } c(vv') - y(v').$$

These duals are optimal on G. (This can be proved by an argument similar to the algorithm given below. Alternatively, see [G87] for a proof from first principles.)

Suppose a Hungarian search (as in *match*) is done on $G^+; \bar{c}$. It halts with a tree T of eligible edges, rooted at s. Clearly T is a spanning tree. For any $v \in V_0$, augmenting along the sv-path in T gives a 1-optimal matching N_v in $G^+ - v; \bar{c}$. N_v is a minimum

perfect matching in $G^+ - v; c$. This follows from Lemma 2.1, since $G^+ - v$ and G have the same number of vertices. Hence N_v qualifies as M_v.

In summary, the following procedure finds optimal duals. Given is the output of the matching algorithm, i.e., a 1-optimal matching in $G; \bar{c}$ with duals y. Form $G^+; \bar{c}$, defining $c(sv) = \lceil y(v)/(n+1) \rceil$ for each $v \in V_1$; also set $y(s) \leftarrow 0$ (this gives 1-feasible duals). Do a Hungarian search to construct a spanning tree T of eligible edges rooted at s. Do a depth-first search of T to find $c(M_v)$ for each $v \in V_0$. Define optimal duals $y(v)$ by the above formula.

The time for this algorithm is $O(m)$. This is clear, except perhaps for the time for the Hungarian search. The choice of $c(sv)$ ensures that $\Delta \leq n$. Hence, as in *match*, the Hungarian search can be implemented using an array Q. This gives $O(m)$ time.

COROLLARY 2.2. *Optimal dual variables on a bipartite graph can be found in* $O(\sqrt{n}m \log(nN))$ *time and* $O(m)$ *space.* □

This implies the next result. Consider a directed graph with n vertices, m edges, and arbitrary (possibly negative) edge lengths.

THEOREM 2.2. *The single-source shortest path problem on a directed graph with arbitrary integral edge lengths can be solved in* $O(\sqrt{n}m \log(nN))$ *time and* $O(m)$ *space.*

Proof. This problem can be solved by finding optimal duals on a bipartite graph whose costs are the edge lengths and then running Dijkstra's algorithm [G85]. □

Obviously the same bound holds for $O(\sqrt{n})$ sources.

3. Degree-constrained subgraphs and extensions. This section extends the assignment algorithm to derive the last three bounds of Table 1. Section 3.1 gives an algorithm for the minimum perfect degree-constrained subgraph problem, deriving time bounds for finding a degree-constrained subgraph and for solving the transportation problem. Section 3.2 discusses scaling edge multiplicities, which improves the bounds when edge multiplicities are large. Section 3.3 extends the results to network flow. Throughout § 3, n denotes the number of vertices in the input graph. The problems of §§ 3.1–3.2 are defined on a multigraph. Recall that for a multigraph m denotes the number of edges and $\overline{m}$ the number of edges counting multiplicities.

3.1. The degree-constrained subgraph algorithm. This section gives an algorithm for the perfect degree-constrained subgraph problem. Note that a perfect DCS problem on a multigraph of n vertices and $\overline{m}$ edges can be reduced in linear time to a perfect matching problem on a graph of $O(\overline{m})$ vertices and edges [G87]. Hence Theorem 2.1 immediately implies a bound of $O(\overline{m}^{3/2} \log(\overline{m}N))$ for the DCS problem. We now derive the better bound given in Table 1.

For a DCS D, the *cost-length* of edge e is

$$cl(e) = c(e) + (\textbf{if } e \notin D \textbf{ then } 1 \textbf{ else } 0).$$

A 1-*feasible* DCS is a DCS D and dual variables $y(v)$ for each vertex v, such that for any edge vw,

$$y(v) + y(w) \leq cl(vw), \qquad \text{for } vw \notin D,$$
$$y(v) + y(w) \geq cl(vw), \qquad \text{for } vw \in D.$$

A 1-*optimal* DCS is a perfect DCS that is 1-feasible. (Note that the definition of a 1-feasible matching is slightly different — the second relation holds with equality. The difference is not significant: if we treat a matching problem as a DCS problem, a 1-feasible DCS gives a 1-feasible matching, by lowering duals as necessary to achieve the desired equalities.)

As in Lemma 2.1, if every cost is divisible by k, $k > n/2$, then a 1-optimal DCS is a minimum perfect DCS. This is essentially a result of Bertsekas [Ber79], [Ber86]. In proof, note that a perfect DCS D has minimum cost if any alternating (simple) cycle C has $c(C \cap D) \leq c(C - D)$. This condition can be verified for a 1-optimal DCS D by a calculation similar to Lemma 2.1.

Now we describe the algorithm. Many details are exactly as in § 2, so we elaborate only on the parts that change. All data structures have size $O(m)$ (not $O(\overline{m})$). Clearly the multigraph G can be represented by such a structure.

The main routine works in (at most) $\lfloor \log(n+2)N \rfloor$ scales. (This is justified by the above analog of Lemma 2.1; each original cost is multiplied by $\lceil (n+1)/2 \rceil$.) Steps 1–2 and *scale_match* are unchanged. Let D_0 be the 1-optimal DCS of the previous scale. Note that the *match* routine is called with integral costs $c(e)$ that are at least -1 for $e \notin D_0$ and at most three for $e \in D_0$.

The *match* routine initializes all duals $y(v)$ to 0 and D to $\{e | c(e) < -1\}$. (Clearly D does not violate any degree constraint.) The definition of an *eligible* edge vw is still $y(v) + y(w) = cl(vw)$. Step 1 of *match* finds a maximal set of edge-disjoint augmenting paths of eligible edges such that any vertex v is an end of at most $\phi(v, D)$ paths. (In a multigraph, "edge-disjoint" means a given copy of an edge is in at most one path.) It augments the DCS along each path. Unlike § 2, no duals are changed after an augment; the new DCS is 1-feasible, and the edges on an augmenting path become ineligible. Step 2 does a Hungarian search to adjust duals and find an augmenting path of eligible edges.

Note that this algorithm is correct: Since the Hungarian search maintains 1-feasiblity, the algorithm halts with a 1-optimal DCS (assuming a perfect DCS exists).

Step 1 is implemented by a depth-first search similar to that of § 2, modified for degree constraints larger than one: Each augmenting path P is initialized to a vertex $x \in V_0$ with positive deficiency; x is used to initialize paths P until its deficiency becomes zero or it is deleted from P. P is grown as an alternating path, so that when its last vertex x is in V_0 an edge not in D is scanned, and when x is in V_1 an edge of D is scanned. Instead of vertex marks, each vertex has a pointer to its last unscanned edge. The last edge of P gets deleted if x has no more unscanned edges. It is easy to see the time for Step 1 is $O(\overline{m})$. (As shown below, each augmenting path is simple, although this fact is not needed for correctness.)

The details of the Hungarian search are similar to § 2. The main differences stem from the fact that the search forest $\mathcal{F}$ is grown edge by edge, rather than in pairs of unmatched and matched edges. The time for the search is $O(m)$. This assumes that, as in § 2, an array $Q[1..dn]$ is used to compute minima; here d is the constant of Lemma 3.3, which justifies using the array.

This completes the description of the DCS algorithm. The discussion shows that it is correct. The efficiency analysis uses three inequalities, each analogous to (3) of § 2. To state the inequalities, we use notation similar to that of § 2: D is the DCS at any point in *match*. D_0 is the 1-optimal DCS of the previous scale; hence each of its edges costs at most $a = 3$. F is the set of vertices in V_0 with positive deficiency; Φ is their total deficiency,

$$\Phi = \sum \{\phi(v, D) | v \in F\}.$$

Δ is the sum of all dual adjustment quantities δ in all Hungarian searches so far. Each $x \in F$ has $y(x) = \Delta$. P_x denotes any one of the augmenting paths in $D_0 \oplus D$ that contains x.

LEMMA 3.1. *For some constant b, at any point in* match, $\Phi\Delta \leq bU$.

Proof. The argument of § 2 gives an analog of (1),

$$cl(D_0 \oplus D) \geq \sum\{\phi(v, D)y(v) | v \in F\}.$$

An edge of $D_0 - D$ has cost-length at most $a + 1$; an edge of $D - D_0$ has cost-length at least -1. Hence the lemma holds with $b = (a + 2)/2$. □

The second inequality is for graphs with bounded multiplicity. It generalizes [ET]. Recall that M denotes the maximum multiplicity of an edge in the multigraph.

LEMMA 3.2. *For some constant c, at any point in* match, $\Delta\sqrt{\Phi} \leq cn\sqrt{M}$.

Proof. Set $b = a + 2$. For each integer j define

$$U_j = \{u \in V_0 | y(u) \in [b(j-1)..bj)\},$$
$$W_j = \{w \in V_1 | y(w) - a - 1 \in (-bj.. - b(j-1)]\}.$$

We will show that for any $j \in [1..\lceil \Delta/b + 1 \rceil]$, each P_x has an edge uw with $u \in U_j$, $w \in W_j$. This implies $M|U_j||W_j| \geq \Phi$. Thus $|U_j|$ or $|W_j|$ is at least $\sqrt{\Phi/M}$. Hence $n \geq \sqrt{\Phi/M}(\Delta/b)$, as desired.

To find the desired edge uw of P_x, let the edges in $P_x - D$ be $u_i w_i$, $i = 1, \cdots, k$ (thus $u_1 = x$, and $u_{i+1}w_{i+1}$ follows $u_i w_i$). Since $P_x \subseteq D_0 \oplus D$,

$$\begin{aligned} y(u_i) + y(w_i) &\leq a + 1, \\ y(w_i) + y(u_{i+1}) &\geq -1. \end{aligned} \tag{4}$$

Note that $y(u_1) = \Delta$; $y(u_k) < b$ (by (4) and $y(w_k) = 0$); and $y(u_{i+1}) \geq y(u_i) - b$ (also by (4)). These three inequalities imply that for any $j \in [1..\lceil \Delta/b + 1 \rceil]$, P_x has some $u_i \in U_j$. For a given j, choose the last such i. Then $u_{i+1} \in U_{j-1}$. Together with (4) this implies $w_i \in W_j$, since

$$-bj < -y(u_{i+1}) - b \leq y(w_i) - a - 1 \leq -y(u_i) \leq -b(j-1).$$

We have shown that $w_i \in W_j$ and $u_i \in U_j$, as desired. □

Before giving the third inequality, we note a useful refinement of Lemma 3.2. Let X be a matching such that every edge not in X has multiplicity at most M_X.

COROLLARY 3.1. *For some constant c, at any point in* match, $\Delta\sqrt{\Phi} \leq cn\sqrt{M_X}$.

Proof. The proof is similar to the lemma. We show that for any $j \in [1..\lceil \Delta/b + 1 \rceil]$, each P_x has an edge uw not in X with $u \in U_j \cup U_{j-1}$, $w \in W_j$. This implies $M_X|U_j \cup U_{j-1}||W_j| \geq \Phi$, which leads to the desired conclusion.

To find the desired edge uw for a path P_x, proceed exactly as in the lemma to find an index i with $u_i \in U_j$, $u_{i+1} \in U_{j-1}$, and $w_i \in W_j$. One of the edges $u_i w_i$, $w_i u_{i+1}$ is not in X and can be taken as uw. □

Another bound on Δ is useful for large multiplicities. It is similar to the bound used in [GoT87a]. It justifies using the array $Q[1..dn]$ to compute minima in the Hungarian search.

LEMMA 3.3. *For some constant d, at any point in* match, $\Delta \leq dn$.

Proof. The proof of Lemma 3.2 shows that for any $j \in [1..\lceil \Delta/b + 1 \rceil]$, P_x has some $u \in U_j$. □

COROLLARY 3.2. *The number of iterations of* match *is* $O(\min\{\sqrt{U}, n^{2/3}M^{1/3}, n\})$.

Proof. Each execution of Step 1 (except possibly the first) augments along at least one path, i.e., it decreases Φ by at least one. The definition of Step 1 implies that each Hungarian search (except the last) increases Δ by at least one. Now the

first two bounds of the lemma follow because at any point in the algorithm Δ or Φ is at most B, where Lemma 3.1 gives $B = \sqrt{bU}$ and Lemma 3.2 gives $B = (cn)^{2/3}M^{1/3}$. The third bound follows from Lemma 3.3. □

The corollary implies the following time estimates. The estimates are good for graphs or multigraphs of very small multiplicity.

THEOREM 3.1. *A minimum perfect* DCS *on a bipartite multigraph can be found in* $O(\min\{\sqrt{U}, n^{2/3}M^{1/3}\}\overline{m}\log(nN))$ *time. The space is* $O(m)$. □

For example, in a bipartite graph a minimum perfect DCS can be found in $O(\min\{\sqrt{m}, n^{2/3}\}m\log(nN))$ time.

The bounds of the theorem also apply to finding a minimum-cost DCS. To see this let G be the given multigraph or graph. Form $\overline{G}$ by taking two copies of G and adding a set of edges X, where for each $v \in V(G)$, X contains an edge joining the two copies of v, with multiplicity $u(v) - \ell(v)$ and cost zero. It is easy to see that $\overline{G}$ is bipartite, and a minimum perfect DCS on $\overline{G}$ gives a minimum-cost DCS on G. Furthermore, X is a matching, so Corollary 3.1 applies with $M_X = M$. This implies the time bound of the theorem for minimum-cost DCS.

A similar reduction can be used to find a minimum-cost maximum cardinality DCS. The only difference is that an X-edge costs nN rather than zero. A minimum perfect DCS on this graph $\overline{G}$ induces the desired DCS on (either copy of) G. (In proof, let $\overline{D}$ be a minimum perfect DCS on $\overline{G}$. We can assume that $\overline{D}$ contains the same subgraph D in the two copies of G. Suppose D does not have maximum cardinality. Let P be an augmenting path. Then an alternating cycle C is formed by the two copies of P plus two edges of $\overline{D} \cap X$ that are incident to the two ends of P. Furthermore, $c(\overline{D} \oplus C) < c(\overline{D})$, a contradiction.)

Now we derive bounds that are good for multigraphs with moderately sized multiplicities. First observe that Lemma 2.3 still holds: in *match* there is no alternating cycle of eligible edges. The proof is essentially the same: There is no such cycle initially, since the edges initially in D are ineligible. A Hungarian search does not create such a cycle, since immediately after a dual adjustment a cycle leaving $\mathcal{F}$ on a new eligible edge reenters $\mathcal{F}$ on an ineligible edge.

This fact ensures that the time for a depth-first search in Step 1 is $O(m)$ plus the total augmenting path length. Thus the total time for *match* is $O(mB + A)$, where B is the number of iterations and A is the total augmenting path length. Corollary 3.2 bounds B; now we estimate A.

LEMMA 3.4. $A = O(\min\{U\log U, n\sqrt{MU}\})$.

Proof. As in Corollary 2.1, $A \le U + \sum_{i=1}^{U}\Delta_i$. For the first bound, estimate the summation as in Corollary 2.1, using Lemma 3.1. For the second bound, Lemma 3.2 shows that the summation is at most $\sum_{i=1}^{U} cn\sqrt{M/i} = O(n\sqrt{MU})$. □

THEOREM 3.2. *The transportation problem (capacitated or not) can be solved in* $O((\min\{\sqrt{U}, n^{2/3}M^{1/3}, n\}m + \min\{U\log U, n\sqrt{MU}\})\log(nN))$ *time. The space is* $O(m)$. □

To understand this rather involved time bound, first note that the terms containing M are relevant only in the capacitated transportation problem. The main use of the theorem in this paper is when $U = O(nm)$, in which case the time is $O(nm\log n\log(nN))$; this bound is used in § 3.2 to solve transportation problems with larger U. For further applications we concentrate on the range $M = O(n)$. In this case the above bound for $U = O(nm)$ holds, and also the bound $O(n^2\sqrt{m}\log(nN))$; hence in this range the performance is competitive with [GoT87a]. In most of the range $M = O(n)$, the bounds of Theorem 3.2 are those of Theorem 3.1, with $\overline{m}$ replaced by m: Using $U\log U$ as the second term of the time bound and writing Bm

as the first term, the first term dominates if $U = O(Bm/\log n)$. Hence the bound is $O(n^{2/3}M^{1/3}m\log(nN))$ if $U = O(n^{2/3}M^{1/3}m/\log n)$, e.g., $M = O(n/(\log n)^{3/2})$; the bound is $O(\sqrt{mM}m\log(nN))$ if $U = O(\sqrt{mM}m/\log n)$, e.g., $M = O(m/(\log n)^2)$; the bound is $O(\sqrt{nM}m\log(nN))$ if $U = O(\sqrt{nM}m/\log n)$, e.g., all degree constraints are $O(M)$ and $M = O(m^2/(n(\log n)^2))$.

As in Theorem 3.1, the same bounds hold for networks where each node has an upper and lower bound on its desired degree, and the objective is minimum cost or minimum-cost maximum cardinality.

3.2. Scaling edge multiplicities. In multigraphs with large multiplicities, efficiency is gained by scaling the multiplicities. Let D be a DCS. Recall that for an edge e, $u(e)$ and $d(e)$ denote the multiplicities of e in G and D, respectively; for a vertex v, $u(v)$ and $d(v)$ denote the degree constraint of v and the degree of v in D, respectively. The term *u-value* refers to a multiplicity $u(e)$ or a degree constraint $u(v)$. The approach is to scale u-values. The "closeness lemma" needed for scaling is the following.

Let G be a multigraph with u-values for which D is a minimum-cost maximum cardinality DCS. Form u^+ by adding one to the u-values of an arbitrary subset of vertices and edges (in particular a u-value can increase from zero to one). Let I be the number of increased u-values (so $I \leq m + n$). Let D^+ be a minimum-cost maximum cardinality DCS for u^+. Let $D^+ \oplus D$ denote the subgraph that is the direct sum of subgraphs D^+ and D (i.e., for any edge e, $D^+ \oplus D$ has $|d^+(e) - d(e)|$ copies of e). Choose D^+ so that $|D^+ \oplus D|$ is as small as possible. $\phi^+(v, D)$ denotes the deficiency of D at v for u-values u^+.

LEMMA 3.5. *$D^+ \oplus D$ can be partitioned into at most I simple alternating paths and cycles (where "alternating" means with respect to D and D^+).*

Proof. Since both D^+ and D are DCSs for u^+, $D^+ \oplus D$ can be partitioned into simple alternating paths and cycles; for each vertex v, at most $\phi^+(v, D^+)$ paths end at v on a D-edge, and similarly for a D^+-edge. Call an edge vw with $d^+(vw) > d(vw)$ *new* if either

(i) $d(vw) = u(vw)$, or

(ii) vw is an end of a path of $D^+ \oplus D$ and $d(v) = u(v)$.

There are at most I new edges. (A type (i) new edge clearly has an increased u-value. For a type (ii) new edge vw, v has an increased u-value and $\phi^+(v, D) = 1$, so vw is the only type (ii) edge associated with v.) Thus it suffices to show that any alternating path or cycle P of $D^+ \oplus D$ contains a new edge.

P does not begin and end with a D-edge, since D^+ has maximum cardinality. Suppose P does not contain a new edge. Then $D \oplus P$ is a feasible DCS for u. (This follows, since a D^+-edge vw of P has $d(vw) < u(vw)$; further, if this edge vw is an end of P, then $d(v) < u(v)$.) P does not begin and end with a D^+-edge, since D has maximum cardinality. Thus P is an even length alternating path or cycle. This implies that P has zero net cost (with respect to D or D^+). But this contradicts the fact that $|D^+ \oplus D|$ is as small as possible. □

It is convenient to define a new cost function $c_N(e) = c(e) - nN$. Here, as usual, N denotes the largest magnitude of an edge cost. Observe that D^+ is a minimum-cost DCS for c_N. (To prove this, we show that any DCS F with fewer edges than D^+ is not minimum: F has an augmenting path P. The DCS $F \oplus P$ has $c_N(F \oplus P) \leq c_N(F) + (n-1)N - nN < c_N(F)$.)

Lemma 3.5 indicates that $D^+ \oplus D$ can be found in a multigraph G' that models alternating paths. More precisely G' is defined as follows. A vertex $v \in V(G)$ corre-

sponds to v_1, $v_2 \in V(G')$; G' has an edge v_1v_2 of cost 0 and multiplicity I. An edge $vw \in E(G)$ corresponds to edges v_1w_1, $v_2w_2 \in E(G')$, with multiplicities and costs

$$u'(v_1w_1) = u^+(vw) - d(vw), \quad c'(v_1w_1) = c_N(vw);$$
$$u'(v_2w_2) = d(vw), \quad c'(v_2w_2) = -c_N(vw).$$

Finally, each $v \in V(G)$ has upper- and lower-degree constraints $u'(v_1) = u'(v_2) = I$, $\ell'(v_1) = 0$, $\ell'(v_2) = I - \phi^+(v, D)$.

Edges of type v_1v_2 are called *non-G-edges*, and all other edges are *G-edges*; edges of type v_2w_2 are called *D-edges*, and type v_1w_1 are *non-D-edges*. Observe that G' is bipartite, since a cycle has an even number of non-G-edges and the G-edges give a (not necessarily simple) cycle of G.

The desired subgraph D^+ can be chosen as $D' \oplus D$, where D' is a minimum-cost DCS on G'. To prove this, we will show two properties:

(a) A DCS D^+ for u^+ gives a DCS D' on G' of cost $c_N(D^+) - c_N(D)$.

(b) A minimum-cost DCS D' on G' gives a DCS for u^+ of c_N-cost $c_N(D) + c'(D')$.

To see that (a) and (b) suffice, observe that (a) implies $c_N(D^+) - c_N(D) \geq c'(D')$, (b) implies $c_N(D) + c'(D') \geq c_N(D^+)$, implying equality in both relations.

For (a), D' consists of the D-edges of $D - D^+$ and the non-D-edges of $D^+ - D$; additionally, for each vertex v, D' has $I - k$ copies of edge v_1v_2, where v is on k of the I paths and cycles of $D^+ \oplus D$ given by Lemma 3.5. Note that the lower bound constraint for v_2 is satisfied, since D^+ has at most $\phi^+(v, D)$ more non-D-edges than D-edges.

Part (b) follows from the observation that the G-edges of D' can be partitioned into at most I paths and cycles that are alternating for D, and that $D \oplus D'$ is a feasible DCS. The lower bounds $\ell'(v_2)$ ensure that $D \oplus D'$ satisfies all upper bounds u^+.

Now we can state the *capacity-scaling algorithm* for finding a minimum perfect DCS. Given a DCS problem on a multigraph G, let $\overline{u}$ denote the given u-values, with M the largest $\overline{u}$-value. (Without loss of generality, M is the $\overline{u}$-value of a vertex.) Consider each $\overline{u}$-value to be a binary number $b_1 \cdots b_k$ of $k = \lfloor \log M \rfloor + 1$ bits. The routine maintains u as the u-values in the current scale. Each scale constructs a minimum-cost maximum cardinality DCS D for u; d is the function corresponding to D. The routine initializes each $u(e)$, $d(e)$ and each $u(v), d(v)$ to zero. Then it executes the following loop for scale index s going from 1 to k:

Step 1. For each edge e, $d(e) \leftarrow 2d(e)$ and $u(e) \leftarrow 2u(e)+$ (bit b_s of $\overline{u}(e)$). For each vertex v, $u(v) \leftarrow 2u(v)+$ (bit b_s of $\overline{u}(v)$).

Step 2. Form the multigraph G' defined above. (Note that the function u^+ in the definition of G' is given by the function u constructed in Step 1; increased u-values correspond to bits b_s that are one in Step 1.)

Step 3. Let D' be a minimum-cost DCS on G'. Set $D \leftarrow D \oplus D'$, and let d be the function corresponding to D. □

To see that this algorithm is correct, note that the subgraph D constructed in Step 3 is a minimum-cost maximum cardinality DCS for u, by the above discussion. Hence in the last scale, D is the desired minimum perfect DCS. (Note that the algorithm works on both bipartite and general graphs.)

To estimate the running time, assume that Step 3 uses the cost-scaling algorithm of Theorem 3.2 to find the minimum-cost DCS. Noting that $U = O(nm)$ gives the following.

THEOREM 3.3. *The transportation problem (capacitated or not) can be solved in* $O(nm \log n \log(nN) \log M)$ *time. The space is* $O(m)$. □

This result extends to the variants of the perfect DCS problem mentioned above.

Next consider the *transportation problem with cost functions.* This problem allows parallel copies of an edge to have different costs. Specifically the cost of the pth copy of an edge e, $1 \le p \le u(e)$, is given by $c(e,p)$, a nondecreasing function of p that can be evaluated in $O(1)$ time. As usual, these costs are in $[-N..N]$, and each vertex v has a desired degree $u(v)$. The problem is to find a minimum-cost perfect DCS for these degree constraints. Note that the desired DCS can still be represented by an integral function on the edges $d(e)$, where $0 \le d(e) \le u(e)$, since we can assume that the DCS contains the $d(e)$ smallest cost copies of e.

As examples of this problem, $c(e,p) = \lfloor a_e p \rfloor + b_e$ is the original DCS problem for $a_e = 0$, and a simple example of diminishing returns to scale for $a_e > 0$. Alternatively, $c(e,p)$ could be, say, a piecewise quadratic function; in this case evaluating $c(e,p)$ for arbitrary p would probably involve a binary search on the breakpoints. (Note that in the definition of the transportation problem with cost functions, the restriction to nondecreasing cost functions $c(e,p)$ is crucial: without it the problem is NP-hard [GJ, p. 214]. Also note that the solution to the problem is a multigraph with integral multiplicities, by definition. This assumption of integrality is also crucial. This issue is discussed further after Theorem 3.5 for network flows, where real-valued flows make sense.)

In a trivial sense, the algorithm of Theorem 3.3 solves the transportation problem with cost functions — just treat parallel copies of an edge with different costs as different edges. The disadvantage of this approach is that in the time bound, the term m must count each edge e according to the number of distinct costs $c(e,p)$. We show how to avoid this: We extend the capacity-scaling algorithm to the transportation problem with cost functions, preserving the time bound of Theorem 3.3.

First we modify the cost-scaling algorithm to preserve the time bounds of Theorems 3.1–3.2. The derivation of those theorems remains valid for cost functions and gives the desired time bounds, provided all individual steps are implemented to run in essentially the same time as before. This means implementing Step 1 of the main routine and *scale_match* in time $O(m)$ (even though they modify every cost) and similarly for *match.* This can be done because of the following observation. When there are cost functions, the conditions for a DCS D to be 1-feasible are equivalent to a system involving only two inequalities per edge $e = vw$,

$$c(e, d(e)) \le y(v) + y(w) \le c(e, d(e) + 1) + 1.$$

Furthermore, the only copies of e that can be eligible are D-edges costing $c(e, d(e))$ and non-D-edges costing $c(e, d(e) + 1) + 1$.

Step 1 of the main routine and *scale_match* do not explicitly modify edge costs. Instead, *match* computes the cost of an edge when it is needed, in $O(1)$ time using arithmetic operations. Specifically, the pth cost for vw is

$$\bar{c}(vw, p) \div 2^{k-s} - y_0(v) - y_0(w), \tag{5}$$

where $\div$ denotes integer division, $k = \lfloor \log(n+2)N \rfloor$ is the number of cost scales, s is the index of the current cost scale, and y_0 denotes duals at the start of scale s.

The *match* routine starts by initializing D to contain all edges costing less than -1. This is done by examining each edge and adding smallest cost copies to D until

the cost reaches -1. The time is $O(m+U)$, which suffices for the bounds of Theorems 3.1–3.2.

In the depth-first search of Step 1, it is unnecessary to know the multiplicity of each eligible edge when the search begins. Rather, costs $c(e, d(e))$ and $c(e, d(e)+1)$ are used to determine which edges have at least one eligible copy. When the depth-first search finds an augmenting path P, the next cost for each edge $e \in P$ is used to see if there is another eligible copy of e (i.e., for $e \in P \cap D$, another copy of e is eligible if $c(e, d(e) - 1) = c(e, d(e))$, and similarly for $e \in P - D$). Thus the time for the depth-first search is still $O(m)$ plus the total augmenting path length. It is obvious that the Hungarian search, given costs $c(e, d(e))$ and $c(e, d(e) + 1)$, uses time $O(m)$. Thus the bounds of Theorems 3.1–3.2 apply.

Now we modify the capacity-scaling algorithm of Theorem 3.3. The new version works by scaling the domain of the cost functions. The closeness lemma (Lemma 3.5) generalizes as follows. Let G be a multigraph with cost functions c and u-values for which D is a minimum-cost maximum cardinality DCS. Form u^+ by adding one to the u-values of an arbitrary set of vertices and edges. Form c^+ so that for each edge e and $p \in [0..u^+(e))$,

$$c(e, p+1) \geq c^+(e, p+1) \geq c(e, p). \tag{6}$$

(Here $c(e, 0) = -\infty$, $c(e, u(e) + 1) = \infty$.) Let I be the number of vertices with an increased u-value plus the number of edges with an increased u-value or some decreased c-value (so that $I \leq m + n$). Let D^+ be a minimum-cost maximum cardinality DCS for c^+ and u^+, chosen so that $|D^+ \oplus D|$ is minimum ($D^+ \oplus D$ has the obvious interpretation).

LEMMA 3.6. *$D^+ \oplus D$ can be partitioned into at most I simple alternating paths and cycles.*

Proof. The argument is an expanded version of Lemma 3.5. We will explicitly state only the new material. The definition of *new edge* is expanded to include a type (iii) new edge e, defined to have $d^+(e) > d(e)$ and $c(e, d(e) + 1) > c^+(e, d(e) + 1)$, where by definition only the $d(e) + 1$st copy of e is new. (Note that $d^+(e) - d(e)$ may be larger than one.)

The argument remains unchanged until the end, when P is an even length alternating path or cycle not containing a new edge, and we must show that it has zero net cost (with respect to c^+ and D^+). The net cost of P with respect to c^+ and D^+ is nonnegative, by the minimum-cost property of D^+. Hence it suffices to show that the net cost of P with respect to c^+ and D is nonnegative.

This follows from the minimum-cost property of D if for every edge e whose pth copy is in $P \cap D^+$, $c^+(e, p) \geq c(e, d(e) + 1)$. We prove this inequality as follows. The pth copy of e is not new and $p \geq d(e) + 1$. Consider two cases: If $p = d(e) + 1$ then $c^+(e, p) = c(e, d(e) + 1)$, as desired. If $p > d(e) + 1$ then $c^+(e, p) \geq c(e, d(e) + 1)$ by (6), as desired. □

This lemma justifies an algorithm similar to the capacity-scaling algorithm. The main differences are as follows. Step 1, in addition to scaling d and u, scales the cost function domain. Specifically let c_0 denote the given cost function. Then for each e and $p \in [1..u(e)]$ (where $u(e)$ is the new u-value) Step 1 sets $c(e, p) \leftarrow c_0(e, 2^{k-s}p)$, where $k = \lfloor \log M \rfloor + 1$ is the number of capacity scales and s is the index of the current capacity scale. Observe that the DCS corresponding to the new (scaled) d is a minimum-cost maximum cardinality DCS for the new (scaled) u-values rounded down to even numbers and the new costs c with $c(e, 2p - 1)$ increased to $c(e, 2p)$. So

Lemma 3.6 applies and justifies the remaining steps: Step 2 defines G' as before but with costs changed in the obvious way to take cost functions into account. Step 3 computes D' using the cost-scaling algorithm described above.

For efficiency, these three steps are not done explicitly. (For instance, doing Step 2 explicitly would use $\Theta(m^2)$ time, since an edge can be in G' with multiplicity $m+n$.) Step 1 computes only two new costs for each edge e, $c(e, d(e))$ (already known) and $c(e, d(e)+1)$. To do Step 2, G' is initialized to contain only the cheapest copy of each edge of type v_1w_1, v_2w_2. This is the copy that will be added to the DCS D' first. Each copy comes from a cost computed in Step 1. When the cost-scaling algorithm checks to see if there is another eligible copy of an edge e (in the depth-first search), the next higher (or lower) cost copy of e is computed (by the formula of Step 1) and the cost is scaled down using (5).

THEOREM 3.4. *The transportation problem (capacitated or not) with cost functions can be solved in $O(nm \log n \log(nN) \log M)$ time. The space is $O(m)$.* □

We close this section with a variant of the capacity-scaling algorithm of Theorem 3.3. It will be useful in the next section for flow problems with lower bounds. The variant is essentially the (capacity scaling) mincost flow algorithm of Edmonds and Karp [EK]. For completeness we sketch this algorithm, which we call EK (capacity) scaling.

It is convenient to describe EK scaling in terms of two well-known ideas, which we now summarize. The algorithm could be given in terms of 1-feasibility, but it is more natural to use optimal dual variables. Analogous to § 2.2 for matching, optimal duals satisfy the 1-feasible inequalities with cost-length *cl* replaced by cost c. Specifically, variables $y(v)$, $v \in V$, are *optimal* for a DCS D if for any edge vw, $y(v)+y(w) \le c(vw)$ if $vw \notin D$ and $y(v)+y(w) \ge c(vw)$ if $vw \in D$. Any graph with a perfect DCS has a minimum perfect DCS with corresponding optimal duals. (Optimal duals correspond to the optimal linear programming dual variables.)

The classic Hungarian search for the Hungarian matching/DCS algorithm works with such optimal dual variables (see [L], [PS]). In contrast, the Hungarian search of § 3.1 uses 1-feasible duals. The difference in the two Hungarian searches is essentially the definition of "eligible": § 3.1 uses cost-length in the definition of eligible where the standard Hungarian search uses cost. In either case, the purpose of the Hungarian search is to adjust duals, preserving 1-feasibility or optimality as appropriate, and find an augmenting path of eligible edges. A Hungarian search (with optimal duals) can be done in time $O(m + n \log n)$ using Fibonacci heaps [FT].

Hungarian search can be used to do sensitivity analysis for the DCS problem. We will need two sensitivity problems: Given is a minimum perfect DCS D with corresponding optimal duals y. The first problem is to increase the degree constraints of two vertices v, w each by one, and update D and y (if a perfect DCS exists). The second problem is to add an edge vw to the multigraph and update D and y. Both problems can be solved in the time for one Hungarian search. We briefly sketch the algorithms.

First, suppose $u(v)$ and $u(w)$ are each increased by one. Do a Hungarian search from v. Eventually the search finds an augmenting path P of eligible edges from v to w. (The augmenting path can only end at w. If no such path is found, there is no perfect DCS for the new degree constraints.) The algorithm augments along P.

Next, suppose edge vw is added. If $y(v) + y(w) \le c(vw)$, then D and y remain optimal. Suppose $y(v) + y(w) > c(vw)$. Add new vertices v', w', and new edges vv', ww' with $c(vv') = c(ww') = 0$; set $y(v') = -y(v)$, $y(w') = -y(w)$. Do a Hungarian search from v', to adjust duals and find an augmenting path of eligible edges. Note that

the search lowers $y(v)$. If at some point $y(v)$ is lowered so that $y(v) + y(w) \leq c(vw)$ the search stops, since now vw can be added to the graph. Otherwise, the search finds an augmenting path P of eligible edges from v' to w'. (Note that if there is no augmenting path from v' to w', the Hungarian search can eventually lower $y(v)$ so that the first case holds.) The algorithm augments along P and adds vw to the DCS. (This is permissible, since $y(v) + y(w) > c(vw)$.) Finally, the new vertices and edges are deleted.

The EK-scaling algorithm scales capacities similar to the capacity-scaling algorithm. Since it works with optimal duals, it modifies the graph of each scale to ensure that a perfect DCS exists. This is done by using the graph $\overline{G}$ (defined in § 3.1): $\overline{G}$ consists of two copies of G plus edges X, where for each $v \in V(G)$, X contains an edge joining the two copies of v, with multiplicity $u(v)$ and cost nN. Note that (as in § 3.1) a minimum perfect DCS on $\overline{G}$ induces a minimum-cost maximum cardinality DCS on G. Hence in the last scale, the desired minimum perfect DCS is found.

Now we present the EK-*scaling algorithm.* It finds a minimum perfect DCS. Given a DCS problem on a multigraph G, define $\overline{u}$, M, k as in the capacity-scaling algorithm. The routine maintains u as the u-values in the current scale. Each scale constructs a minimum perfect DCS D for the graph $\overline{G}$; d is the function corresponding to D. The routine initializes each $u(e)$, $d(e)$, and each $u(v)$ to zero. Then it executes the following loop for scale index s going from 1 to k:

Step 1. For each $e \in E(\overline{G})$, $d(e) \leftarrow 2d(e)$ and $u(e) \leftarrow 2u(e)$. For each $v \in V(\overline{G})$, $u(v) \leftarrow 2u(v)$.

Step 2. For each $e \in E(G)$ such that the binary expansion of $\overline{u}(e)$ has bit $b_s = 1$, do the following: For each copy of G, add one to the copy of $u(e)$ and add another copy of e, updating D and y using the above routine for adding an edge.

Step 3. For each $v \in V(G)$ such that the binary expansion of $\overline{u}(v)$ has bit $b_s = 1$, do the following: Add another copy of the edge joining both copies of v to $\overline{G}$. Update D and y using the above routine for adding an edge. Then add one to both copies of $u(v)$, and update D and y using the above routine for increasing upper bounds. □

The correctness of EK scaling follows from the fact that it maintains a set of optimal duals on $\overline{G}$ for u and D. Note that in Step 3 in the update routine for increasing upper bounds, an augmenting path always exists: If the edge vv was added to D in the routine for adding an edge, the augmenting path that was used can now be reused.

In problems where each scale has $I = O(n)$, the total time for EK scaling is $O(n(m + n \log n) \log M)$, slightly improving Theorem 3.3. We will encounter such problems in the next section.

3.3. Network flow. This section extends the results to integral network flows. It is convenient to work with the problem of finding a minimum-cost circulation, defined as follows [L]. Let G be a directed graph where each vertex v has a nonnegative integral capacity $u(v)$, and each edge e has a nonnegative integral capacity $u(e)$, a lower bound $\ell(e)$ and a cost $c(e)$. The *minimum circulation problem* is to find a feasible circulation with smallest possible cost. (If vertex capacities are not given, setting $u(v) = \sum_{vw} u(vw)$ does not change the problem. The circulation problem includes the minimum-cost flow problem as a special case. As already mentioned, the usual definition of the circulation (network flow) problem allows real-valued parameters. However, note that if all capacities and lower bounds are integral, an optimum

circulation (flow) that is integer-valued always exists [L].)

A minimum circulation problem on a network G can be transformed to a minimum perfect DCS problem on a bipartite multigraph B, as follows. A vertex $v \in V(G)$ corresponds to v_1, $v_2 \in V(B)$; B has an edge v_1v_2 of cost 0 and multiplicity $u(v)$. An edge $vw \in E(G)$ corresponds to $v_1w_2 \in E(B)$ with cost $c(vw)$ and multiplicity $u(vw) - \ell(vw)$. The degree constraints on B are

$$u(v_1) = u(v) - \sum\{\ell(vw) | vw \in E(G)\},$$
$$u(v_2) = u(v) - \sum\{\ell(wv) | wv \in E(G)\}.$$

A circulation on G corresponds to a perfect DCS on B costing less by exactly $\sum\{\ell(e)c(e) | e \in E(G)\}$. Thus the flow problem can be solved using the DCS algorithms given above. Note that B has n vertices in each vertex set, $O(m)$ edges, $U = O(\sum\{u(v) | v \in V(G)\})$ and $\overline{m} = O(U + \sum\{u(e) | e \in E(G)\})$. In part (a) below, $\overline{m}$ is the number of edges, with each edge counted according to its capacity.

THEOREM 3.5. *A minimum-cost circulation on a network with all edge capacities and lower bounds in* $[0..M]$ *can be found in the following time bounds (and space* $O(m)$*):*

(a) $O(\min\{\sqrt{\overline{m}}, n^{2/3}M^{1/3}\}\overline{m}\log(nN))$.

(b) $O((\min\{\sqrt{mM}, n^{2/3}M^{1/3}, n\}m + \min\{mM\log(mM), n^2\sqrt{m}\})\log(nN))$.

(c) $O(nm\log n\log(nN)\log M)$.

These bounds also hold when each edge cost is a convex function of its flow.

Proof. These bounds follow essentially from Theorems 3.1–3.4. Note that M does not necessarily bound the multiplicities in B, since we assume no bound on vertex capacities in G. Nonetheless, the bound for part (b) holds. To show this, use Corollary 3.1, with matching X containing all edges of the form v_1v_2; note that $M_X = M$. Also the bound for part (c) holds: There are $\log(mM)$ capacity scales, but the time bound involves the factor $\log M$, because each of the first $\log m$ scales is trivial. □

Note that in Theorem 3.5(c), the algorithm for convex cost functions finds an optimal integral-valued flow. However, this flow need not be the global optimum, which may involve real-valued flow values. Finding this solution appears to be much harder. For instance, if the cost of an edge is a quadratic function of its flow, finding a minimum-cost flow is NP-hard [GJ], [H].

Next, consider a minimum circulation problem in which $O(n)$ vertices and edges have finite capacity. As usual, every edge has a lower bound, perhaps zero. Such problems arise as covering problems; a common special case is circulations with lower bounds but no upper bounds (e.g., the aircraft scheduling problem of [L, p. 139]).

THEOREM 3.6. *A minimum circulation on a network with lower bounds but only* $O(n)$ *finite capacities, all lower bounds and finite capacities in* $[0..M]$*, can be found in* $O(n(m + n\log n)\log(nM))$ *time and* $O(m)$ *space.*

Proof. Without loss of generality assume that no cycle has negative cost and infinite capacity. (Such a cycle can be detected in time $O(nm)$ using Bellman's algorithm [Bel].) Recall that a circulation can be decomposed into flows around cycles [Tarj]. Hence it is easy to see that all infinite capacities (on edges or vertices) can be replaced by any number that is at least $S = \sum\{\textbf{if } c(e) \text{ is finite } \textbf{then } c(e) \textbf{ else } \ell(e) | e \in E(G)\} + \sum\{c(v) | v \in V(G),\ c(v) \text{ is finite}\}$.

The algorithm is as follows: Find S and set $k = \lceil \log S \rceil$. For each infinite capacity vertex v, redefine its capacity to S; for each infinite capacity edge e, redefine its

capacity to $\ell(e) + 2^{k+1}$. Transform the new circulation problem into a DCS problem, as above, and solve the DCS problem using EK scaling.

The correctness of this algorithm follows from the definition of S. To estimate the efficiency, note that in the DCS problem, every infinite capacity edge of G has multiplicity 2^{k+1} and every vertex v_1, v_2 has degree constraint at most $S \leq 2^k$. Hence the first scale is trivial — no edges are in the DCS, and the duals can be set to any sufficiently small values (say $\min\{c(e)/2 | e \in E\}$). Every scale after the first has $I = O(n)$ (recall that I is the number of increased u-values), since the u-values of infinite capacity edges double. Hence the total time is $O(n(m + n \log n) \log S)$, implying the desired bound. □

The term $\log(nM)$ in the time bound can be replaced by $\log M$, or more precisely, $(1 + \log(S/n))$. This modified bound is an asymptotic improvement for S very close to n. The modified bound follows because, although there are $\log S$ scales, the first $\log n$ scales do $O(n)$ augmentations. (This in turn holds, since every unit of I in the first $\log n$ scales contributes at least S/n to the sum for S. Actually, to achieve this requires a slight modification to the algorithm: the capacity of an infinite capacity vertex v is changed to $2^k + \sum\{\ell(e) | e \text{ is incident to } v\}$.) Bounds similar to this are in [EK], [AL].

As an example of an application of these bounds, consider the directed Chinese postman problem. A complete definition of the problem is given in [EJ], [PS]; it is a special case of the above problem with $S = O(m)$. The theorem gives time $O(n(m+n \log n) \log n)$ for this problem; the modified bound is slightly better, $O(n(m+n \log n) \log(m/n))$. (For instance, it is easy to see that the modified bound is no worse than $O(nm \log n)$.) Aho and Lee [AL] give a complete discussion of covering problems such as this one.

4. Concluding remarks. Table 1 shows that in terms of asymptotic estimates, many network problems can be solved efficiently by scaling. Scaling algorithms also tend to be simple to program. For instance, the assignment algorithm consists of an outer scaling loop plus an inner loop that does a depth-first search, followed by a Dijkstra calculation. We believe that such algorithms will run efficiently in practice. Note that in the experiments done by Bateson [Ba] the scaling algorithm of [G85] ran faster than the Hungarian algorithm as long as the cost of the matching could be stored in a machine integer. Our assignment algorithm has even simpler code than [G85] and so should do even better.

We have extended the assignment algorithm in three other directions. The first direction is parallel computation. Almost-optimum speedup can be achieved for a large number of processors. Specifically, the time bound for the assignment problem improves by a factor of $(\log(2p))/p$ for a version of the algorithm running on an EREW PRAM with p processors, for $p \leq m/(\sqrt{n} \log^2 n)$. Details are in [GabT88]. The second direction is matroid generalizations of bipartite matching, such as the independent assignment problem and weighted matroid intersection. As in this paper, time bounds very close to the best-known bounds for the cardinality versions of the problems can be achieved; see [GX89a], [GX89b]. The third direction is matching on general graphs. The time bound for finding a minimum perfect matching on a general graph is $O(\sqrt{n\alpha(m,n) \log n}\ m \log(nN))$. The algorithm is more complicated than the assignment algorithm because of "blossoms" that occur in general matching. Blossoms compound the error due to scaling. Details are in [GabT89].

Since the initial writing of this paper several related results have also been obtained by others. Orlin and Ahuja [OA] discovered an alternative $O(\sqrt{nm} \log(nN))$-

time algorithm for the assignment problem. Their algorithm uses our scaling approach in combination with a hybrid inner loop that uses the Goldberg–Tarjan "preflow-push" method in a first phase and single augmentations in a second phase. (We chose not to use this approach because of the conceptual and practical advantages of a uniform algorithm.) Orlin and Ahuja also show how to use their assignment algorithm (or ours) to find a minimum average cost cycle in a directed graph with edge costs, in the same time bound. Ahuja, Goldberg, Orlin, and Tarjan [AGOT] have studied other double scaling algorithms for the minimum-cost flow problem. Their fastest algorithm runs in $O(nm \log(nN) \log\log M)$ time with sophisticated data structures or $O(nm \log M(1 + \log(nN)/\log\log M))$ time with simple data structures. Aho and Lee [AL] have investigated the use of Edmonds–Karp scaling in covering problems, as already mentioned in § 3.3.

Acknowledgment. We thank Andrew Goldberg for sharing his ideas, which inspired this work.

REFERENCES

[AGOT] R.K. AHUJA, A.V. GOLDBERG, J.B. ORLIN AND R.E. TARJAN, *Finding minimum-cost flows by double scaling*, Tech. Report CS-TR-164-88, Dept. of Comput. Sci., Princeton University, Princeton, NJ, 1988; submitted for publication.

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

[AL] A.V. AHO AND D. LEE, *Efficient algorithms for constructing testing sets, covering paths, and minimum flows*, unpublished manuscript, 1988.

[Ba] C.A. BATESON, *Performance comparison of two algorithms for weighted bipartite matching*, M.S. thesis, Dept. of Comput. Sci., University of Colorado, Boulder, CO, 1985.

[Bel] R.E. BELLMAN, *On a routing problem*, Quart. Appl. Math, 16 (1958), pp. 87–90.

[Ber79] D.P. BERTSEKAS, *A distributed algorithm for the assignment problem*, unpublished working paper, Laboratory for Information and Decision Sciences, Mass. Inst. of Technology, Cambridge, MA, 1979.

[Ber86] ———, *Distributed asynchronous relaxation methods for linear network flow problems*, LIDS Report P-1606, Mass. Inst. of Technology, Cambridge, MA, 1986; preliminary version in Proc. 25th IEEE Conference on Decision and Control, December 1986.

[Ber87] ———, *The auction algorithm: A distributed relaxation method for the assignment problem*, LIDS Report P-1653, Mass. Inst. of Technology, Cambridge, MA, 1987.

[D] R.B. DIAL, *Algorithm* 360: *Shortest path forest with topological ordering*, Comm. Assoc. Comput. Mach., 12 (1969), pp. 632–633.

[EJ] J. EDMONDS AND E.L. JOHNSON, *Matching, Euler tours and the Chinese postman*, Math. Programming, 5 (1973), pp. 88–124.

[EK] J. EDMONDS AND R.M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[ET] S. EVEN AND R.E. TARJAN, *Network flow and testing graph connectivity*, SIAM J. Comput., 4 (1975), pp. 507–518.

[FT] 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.

[G85] H.N. GABOW, *Scaling algorithms for network problems*, J. Comput. System Sci., 31 (1985), pp. 148–168.

[G87] ———, *Duality and parallel algorithms for graph matching*, unpublished manuscript, 1987.

[GabT88] H.N. GABOW AND R.E. TARJAN, *Almost-optimum speed-ups of algorithms for bipartite matching and related problems*, Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 514–527; submitted for publication.

[GabT89] ———, *Faster scaling algorithms for general graph matching problems*, Tech. Report CU-CS-432-89, Dept. of Comput. Sci., University of Colorado, Boulder, CO, 1989; submitted for publication.

[GX89a] H.N. GABOW AND Y. XU, *Efficient theoretic and practical algorithms for linear matroid intersection problems*, Tech. Report CU-CS-424-89, Dept. of Comput. Sci., University of Colorado, Boulder, CO, 1989; submitted for publication.

[GX89b] ———, *Efficient algorithms for independent assignment on graphic and linear matroids*, Proc. 30th Annual Symposium on Foundations of Computer Science, 1989, to appear.

[GalT] Z. GALIL AND É. TARDOS, *An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm*, Proc. 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 1–9.

[GJ] M.R. GAREY AND D.S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco, CA, 1979.

[Go] A.V. GOLDBERG, *Efficient graph algorithms for sequential and parallel computers*, Ph. D. dissertation, Dept. of Electrical Engrg. and Comput. Sci., Mass. Inst. of Technology, Tech. Report MIT/LCS/TR-374, Cambridge, MA, 1987.

[GoT86] A.V. GOLDBERG AND R.E. TARJAN, *A new approach to the maximum flow-problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[GoT87a] ———, *Solving minimum-cost flow problems by successive approximation*, Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 7–18.

[GoT87b] ———, *Finding minimum-cost circulations by successive approximation*, Tech. Report CS-TR-106-87, Dept. of Comput. Sci., Princeton University, Princeton, NJ, 1987; Math. Oper. Res., to appear.

[H] P.P. HERRMANN, *On reducibility among combinatorial problems,* Report No. TR-113, Project MAC, Mass. Inst. of Technology, Cambridge, MA, 1973.

[HK] J. HOPCROFT AND R. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[K55] H.W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–97.

[K56] ———, *Variants of the Hungarian method for assignment problems*, Naval Res. Logist. Quart., 3 (1956), pp. 253–258.

[L] E.L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[OA] J.B. ORLIN AND R.K. AHUJA, *New scaling algorithms for the assignment and minimum cycle mean problems*, Sloan Working Paper No. 2019-88, Sloan School of Management, Mass. Inst. of Technology, Cambridge, MA, 1988.

[PS] C.H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[Tard] É. TARDOS, *A strongly polynomial minimum cost circulation algorithm*, Combinatorica, 5 (1985), pp. 247–255.

[Tarj] R.E. TARJAN, *Data Structures and Network Algorithms*, CBMS – NSF Regional Conference Series 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[W] R.A. WAGNER, *A shortest path algorithm for edge-sparse graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 50–57.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 1037–1047, October 1989

012

A NEW BASE CHANGE ALGORITHM FOR PERMUTATION GROUPS *

CYNTHIA A. BROWN†, LARRY FINKELSTEIN†, AND PAUL W. PURDOM, JR. ‡

Abstract. The computation of a strong generating set for a permutation group acting on a set Ω of n points is the fundamental operation that underlies most of the algorithms in computational group theory. Sims gave a *change of basis* algorithm that transforms a strong generating set relative to one ordering of Ω into a strong generating set relative to a different ordering. Base change is crucial for many of the important algorithms that have been implemented in the Cayley system, and is also important for many applications of computational group theory to combinatorial and search problems. Sims's base change has worst-case time $O(n^5)$. The main result of this paper is a new change of basis algorithm that has worst-case time $O(n^3)$.

Key words. base change, permutation group, algorithm

AMS(MOS) subject classifications. 20, 68

1. Introduction.

Let G be a permutation group acting on a set Ω, and let $\pi = \pi_1, \pi_2, \cdots, \pi_n$ be an ordering of the points of Ω. Define $G^{(i)}$ to be the subgroup of G consisting of all elements of G that fix each of the points $\pi_1, \pi_2, \cdots, \pi_{i-1}$, $1 \leq i \leq n$. The *point stabilizer sequence* for G relative to π is the chain of subgroups

$$G = G^{(1)} \supseteq G^{(2)} \supseteq \cdots \supseteq G^{(n-1)} \supseteq G^{(n)} = \{e\}.$$

A set S of generators for G is said to be a *strong generating set* for G relative to π, if

$$G^{(i)} = \langle S \cap G^{(i)} \rangle,$$

i.e., those *generators* in S that fix $\pi_1, \pi_2, \cdots, \pi_{i-1}$ generate $G^{(i)}$.

The computation of a strong generating set for G relative to an ordering π is the fundamental operation that underlies many important algorithms in computational group theory. Sims [11] developed the first algorithm for constructing a strong generating set, using the *Schreier vector* data structure for storing the resulting generators. Sims also gave an efficient *change of basis* algorithm that transforms a strong generating set relative to one ordering of Ω into a strong generating set relative to a different ordering. Butler and Lam [5] have shown that Sims's base change algorithm has worst-case time $O(n^5)$. This paper presents a change of basis algorithm that has worst-case time $O(n^3)$.

Base change is crucial for an efficient application of Sims's backtracking method for performing fundamental group computations. This method underlies many of the algorithms implemented in the Cayley system [6] and is also important for applications of computational group theory to combinatorial and search problems [3], [5], [10].

---

*Received by the editors December 23, 1987; accepted for publication (in revised form) December 14, 1988. † The work of these authors was supported in part by the National Science Foundation under grant number DCR-8603293.

†College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, Massachusetts 02115.

‡Department of Computer Science, Indiana University, 101 Lindley Hall, Bloomington, Indiana 47405.

Sims's method involves a systematic search through the elements of G in order to compute a subgroup H of G satisfying a certain property (such as centralizing an element of G). If any subgroup K of H is known in advance, or is computed in the course of the search, the algorithm uses K to prune the search space for G. In order to do the pruning efficiently, it is necessary to be able to rapidly compute K_Γ, the pointwise stabilizer in K of an arbitrary subset Γ of Ω [4]. With an efficient change of basis algorithm, strong generators for K relative to an ordering of Ω for which Γ forms a prefix can be computed, and so generators for K_Γ are readily available.

Our base change algorithm is also useful for the pointwise stabilizer problem. Recently, Babai, Luks, and Seress [1] have described an $O(n^4(\log(n)^c))$ algorithm for obtaining a strong generating set for a permutation group. Their method requires an initial ordering of the points to be compatible with an "extended structure tree" determined by the action of the permutation group. Thus the Babai-Luks-Seress algorithm can find the pointwise stabilizer for a set of points that are the leftmost consecutive leaves of the structure tree. As they state, the method can be extended to obtain the pointwise stabilizer of an arbitrary subset of Ω within the same asymptotic worst-case time. However, a simple alternative for the general pointwise stabilizer problem is to use a structure tree to obtain a strong generating set in $O(n^4(\log(n)^c))$ time, and then use our change of basis algorithm to obtain a strong generating set relative to a suitable reordering of the points in time $O(n^3)$.

Our change of basis algorithm uses the *labeled branching* data structure to represent a strong generating set for a permutation group. This data structure was introduced by Jerrum [8] as a compact way of (implicitly) storing a coset table for the point stabilizer chain of subgroups relative to an ordering π. The labeled branching for G requires the storage of at most $n-1$ permutations and can be used to compute a coset representative for $G^{(i+1)}$ in $G^{(i)}$ with a single permutation multiplication. The permutations in the labeled branching for G also constitute a set of strong generators for G relative to π. A labeled branching occupies $O(n^2)$ storage, versus the $O(n^3)$ storage required for the Schreier vector method, and can be computed directly in $O(n^5)$ time using an algorithm described by Jerrum [8]. (See also [7] for a description of an efficient implementation of this algorithm.) Alternatively, a labeled branching for G relative to π can easily be computed in $O(n^3)$ time once a strong generating set for G relative to π is known.

Let $\mathcal{B}$ be a labeled branching for G relative to an ordering $\pi = \pi_1, \pi_2, \cdots, \pi_n$. The key step in our algorithm is a fast method for constructing a labeled branching $\mathcal{B}'$ for G relative to a new ordering $\pi_1, \cdots, \pi_{r-1}, \pi_s, \pi_r, \pi_{r+1}, \cdots, \pi_{s-1}, \pi_{s+1}, \cdots, \pi_n$, where $r < s$. This step, which amounts to a "right cyclic shift" of the points $\pi_r, \cdots, \pi_s$, can be performed in $O(n^2)$ time. The ordering $\pi_1, \pi_2, \cdots, \pi_n$ can be transformed into an arbitrary ordering $\pi'_1, \pi'_2, \cdots, \pi'_n$ by performing at most $n-1$ such shifts, giving a total time of $O(n^3)$. (There are some applications where a fast algorithm for a single cyclic right shift is of direct interest [3].) In Sims's original formulation, the key step is to use an elegant trick to get a strong generating set for G relative to a new ordering of the form $\pi_1, \cdots, \pi_{i-1}, \pi_{i+1}, \pi_i, \cdots, \pi_n$ without performing any "sifting" operations. It then takes $O(n^2)$ transpositions to reach the final ordering. Our method generalizes Sims's trick for transpositions to a right cyclic shift, still without requiring any sifting operations.

2. Labeled branchings. Let permutation group G act on $\Omega = \{1, 2, \cdots, n\}$, and let $\pi = \pi_1, \pi_2, \cdots, \pi_n$ be an ordering of the points of Ω. A *branching* on Ω relative to π is a directed forest in which each edge has the form (π_i, π_j) for $i < j$. A branching

$\mathcal{B}$ is said to be a *labeled branching for* G relative to π if each edge (π_i, π_j) is labeled by a permutation σ_{ij} so that the following properties hold:

(i) $\sigma_{ij} \in G^{(i)}$ and moves π_i to π_j, i.e., $\pi_k^{\sigma_{ij}} = \pi_k$ for $1 \leq k < i$ and $\pi_i^{\sigma_{ij}} = \pi_j$.

(ii) The set of edge labels of $\mathcal{B}$ generates G.

A labeled branching $\mathcal{B}$ is said to be *complete* if the following additional property holds:

(iii) If π_k is in the $G^{(i)}$ orbit of π_i, then there is a directed path in $\mathcal{B}$ from π_i to π_k.

Criterion (iii) ensures that the edge labels of $\mathcal{B}$ form a strong generating set for G relative to the ordering π.

Labeled branchings can be implemented as an array of structures using $O(n^2)$ storage. Although we have described the labeled branching in terms of the edge labels σ, the actual permutations stored in the branching data structure are products of the edge labels. Let π_r be the root of the connected component of $\mathcal{B}$ containing π_j. Associate with each node a *node label* $\tau(j)$, where $\tau(j)$ is the product of the edge labels from the root π_r to π_j if $\pi_r \neq \pi_j$, and is the identity if $\pi_r = \pi_j$. Since $\tau(j)^{-1}$ moves π_j to its root π_r, we may recover the label for edge (π_i, π_j), σ_{ij}, as $\sigma_{ij} = \tau(i)^{-1}\tau(j)$. Thus, the node labels are an implicit way of storing the edge labels, as they allow an edge label to be recovered at the cost of one permutation multiply. Furthermore, if there is a path from π_k to π_j in $\mathcal{B}$, then $\tau(k)^{-1}\tau(j)$ is the product of the edge labels along the path from π_k to π_j. This means that coset representatives can also be recovered from the data structure at the cost of one multiply, as opposed to $O(n)$ multiplies if edge labels were stored.

In addition to the τ field, each node π_j of $\mathcal{B}$ has a *parent* field, where $parent(j) = k$ if (π_k, π_j) is an edge, and -1 (a value less than any point) otherwise. When a constant factor of time is more important than a constant factor of space, one should also store $\tau(j)^{-1}$. For some applications, it is useful to keep a list of the children for each node.

The main operation on a labeled branching is "sifting" a permutation. This is the key step in all algorithms for finding a strong generating set. The basic idea behind sifting is to see if a given permutation g can be written in the form $g = g_{n-1}g_{n-2}\cdots g_1$, where g_i is an element of a fixed set $U^{(i)}$ of coset representatives for $G^{(i+1)}$ in $G^{(i)}$, $1 \leq i \leq n-1$. If g cannot be written in this form, then there exists an index i such that $g' = gg_1^{-1}\cdots g_{i-1}^{-1}$, where $g_j \in U^{(j)}$, for $1 \leq j \leq i-1$, $g' \in G^{(i)}$, and there does not exist a path in $\mathcal{B}$ from i to $j = i^{g'}$. Sift then attempts to create a new edge (i, j) with edge label g'. If $indegree(j) = 0$, then this can be done directly. However, if $indegree(j) = 1$, then care must be taken to preserve both the branching property as well as "connectivity" and "generational" properties already built into the current branching (see [8] and [2, Prop. 8.1, 8.2]).

Jerrum proved that a complete labeled branching for a permutation group can be computed in $O(n^5)$ time using $O(n^2)$ storage. The basic idea of the algorithm is to sift into an empty labeled branching generators for G, and then to sift in "Schreier generators" for $G^{(i+1)}$, $i = 1, \cdots, n-2$, successively. Schreier generators for $G^{(i+1)}$ can be built from a set of generators for $G^{(i)}$ and a set of coset representatives for $G^{(i+1)}$ in $G^{(i)}$ that are available from the branching and were entered in the previous iteration. Knuth [9] also has an $O(n^5)$ time algorithm for computing a strong generating set, but his algorithm requires $O(n^3)$ storage.

3. The base change algorithm. Let $\mathcal{B}$ be a labeled branching for G with respect to the ordering

$$\pi = \pi_1, \cdots, \pi_{r-1}, \pi_r, \cdots, \pi_s, \pi_{s+1}, \cdots, \pi_n,$$

let $\Omega = \{1, 2, \cdots, n\}$, and let

$$\pi' = \pi_1, \cdots, \pi_{r-1}, \pi_s, \pi_r, \pi_{r+1}, \cdots, \pi_{s-1}, \pi_{s+1}, \cdots, \pi_n$$

(i.e., $\pi'_i = \pi_i$, for $1 \leq i < r$, $s < i \leq n$, and $\pi'_r = \pi_s$, $\pi'_i = \pi_{i-1}$, for $r < i \leq s$). Permutation π' is obtained from π by a *cyclic shift of values* (the values between r and s). In this case, the labeled branching $\mathcal{B}'$ with respect to π' is said to be obtained from $\mathcal{B}$ by a *cyclic shift of base points.* We describe a function *Cycle-node*, which transforms $\mathcal{B}$ to $\mathcal{B}'$.

The function Cycle-node has as input the labeled branching $\mathcal{B}$ for G relative to π and a new ordering π' obtained from π by a cyclic shift of values. It returns a labeled branching $\mathcal{B}'$ for G relative to π'. We start with a trivial labeled branching $\mathcal{B}'$ and build $\mathcal{B}'$ into a complete labeled branching for G with respect to π' from the leaves to the root. In the course of building $\mathcal{B}'$, it is convenient to store the edge label connecting π'_j to its parent π'_i (i.e., σ'_{ij}) in the τ'_j field. After all the appropriate edge labels have been constructed, we restore the τ' fields to their original purpose in $O(n^2)$ time. We do this to avoid the necessity of constantly updating the τ fields as new connections are made. Since we are building $\mathcal{B}'$ from leaves to the root, we always introduce new edges of the form (π'_i, π'_j) where π'_j is the current root of a subtree of $\mathcal{B}'$. If we used the standard interpretation of τ, then we would have to update the τ fields of each descendant π'_k of π'_j each time we added an edge, and this takes $O(n^2)$ time. In order to compensate for not using the standard interpretation of τ, we maintain an array *root* of length n with the property that if $i = root[j]$, then π'_i is the root of the subtree containing π'_j.

Let $G_{ab\cdots c}$ be the subgroup of G that fixes the points $a, b, \cdots, c$ of Ω. We denote $G^{(i)} = G_{\pi_1 \cdots \pi_{i-1}}$, the subgroup that fixes the first $i-1$ points, $\pi_1, \cdots, \pi_{i-1}$ and $\Delta^{(i)} = \pi_i^{G^{(i)}}$, the orbit of point π_i in $G^{(i)}$. $G'^{(i)} = G_{\pi'_1 \cdots \pi'_{i-1}}$ and $\Delta'^{(i)} = \pi_i'^{G'^{(i)}}$ are used for the transformed points. We use primes for distinguishing between node fields of $\mathcal{B}$ and $\mathcal{B}'$.

Before performing Cycle-node, we initialize $\mathcal{B}'$ to the empty branching and set $root[i] = i$, $1 \leq i \leq n$. Set the fields of nodes $s+1, \cdots, n$ of $\mathcal{B}'$ to reflect the fact that $\pi_j = \pi'_j$ for j in the range $s+1, \cdots, n$. For each j in the range from $s+1$ to n, if π_i is the parent of π_j and $s+1 \leq i$, we set $parent'(j) = i$, $\tau'(j) = \tau(i)^{-1}\tau(j)$, and $root[j] = root[i]$. If $i < s+1$, then $\tau'(j)$, $parent'(j)$, and $root[j]$ are unmodified from their initial values. After this step is completed, $\mathcal{B}'$ is a labeled branching for $G'^{(s+1)}$ relative to π', subject to adjusting the τ fields.

The next step is the crucial one. We successively fill in the edge labels of $\mathcal{B}'$ that emanate from π'_j as j goes from s to r. This requires computing

$$\Delta'^{(j)} = \pi_j'^{G'^{(j)}}.$$

Now, $\pi'_j = \pi_{j-1}$ for $r < j \leq s$, and $\pi'_r = \pi_s$. Furthermore, for $r < j \leq s$

$$G^{(j)} = G_{\pi'_1 \cdots \pi'_{j-1}} = G_{\pi_1 \cdots \pi_{j-2} \pi_s},$$

and

$$G'^{(r)} = G^{(r)}.$$

Thus, for $r < j \leq s$,

$$\Delta'^{(j)} = \pi_{j-1}^{G_{\pi_1 \cdots \pi_{j-2} \pi_s}}$$

and

$$\Delta^{(r)} = \pi_s^{G^{(r)}}.$$

Since it is straightforward to compute $\Delta'^{(r)}$, we concentrate on the case where $r < j \leq s$. For these values of j, $\Delta'^{(j)} \subseteq \Delta^{(j-1)}$. It is easy to compute $\Delta^{(j-1)}$, so we need a criterion for deciding whether a point of $\Delta^{(j-1)}$ is in $\Delta'^{(j)}$. The following is a generalization of a result of Sims [11].

LEMMA 3.1. *Let $r < j \leq s$ and let $\pi_m = \pi_{j-1}^{\alpha} \in \Delta^{(j-1)}$ for some $\alpha \in G^{(j-1)}$. Then*

$$\pi_m \in \Delta'^{(j)} \iff \pi_s^{\alpha^{-1}} \in \pi_s^{G^{(j)}}.$$

Proof. Suppose that $\pi_m \in \Delta'^{(j)}$. Then $\pi_m = \pi_{j-1}^{\beta}$ for some $\beta \in G'^{(j)} = G_{\pi_1 \cdots \pi_{j-2} \pi_s}$. Now, $\beta\alpha^{-1}$ fixes π_{j-1}, and so $\beta\alpha^{-1} \in G^{(j)}$. Furthermore, β fixes π_s. Thus,

$$\pi_s^{\alpha^{-1}} = \pi_s^{\beta\alpha^{-1}} \in \pi_s^{G^{(j)}}.$$

Conversely, suppose that $\pi_s^{\alpha^{-1}} \in \pi_s^{G^{(j)}}$. Let $\pi_s^{\alpha^{-1}} = \pi_s^{\beta}$ for some $\beta \in G^{(j)}$. Then $\beta\alpha \in G'^{(j)}$ and

$$\pi_m = \pi_{j-1}^{\alpha} = \pi_{j-1}^{\beta\alpha} \in \Delta'^{(j)}$$

as required. □

In preparation for inserting the edge labels of $\mathcal{B}'$ that emanate from π'_j for $r \leq j \leq s$, we construct a Boolean array *orbit* of length n where $orbit[i] = 1$ if and only if $\pi_i \in \pi_s^{G^{(s)}}$. Additionally, we construct an array *cosetrep* of length n where $cosetrep[i]$ is a permutation of $G^{(s)}$ that moves π_s to π_i if $orbit[i] = 1$, and $cosetrep[i]$ is *nil* if $orbit[i] = 0$.

Now assume that we have inserted into $\mathcal{B}'$ the edges that emanate from π'_i, for $r < j < i \leq s+1$, so that $\mathcal{B}'$ is a labeled branching for $G'^{(j+1)}$ relative to π' (subject to adjusting the τ' fields). Further, assume that we have updated *root* and extended *orbit* and *cosetrep* to reflect the fact that *orbit* and *cosetrep* are defined for $\pi_s^{G^{(j+1)}}$. We describe how to insert the edges of $\mathcal{B}'$ that emanate from π'_j.

The first step is to extend *orbit* and *cosetrep* for the orbit $\pi_s^{G^{(j)}}$. This amounts to throwing into the pool of generators for the previous orbit and coset calculations those edge labels of $\mathcal{B}$ that move π_j. Each of these new generators must be applied to each point currently in the orbit, and all generators for $G^{(j)}$ must then be applied against all new points discovered. The total cost of updating *orbit* and *cosetrep* for $r \leq j \leq s$ is thus equal to the cost of computing *orbit* and *cosetrep* for $\pi_s^{G^{(r)}}$ using generators for $G^{(r)}$ available from $\mathcal{B}$.

We now apply Lemma 3.1. Let $\Gamma = \Delta^{(j-1)}$. For each point $\pi_m = \pi_{j-1}^{\alpha} \in \Gamma$, we use *orbit* to find out whether $\pi_s^{\alpha^{-1}} \in \pi_s^{G^{(j)}}$. If it is, we then find q such that $\pi_m = \pi'_q$ and check whether $root[q] = q$. We only want to connect π'_j to a node of $\mathcal{B}'$ that is a root of a subtree. Thus if π'_q is not a root, we move on to the next point of Γ, knowing that the connection will eventually be made between π'_j and π'_q via the root of the subtree containing π'_q. If π'_q is a root, we use *cosetrep* to find β, as in Lemma 3.1, so that for $\gamma = \alpha\beta \in G'^{(j)}$, $\pi_m = \pi_{j-1}^{\gamma}$, and hence $\pi'_q = \pi_j'^{\gamma}$. We now set $\tau'(q) = \gamma$, $parent'(q) = j$. Furthermore, we set $root[q] = j$ and modify *root* so that if $root[i] = q$ then set $root[i] = j$. After examining all points of Γ, $\mathcal{B}'$ will be a labeled branching for $G'^{(j)}$.

We continue decrementing j and constructing $\mathcal{B}'$ in this way until $j = r$. Since $\Delta'^{(r)} = \pi_s^{G^{(r)}}$, for $j = r$ we extend *orbit* and *cosetrep* as before, but this time we do

not require the use of Lemma 3.1. We simply check each point $\pi_m \in \Delta^{(r)}$ to see if its corresponding value π'_q is a root of $\mathcal{B}'$ or not. If it is, then we use *cosetrep* to find an element $\gamma \in G^{(r)} = G'^{(r)}$ that moves π'_r to π'_q. We then install a new edge from π'_r to π'_q and update both $\mathcal{B}'$ and *root* as is done above.

At this point, $\mathcal{B}'$ is a labeled branching for $G'^{(r)}$ relative to π'. Since $\pi'_i = \pi_i$ for $1 \le i \le r-1$, the remaining data for $\mathcal{B}'$ may be filled in using a slight modification of the procedure above. This time we may use the information stored in $\mathcal{B}$ instead of relying on *orbit* and *cosetrep*. In particular, suppose that the data for $\mathcal{B}'$ have been entered for all nodes π'_j such that $i < j$. For each $\pi_m \in \Delta^{(i)} - \{\pi_i\}$, we check if π'_q is a root of $\mathcal{B}'$. If it is, we set $\tau'(q) \leftarrow \tau^{-1}(i)\tau(m)$, $parent'(q) \leftarrow i$, and update *root*. Once this is done, $\mathcal{B}'$ is a labeled branching for $G'^{(i)}$. We then continue decrementing i until $i = 1$.

The final step in the process is to modify all the τ' fields so that $\tau'(i)$ is the product of the edge labels along the path from the root π'_r of the subtree containing π'_i to π'_i. This is easily done, starting from roots of subtrees of $\mathcal{B}$ and working down.

We now give a pseudocode version of Cycle-node. The work of Cycle-node is performed by three main functions, which partition the work along the lines just described. These are Cycle-node-bottom, Cycle-node-middle, and Cycle-node-top. We present pseudocode for each function in sufficient detail to allow for a simple proof of correctness and an analysis of the running time. It is assumed that arrays are always passed by reference.

Cycle-node. *Input Parameters*: $\mathcal{B}$, π, π', r and s, where $\mathcal{B}$ is a labeled branching for G with respect to π, and π' is a new ordering obtained from π by a cyclic shift of values in the range r to s. *Returned Value*: A labeled branching $\mathcal{B}'$ for G with respect to π'. *Notation*: $\pi = \pi_1, \cdots, \pi_n$, and $\pi' = \pi'_1, \cdots, \pi'_n$ where $\pi'_i = \pi_i$, $1 \le i < r$, $s < i \le n$, and $\pi'_r = \pi_s$, $\pi'_i = \pi_{i-1}$, $r < i \le s$.

Set $\mathcal{B}'$ to the identity branch on n points.
Set *root* so that each point is its own root.
Cycle-node-bottom($\mathcal{B}, \mathcal{B}', root, s$) (enter data in $\mathcal{B}'$ for nodes π'_{s+1} to π'_n, and update *root*).
Cycle-node-middle($\mathcal{B}, \mathcal{B}', \pi, \pi', root, r, s$) (enter data in $\mathcal{B}'$ for nodes π'_r to π'_s, and update *root*).
Cycle-node-top($\mathcal{B}, \mathcal{B}', \pi, \pi', root, r$) (enter data in $\mathcal{B}'$ for nodes π'_1 to π'_{r-1}).
(Complete $\mathcal{B}'$ by modifying the τ' fields for nodes π'_1 to π'_n to reflect path products.)
For $j \leftarrow 1$ to n
    set $i \leftarrow parent'(j)$
    if $i \neq -1$ then set $\tau'(j) \leftarrow \tau'(i)\tau'(j)$.
Return($\mathcal{B}'$).

Cycle-node-bottom. *Input Parameters*: $\mathcal{B}, \mathcal{B}', root, s$. *Purpose*: Modify $\mathcal{B}'$ by copying the edge labels from $\mathcal{B}$ for nodes $\pi'_{s+1}, \cdots, \pi'_n$.

For $j \leftarrow s+1$ to n
    let $i = parent(j)$
    if $(s < i)$ then
        set $parent'(j) \leftarrow i$, $root[j] \leftarrow root[i]$ and $\tau'(j) \leftarrow \tau^{-1}(i)\tau(j)$
            (in this case, $\tau'(j)$ is an edge label from π'_i to π'_j).

Cycle-node-middle. *Input Parameters*: $\mathcal{B}, \mathcal{B}', \pi, \pi', root, r, s$. *Purpose*: Build $\mathcal{B}'$ for nodes π'_r to π'_s by implementing Lemma 3.1. *Local Variables*: *orbit* and *cosetrep* are used to define $\pi_s^{G^{(j)}}$, $r \le j \le s$ in the sense that *orbit* is an n-dimensional Boolean

array with $orbit[i] = 1$ if and only if $\pi_i \in \pi_s^{G^{(j)}}$ and *cosetrep* is an n-dimensional array with $cosetrep[i]$, an element of $G^{(j)}$ that moves π_s to π_i if $orbit[i] = 1$; *genlist* is a list of edge labels of $\mathcal{B}$ which generate $G^{(j)}$ and *orbitlist* is a list of points in $\pi_s^{G^{(j)}}$.

Initialize *orbit*, *cosetrep*, *orbitlist*, and *genlist* for $\pi_s^{G^{(s)}}$ using $\mathcal{B}$.
For $j \leftarrow s$ down to $r+1$
    set $\Delta \leftarrow \Delta^{(j-1)} = \pi_{j-1}^{G^{(j-1)}}$ (this can be computed using $\mathcal{B}$)
    for each $\pi_m \in \Delta$
        let $\pi_p = \pi_s^{\tau^{-1}(m)\tau(j-1)}$ ($\tau^{-1}(j-1)\tau(m)$ is the path product from π_{j-1} to π_m)
        if $orbit[p] = 1$ ($\pi_p \in \Delta^{(j-1)}$), then
            let $\pi'_q = \pi_m$
            if $q \neq j$ and π'_q is a root of $\mathcal{B}'$, then (connect π'_j to π'_q in $\mathcal{B}'$)
                set $\tau'(q) \leftarrow cosetrep[p]\tau^{-1}(j-1)\tau(m)$
                    (in Lemma 3.1, $\beta = cosetrep[p]$ and $\alpha = \tau^{-1}(j-1)\tau(m)$)
                for $i \leftarrow j+1$ to n (update *root*)
                    if $root[i] = q$, then set $root[i] \leftarrow j$
    (Extend *orbit* and *cosetrep* to $\pi_s^{G^{(j-1)}}$.)
    if π_{j-2} is not a leaf (otherwise, the data for the orbit is unchanged), then
        (*orbitlist*, *genlist*) $\leftarrow$
            update-orbit($\mathcal{B}, \pi$, *orbit*, *cosetrep*, *genlist*, *orbitlist*, $s, j-2$).
(Now *orbit* and *cosetrep* are defined for $\pi_s^{G^{(r)}}$. Enter the data for node π'_r.)
Set $\Delta \leftarrow \pi_s^{G^{(r)}}$.
For each $\pi_m \in \Delta$
    let $\pi'_q = \pi_m$
    if $root[q] = q$ and $q \neq r$ (π'_q is a root of $\mathcal{B}'$), then
        (Connect π'_r to π'_q.)
        set $\tau'(q) \leftarrow cosetrep[m]$ ($cosetrep[m] \in G^{(r)}$ and moves $\pi_s = \pi'_r$ to $\pi_m = \pi'_q$)
        for $i \leftarrow r+1$ to n (update *root*)
            if $root[i] = q$, then set $root[i] \leftarrow r$.

Cycle-node-top. *Input Parameters*: $\mathcal{B}, \mathcal{B}', \pi, \pi', root, r$. *Purpose*: Modify $\mathcal{B}'$ by inserting the data for nodes π'_1 to π'_{r-1}.

For $i \leftarrow r-1$ to 1 do
    set $\Delta \leftarrow \Delta^{(i)} = \pi_i^{G^{(i)}}$
    for each $\pi_m \in \Delta$
        let $\pi'_q = \pi_m$
        if $root[q] = q$, then
            set $\tau'(q) \leftarrow \tau^{-1}(i)\tau(m)$ and $parent'(q) \leftarrow i$
            for $k \leftarrow i+1$ to n (update *root*)
                if $root[k] = q$, then set $root[k] \leftarrow i$.

We complete this part of the discussion by presenting pseudocode for the procedure update-orbit in Cycle-node-middle.

Update-orbit. *Input Parameters*: $\mathcal{B}, \pi$, *orbit*, *cosetrep*, *genlist*, *orbitlist*, s, j. *Return value*: The list (*orbitlist*, *genlist*). *Purpose*: Extend *orbit*, *cosetrep*, *genlist*, *orbitlist* from $\pi_s^{G^{(j+1)}}$ to $\pi_s^{G^{(j)}}$. *Local variables*: A list *newpoints* for holding new points in the orbit; a temporary list of points, *pointlist*; and a list *newgens* for holding the edge labels of $\mathcal{B}$ in $G^{(j)}$ that move π_j.

Set *newpoints* $\leftarrow$ *nil* and *newgens* $\leftarrow$ the edge labels of $\mathcal{B}$ in $G^{(j)}$ that move π_j.
For each $g \in$ *newgens* (apply each new generator against each old point)

for each $p \in$ *orbitlist*
let $\pi_q = \pi_p^g$
if *orbit*$[q] \neq 1$ (a new point in $\pi_s^{G^{(j)}}$ has been found), then
set *orbit*$[q] \leftarrow 1$, *cosetrep*$[q] \leftarrow$ *cosetrep*$[p] * g$, and append q to *newpoints*.
(Append *newgens* to *genlist* and apply each generator in *genlist* to each point in *newpoints*.)
While *newpoints* $\neq$ *nil*
append *newpoints* to *orbitlist*, set *pointlist* $\leftarrow$ *newpoints* and *newpoints* $\leftarrow$ *nil*
for each $g \in$ *genlist* (apply each generator against each old point)
for each $p \in$ *pointlist*
let $\pi_q = \pi_p^g$
if *orbit*$[q] \neq 1$ (a new point in $\pi_s^{G^{(j)}}$ has been found), then
set *orbit*$[q] \leftarrow 1$, *cosetrep*$[q] \leftarrow$ *cosetrep*$[p] * g$, and append q to *newpoints*.
Return(*orbitlist*, *genlist*).

4. Proof of correctness of the base change algorithm. In this section we give a straightforward proof that base-change performs correctly, namely, that it returns a labeled branching for G relative to the new ordering π'. This will follow directly once we have established the same result for the procedure Cycle-node.

PROPOSITION 4.1. *Let $\mathcal{B}$ be a labeled branching for G relative to the ordering π and let π' be a new ordering obtained from π by a cyclic shift of values. Then Cycle-node($\mathcal{B}, \pi, \pi'$) returns a labeled branching for G relative to π'.*

Proof. Let $\pi = \pi_1, \cdots, \pi_n$, and $\pi' = \pi'_1, \cdots, \pi'_n$ where $\pi'_i = \pi_i$, $1 \leq i < r, s < i \leq n$, and $\pi'_r = \pi_s$, $\pi'_i = \pi_{i-1}$, $r < i \leq s$. In addition, let $G'^{(j)} = G_{\pi'_1 \cdots \pi'_{j-1}}$ for $1 \leq j \leq n$.

Consider the state of the labeled branching $\mathcal{B}'$ created by Cycle-node just after Cycle-node-top returns. We will prove by induction on i for $i = n$ down to 1 that if

$$\pi'_j \in \Delta'^{(i)} = \pi_i'^{G'^{(i)}},$$

then there exists a path in $\mathcal{B}'$ from π'_i to π'_j, and that the product of the edge labels from π'_i to π'_j is an element of $G'^{(i)}$ that moves π'_i to π'_j. Once this has been shown, it follows directly that the last step of Cycle-node creates a labeled branching for G relative to π' in which the τ fields have the correct values.

The result is clearly true when $i = n$, so assume that $i < n$ and the result holds for all k such that $i < k \leq n$. The entries for node π'_i are set in one of the functions Cycle-node-bottom, Cycle-node-middle, or Cycle-node-top. In all cases, we examine a complete set of points that contains all possible candidates for $\pi'_j \in \Delta'^{(i)}$. This follows from Lemma 3.1 in the case where $r < j \leq s$ and directly in the remaining cases. If π'_j is such a candidate, then we first check if π'_j is currently a root of $\mathcal{B}'$. If so, then we connect π'_i to π'_j and label the edge by a permutation in $G'^{(i)}$ that moves π'_i to π'_j. Otherwise, we do nothing. To see why this works, assume that $\ell = root[j] < j$, so that there is a path from π'_ℓ to π'_j in $\mathcal{B}'$. Assume first that $i = \ell$. This could happen if π'_j is not a root of $\mathcal{B}'$ at the end of the previous iteration, but its root π'_m is connected to π'_i, and hence *root* updated before π'_j is encountered. But then, by induction, there is a path product from π'_i to π'_j through π'_m and the composition of the path products from π'_i to π'_m and π'_m to π'_j moves π'_i to π'_j and is an element of $G'^{(i)}$. Otherwise, $i < \ell$, and we may invoke our inductive assumption to conclude that the path product σ from π'_ℓ to π'_j moves π'_ℓ to π'_j and lies in $G'^{(\ell)}$. But since $\pi'_j \in \Delta'^{(i)}$ and $\sigma \in G'^{(i)}$,

it then follows that $\pi'_\ell \in \Delta'^{(i)}$ as well. Hence, we are certain of choosing π'_ℓ as one of our candidates. When this occurs, we connect π'_i to π'_ℓ with edge label $\rho \in G'^{(i)}$ that moves π'_i to π'_ℓ. But then $\rho\sigma \in G'^{(i)}$ and moves π'_i to π'_j. Therefore, the result holds for i and hence for all nodes by induction. □

5. Analysis of the base change algorithm. The main result of this section is an analysis of the running time for Cycle-node.

PROPOSITION 5.1. *Cycle-node has running time* $O(n^2)$.

Proof. We will first show that each of Cycle-node-bottom, Cycle-node-middle, and Cycle-node-top have running time $O(n^2)$.

The running time for Cycle-node-bottom is dominated by at most $n - s$ permutation multiplies of the form $\tau^{-1}(i)\tau(j)$. Thus Cycle-node-bottom takes time $O(n^2)$.

The analysis of Cycle-node-middle is a little more complex. Let us first compute the total cost for adding new edges and updating *root*. Each time we add a new edge, it costs $O(n)$ time to enter the data and $O(n)$ time to update *root*. However, at most $O(n)$ edges can ever be entered since a labeled branching on n nodes has at most $n-1$ edges. Thus the total cost incurred for adding new edges and updating *root* is $O(n^2)$.

Consider the cost of updating *cosetrep* and *orbit*. These data structures are initialized for $\Delta^{(s)}$ using the edge labels of $\mathcal{B}$ that lie in $G^{(s)}$. Each time we call update-orbit as j goes from s down to r, we extend *orbit* and *cosetrep* to $\pi_s^{G^{(j-1)}}$ by adding to the generators for $G^{(j)}$ those edge labels that move π_{j-1}. Each generator is applied exactly once to each point in the orbit. Furthermore, the union of all generators added incrementally yields a set of edge labels that generate $G^{(r)}$, and hence has cardinality at most $n - 1$. Thus, the cumulative cost of all the calls to update-orbit is equal to the cost of building *orbit* and *cosetrep* for $\pi_s^{G^{(r)}}$, and hence is $O(n^2)$. Doing the orbit calculation in time $O(n^2)$ is perhaps the most delicate part of the algorithm. We considered several alternate ways of doing this part of the calculation, but the alternate ways always resulted in larger running times.

We now incorporate these two observations into the analysis of Cycle-node-middle. Consider the first "for" loop on j for j goes from s down to $r + 1$. The first step is to compute $\Delta = \Delta^{(j-1)}$ from $\mathcal{B}$. This step takes $O(n)$ time since it simply involves determining all points in $\mathcal{B}$ that can be reached from π_j. For each point $\pi_m \in \Delta$, we compute the image

$$\pi_p = \pi_s^{\tau^{-1}(m)\tau(j)}$$

and check if

$$\pi_p \in \pi_s^{G^{(j)}}.$$

This takes constant time. If $\pi_p \notin \pi_s^{G^{(j)}}$, then we continue. Otherwise, we enter the data for node π'_q where $\pi'_q = \pi_m$. This takes a constant amount of time plus the time to compute $\tau'(q)$ and update *root*. Thus the cost of each iteration of the for loop on Δ is $O(1)$ plus the cost of computing new edge labels and updating *root* each time a new edge label is added. We can then summarize the total work on the for loop on j as $O((s - r)n)$ + (the cost of adding new edge labels and updating *root*) + (the cost of updating *orbit* and *cosetrep*) = $O(n^2)$. Finally, consider the last for loop on Δ used to fill in the edges of $\mathcal{B}'$ which emanate from π'_r. The orbit Δ has already been computed through the process of updating *orbit* and *cosetrep*. As before, the cost of executing this loop is $O(1)$ per iteration plus the cost of adding new edge labels and updating *root*. Combining this with the previous computation results in a running time Cycle-node-middle of $O(n^2)$.

Using the same reasoning, the running time for Cycle-node-top can be computed to be $O(n)$+(the cost of adding new edge labels and updating*root*) $= O(n^2)$. Finally, the cost of completing $\mathcal{B}'$ is dominated by $O(n)$ permutation multiplies and hence is $O(n^2)$. Since Cycle-node-bottom, Cycle-node-middle, Cycle-node-top, and the final completion of $\mathcal{B}'$ each have running times of $O(n^2)$, the same holds for Cycle-node and the proof is completed. □

Since at most $O(n)$ calls to Cycle-node are made by our base change algorithm, we have the following result as a corollary.

COROLLARY 5.2. *The base change algorithm runs in time* $O(n^3)$.

6. Implementation issues. The pseudocode presented in §3 for Cycle-node was designed to give a clean exposition of the algorithm and to facilitate the analysis and proof of correctness. In this section, we indicate how a more complex version of Cycle-node can avoid much of the duplication inherent in the earlier version. This more complex version is the one we actually programmed.

The main difference in the new version of Cycle-node occurs after we have built the labeled branching $\mathcal{B}'$ for nodes π'_r, π'_{r+1}, $\cdots$, π'_n. The idea is to integrate the information stored in $\mathcal{B}'$ back into $\mathcal{B}$ and return the labeled branching for G relative to π' in $\mathcal{B}$. There are two reasons for doing this. First, $\tau'(i) = \tau(i)$ and $parent(i) = parent'(i)$ for $1 \leq i \leq r-1$, so the top of the branching is unchanged. Second, as r increases, most of the edge labels in the new branching will emanate from nodes π'_1, $\cdots$, π'_{r-1}; i.e., most of the nodes of $\mathcal{B}'$ at the end of Cycle-node-middle will be roots. When the edges emanating from nodes with index less than r terminate in a root π'_q, with $r \leq q$, then $\tau'(q) = \tau(m)$ where $\pi_m = \pi'_q$. Therefore $\tau'(q)$ has already been computed in $\mathcal{B}$.

We briefly describe how we weave into $\mathcal{B}$ the knowledge gained from the structure of $\mathcal{B}'$ in order to produce the new branching. Consider the state of $\mathcal{B}'$ at the time just after the edge labels emanating from π'_r are inserted, and let $\mathcal{R}$ be a list of the roots of $\mathcal{B}'$, excluding nodes π'_i, $i < r$. We use a routine insert-roots to connect edges from nodes $\pi_i = \pi'_i$, $i < r$, to nodes π'_j, $j \geq r$, where π'_j is a root of $\mathcal{B}'$.

For each $\pi'_j \in \mathcal{R}$, let $\pi_m = \pi'_j$ and follow π_m backwards in $\mathcal{B}$ until we come to the first node π_i, if any, such that $i < r$. If such a node is not found, then π'_j is a root in the new branching and we set $\tau(j) \leftarrow identity$ and $parent(j) \leftarrow -1$. Otherwise, we connect π_i to π'_j. Note that $\tau^{-1}(i)\tau(m)$ is the path product from $\pi_i = \pi'_i$ to $\pi_m = \pi'_j$ and so moves π'_i to π'_j. Thus, $\tau(j) \leftarrow \tau(i)(\tau^{-1}(i)\tau(m)) = \tau(m)$. By manipulation of pointers, the proper τ value can be transferred from node m to j in a single assignment, avoiding unnecessary copying. When insert-root returns, the node fields of $\mathcal{B}$ are complete for all nodes π'_i such that either π'_i is a root or $parent'(i) < r$. The fields for nodes that are children of roots in $\mathcal{B}'$ can now be entered in $\mathcal{B}$ in a straightforward manner.

An extra measure of efficiency can be obtained by recognizing situations where the correct information is already stored in $\mathcal{B}$ and need not be copied from $\mathcal{B}'$. If, at the end of Cycle-node-middle, a node $\pi_j > s$ is a root in $\mathcal{B}'$, then the τ fields for itself and its descendants are the same in the new branching as in the old, and so no modifications need be considered for them. If π_j is not a root in $\mathcal{B}$, its parent in the new branching is unchanged from that in $\mathcal{B}$. If there is a component of $\mathcal{B}$ whose root π_j is greater than s, that component will never be affected by the base change algorithm at all. If $\mathcal{B}'$ is used only to update those parts of $\mathcal{B}$ that are changed, additional savings result. These modifications do not affect the worst-case performance of the algorithm, but they can greatly increase its speed in many practical situations where

the base change affects only a few points of Ω.

REFERENCES

[1] L. BABAI, E. M. LUKS, AND A. SERESS, *Fast management of permutation groups*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Long Beach, CA, 1988, pp. 272–282.

[2] C. A. BROWN, L. FINKELSTEIN, AND P. W. PURDOM, *An efficient implementation of Jerrum's algorithm*, Tech. Report NU-CCS-87-19, Northeastern University, Boston, MA, 1987.

[3] ———, *Backtrack searching in the presence of symmetry*, in Proc. AAECC-6, Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, to appear.

[4] G. BUTLER AND J. J. CANNON, *Computing in permutation and matrix groups* I: *Normal closure, commutator subgroups, series*, Math. Comp., 39 (1982), pp. 663–670.

[5] G. BUTLER AND C. W. H. LAM, *A general backtrack algorithm for the isomorphism problem of combinatorial objects*, J. Symbolic Comput., 1 (1985), pp. 363–382.

[6] J. J. CANNON, *An introduction to the group theory language, Cayley*, in Computational Group Theory, M. D. Atkinson, ed., Academic Press, New York, 1984, pp. 145–184.

[7] G. D. COOPERMAN, L. A. FINKELSTEIN, AND P. W. PURDOM, *Fast group membership using a strong generating test for permutation groups*, in Computers and Mathematics, E. Kaltofen and S. M. Watt, eds., Springer-Verlag, Berlin, New York, 1989, pp. 27–36.

[8] M. JERRUM, *A compact representation for permutation groups*, J. Algorithms, 7 (1986), pp. 60–78.

[9] D. E. KNUTH, *Notes on efficient representation of perm groups*, unpublished notes, 1980.

[10] J. LEON, *Computing automorphism groups of combinatorial objects*, in Computational Group Theory, M. D. Atkinson, ed., Academic Press, New York, 1984, pp. 321–337.

[11] C. C. SIMS, *Computation with permutation groups*, in Proc. Second Symposium on Symbolic and Algebraic Manipulation, S. R. Petrick, ed., Association for Computing Machinery, New York, 1971, pp. 23–28.

SIAM J. COMPUT.
Vol. 18, No. 5, pp. 1048–1055, October 1989

013

RATIO ESTIMATORS ARE MAXIMUM-LIKELIHOOD ESTIMATORS FOR NON-CONTEXT-FREE GRAMMARS*

KEITH HUMENIK†

Abstract. This paper shows that straightforward ratio estimators for the production probabilities of non-context-free unambiguous probabilistic grammars are maximum-likelihood estimators. These ratio estimates are obtained by analyzing the derivations of the strings in a random sample of strings taken from the language derived by the grammar.

Key words. probabilistic grammars, ratio estimators, maximum-likelihood estimators, random sample, unambiguous grammar

AMS(MOS) subject classifications. 60J99, 62F10, 68Q50, 68Q75

1. Introduction. The structure of most programming languages can be described by context-free grammars. However, many other interesting questions regarding the actual usage of programming languages can only be answered by adding the notion of probability to a language. One method of accomplishing this task is to define a probabilistic grammar by assigning probabilities to each production in the grammar, G. Once this is done, the probability of a word in $L(G)$ can be calculated [3].

Often the grammar, G, and the language, $L(G)$, are known, but the production probabilities are not. Humenik and Pinkham [4],[5] have shown that these production probabilities can be estimated for context-free grammars, by taking a random sample of strings from $L(G)$ and parsing the strings to obtain the relative frequency with which each production in G is used. The probabilities are then estimated by using simple ratio estimators. A random sample of 40 or more strings yields excellent results. Maryanski and Booth [6] have shown that these ratio estimators are the maximum-likelihood estimates for the production probabilities in G. Chaudhuri and Rao [3] have shown that the true probabilities may be obtained as the limit of the estimates inferred from an increasing sequence of randomly drawn samples from $L(G)$.

There are many structures, however, that cannot be adequately described by context-free grammars. In this paper we review the notion of probabilistic grammars that are not context-free and prove that ratio estimators remain maximum-likelihood estimates of the production probabilities of an unambiguous non-context-free grammar. This result is not surprising, since we would expect the frequency of production use to provide much of the necessary information to estimate the production probabilities.

2. Preliminaries. A *probabilistic grammar* (*p-grammar*) is a 5-tuple, $G=(V_T, V_N, R, P, \alpha_1)$, where V_T is a finite set of *terminals*; V_N is a finite set of *nonterminals*; R is a mapping $(V_N \cup V_T)^* V_N (V_N \cup V_T)^* \times (V_N \cup V_T)^*$, where each element of R is of the form $\alpha_i \rightarrow \beta_{ij}$, $i=1,\cdots,m$, $j=1,\cdots,m_i$, and is called a *production* or *rule*;

*Received by the editors October 26, 1987; accepted for publication (in revised form) January 3, 1989.

†Department of Computer Science, University of Maryland– Baltimore County, Baltimore, Maryland 21228.

$\alpha_1 \in V_N$ is the *start symbol*; and P is a set of *production probabilities* such that the probability associated with $\alpha_i \to \beta_{ij}$ is p_{ij}. The set R and, hence, the set P are finite. α_i is called the *premise* of the production $\alpha_i \to \beta_{ij}$; β_{ij} is called the *consequence*.

A probabilistic grammar in which all productions in R satisfy $|\alpha_i| \leq |\beta_{ij}|$ is called a *context-sensitive p-grammar* (CSPG). A p-grammar that is not context-sensitive is called an *unrestricted p-grammar* (UPG).

In this paper we shall assume that all p-grammars are unambiguous, using the definition of Aho and Ullman [1]. That is, let G be an unrestricted p-grammar. Let D be the set of all derivations of the form $\alpha_1 \overset{+}{\Rightarrow} w$. That is, elements of D are sequences of the form $(\beta_1, \beta_2, \cdots, \beta_n)$, such that $\beta_1 = \alpha_1$, $\beta_n \in V_T^*$, and $\beta_{i-1} \Rightarrow \beta_i$, $1 \leq i \leq n$.

Define a relation R_0 on D by $(\beta_1, \beta_2, \cdots, \beta_n) R_0 (\gamma_1, \gamma_2, \cdots, \gamma_n)$ if and only if there exists some i between 1 and n such that

(1) $\beta_j = \gamma_j$ for all $1 \leq j \leq n$ such that $j \neq i$.

(2) We can write $\beta_{i-1} = \delta_1\delta_2\delta_3\delta_4\delta_5$ and $\beta_{i+1} = \delta_1\epsilon\delta_3\eta\delta_5$ such that $\delta_2 \to \epsilon$ and $\delta_4 \to \eta$ are in R; and either $\beta_i = \delta_1\epsilon\delta_3\delta_4\delta_5$ and $\gamma_i = \delta_1\delta_2\delta_3\eta\delta_5$, or conversely.

Let R be the smallest equivalence relation containing R_0. Each equivalence class of R represents the essentially similar derivations of a given sentence.

A p-grammar is *unambiguous* if each w in $L(G)$ appears as the last component of a derivation in one and only one equivalence class under R.

Example 1.

- Let G_1 be given by the following

$$S \to abC \mid aB$$
$$B \to bc$$
$$bC \to bc$$

where the four productions have probabilities p_{11}, p_{12}, p_{21}, and p_{31}, respectively. G_1 is an ambiguous grammar, since the sequences (S, abC, abc) and (S, aB, abc) are in two distinct equivalence classes.

- Let G_2 be given by

$$S \to aSBC \mid aBC$$

with probabilities p_{11} and p_{12}, respectively, and

$$p_{21} : aB \to ab$$
$$p_{31} : bB \to bb$$
$$p_{41} : CB \to BC$$
$$p_{51} : bC \to bc$$
$$p_{61} : cC \to c$$

Then G_2 is an unambiguous grammar.

We shall assume herein that the probabilities associated with the productions of R depend only on the deterministic properties of the grammar, G, and not upon the sequence in which the productions are applied during the generation of a given string of the language [2]. That is, the use of a production to expand a sentential form β_i depends only on β_i itself and not upon how β_i was derived in G. We shall call a grammar that has this property a *Markov grammar*.

We shall assume that all grammars are *proper*, that is

$$\sum_{j=1}^{m_i} p_{ij} = 1$$

for all $i=1,\cdots,m$.

The probability of $w \in L(G)$ is given by the product of the production probabilities associated with the productions used in the derivation of w. That is,

$$P(w) = \prod_{k=1}^{K} p_k,$$

where K is the number of steps in the derivation of w and $p_k \in P$ is the probability of the production used at the kth step. Note that since G is unambiguous, there is only one essentially distinct derivation of w, and this derivation can always be found, since there are a finite number of productions in R.

A grammar that satisfies $\sum_{w\in L(G)} P(w) = 1$ is called a *consistent grammar*. Not every grammar is consistent.

A *random sample*, RS, of size M from $L(G)$ is a multiset of M strings from $L(G)$, obtained by drawing a string from $L(G)$ such that each string is equally likely to be drawn. Thus a string is drawn randomly, its derivation is determined, and it is replaced [3].

Example 2. For the grammar G_2 in Example 1, the derivation of the string $a^ib^ic^i$ is given by

$$S \overset{i-1}{\Rightarrow} a^{i-1}S(BC)^{i-1} \Rightarrow a^i(BC)^i \overset{(i-1)i/2}{\Rightarrow} a^iB^iC^i$$

$$\Rightarrow a^ibB^{i-1}C^i \overset{i-1}{\Rightarrow} a^ib^iC^i \Rightarrow a^ib^icC^{i-1} \overset{i-1}{\Rightarrow} a^ib^ic^i.$$

Then $P(a^ib^ic^i)= p_{11}^{i-1}p_{12}p_{41}^{(i-1)i/2}p_{21}p_{31}^{i-1}p_{51}p_{61}^{i-1}$. But if G_2 is proper, then $p_{j1} = 1$ for $2 \le j \le 6$, and $P(a^ib^ic^i)= p_{11}^{i-1}p_{12}$. Note that

$$\sum_{i=1}^{\infty} P(a^ib^ic^i) = \sum_{i=1}^{\infty} p_{11}^{i-1}p_{12} = p_{12}\frac{1}{1-p_{11}} = 1,$$

so G_2 is consistent.

3. Main result. Consider a probabilistic, unambiguous grammar, $G = (V_T, V_N, R, P, \alpha_1)$, and assume that we know everything except the values for the rule probabilities P. Using the information in the sample RS, these values are to be estimated. The sets R and P of G look like the following:

$$\begin{array}{lcccccc} & & p_{11} & & p_{12} & \cdots & p_{1m_1} \\ \alpha_1 & \rightarrow & \beta_{11} & | & \beta_{12} & | \ \cdots \ | & \beta_{1m_1} \end{array}$$

$$\begin{array}{lcccccc} & & p_{21} & & p_{22} & \cdots & p_{2m_2} \\ \alpha_2 & \rightarrow & \beta_{21} & | & \beta_{22} & | \ \cdots \ | & \beta_{2m_2} \end{array}$$

$$\begin{matrix} & p_{m1} & p_{m2} & \cdots & p_{mm_m} \\ \alpha_m \rightarrow & \beta_{m1} & \beta_{m2} & \cdots & \beta_{mm_m} \end{matrix}$$

Partition R and P so that

$$R = R_1 \cup R_2 \cup \cdots \cup R_m$$

and

$$P = P_1 \cup P_2 \cup \cdots \cup P_m$$

such that R_i and P_i are the rules and their associated probabilities that have α_i as premise.

According to G, let n_{ij} be the number of times that rule r_{ij} with premise α_i and probability p_{ij} is used in the parsing of the M strings of RS. This rule is $p_{ij} : \alpha_i \rightarrow \beta_{ij}$.

Also let $N(\alpha_i)$ be the total number of times that an α_i production is used in the parsing of the M strings of RS according to G.

THEOREM 1. *The maximum-likelihood estimate $\hat{p}_{ij}$ of the rule probability $p_{ij} \in P$ (the probability of the production $\alpha_i \rightarrow \beta_{ij}$) given RS is*

$$\hat{p}_{ij} = \frac{n_{ij}}{N(\alpha_i)}.$$

Proof. Let $x_k \in L(G)$. Then, if G is unambiguous,

$$p(x_k) = \prod_{i=1}^{m} \prod_{j=1}^{m_i} p_{ij}^{c_{ijk}},$$

where $p_{ij} \in P$ and c_{ijk} is the number of times that production $\alpha_i \rightarrow \beta_{ij}$ is used in the derivation of x_k.

Note. We are assuming that $p(x_k)$ is equal to the product of the probabilities of the individual productions used in the derivation of x_k.

Since the strings making up RS are statistically independent,

$$p(RS) = \prod_{k=1}^{u} p(x_k)^{f_k} = \prod_{k=1}^{u} \left[\prod_{i=1}^{m} \prod_{j=1}^{m_i} p_{ij}^{c_{ijk}} \right]^{f_k},$$

where f_k is the number of times that string x_k, $k = 1, 2, \cdots, u$ appears in the sample.

Taking the log of both sides and collecting terms yields

$$\log(p(RS)) = \sum_{k=1}^{u} f_k \left[\sum_{i=1}^{m} \sum_{j=1}^{m_i} c_{ijk} \log(p_{ij}) \right] = \sum_{k=1}^{u} \left[\sum_{i=1}^{m} \sum_{j=1}^{m_i} f_k c_{ijk} \log(p_{ij}) \right].$$

But

$$n_{ij} = \sum_{k=1}^{u} f_k c_{ijk},$$

and thus

$$\log(p(RS)) = \sum_{i=1}^{m} \sum_{j=1}^{m_i} n_{ij} \log(p_{ij}).$$

The maximum-likelihood estimate of the probabilities is obtained when

$$\frac{\partial \log p(RS)}{\partial p_{ij}} = 0, \ i = 1, \cdots, m; \ j = 1, \cdots, m_i.$$

However, let

$$P_i = \{p_{i1}, p_{i2}, \cdots, p_{im_i}\}$$

be one of the subsets of P. Since

$$\sum_{\nu=1}^{m_i} p_{i\nu} = 1,$$

only $m_i - 1$ of the elements of P_i can be specified independently. Therefore, for $\nu = 1, 2, \cdots, m_i - 1$,

$$\frac{\partial \log p(RS)}{\partial p_{i\nu}} = \frac{\partial}{\partial p_{i\nu}} \left[\sum_{i=1}^{m} \sum_{j=1}^{m_i} n_{ij} \log(p_{ij}) \right] = \frac{n_{i\nu}}{p_{i\nu}} + \frac{n_{im_i}}{p_{im_i}} \frac{\partial p_{im_i}}{\partial p_{i\nu}}.$$

But

$$\frac{\partial p_{im_i}}{\partial p_{i\nu}} = -1,$$

so

$$\frac{n_{i\nu}}{p_{i\nu}} = \frac{n_{im_i}}{p_{im_i}} = g, \tag{1}$$

where g is a constant and $\nu = 1, 2, \cdots, m_i - 1$. By the law of equal proportions,

$$g = \frac{\sum_{\nu=1}^{m_i} n_{i\nu}}{\sum_{\nu=1}^{m_i} p_{i\nu}} = \sum_{\nu=1}^{m_i} n_{i\nu} = N(\alpha_i).$$

But $N(\alpha_i)$ is the total number of α_i productions used in the generation of the M strings of RS. Using (1) gives

$$\hat{p}_{i\nu} = \frac{n_{i\nu}}{N(\alpha_i)},$$

which yields the general form

$$\hat{p}_{ij} = \frac{n_{ij}}{N(\alpha_i)}.$$

Conclusion. Ratio estimators are maximum-likelihood estimators for unambiguous grammars.

4. Examples and numerical results. This section provides two examples that illustrate the application of the theorem in §3. Random samples are generated by using a Pascal program to simulate the derivation of strings in the specific grammar. The program uses a standard random number generator to produce random values, which are then used to select the appropriate production. Of course, for some strings within the derivation only one production is applicable and is applied immediately.

TABLE 1
G_2 *with* M=40 *and* M=100.

Sample	Actual values		Estimated values M=40		Estimated values M=100	
	p_{11}	p_{12}	$\hat{p}_{11}$	$\hat{p}_{12}$	$\hat{p}_{11}$	$\hat{p}_{12}$
1	0.01	0.99	0.0000	1.0000	0.0099	0.9901
2	0.10	0.90	0.0909	0.9091	0.0991	0.9009
3	0.25	0.75	0.2857	0.7143	0.2424	0.7576
4	0.50	0.50	0.4805	0.5195	0.4624	0.5376
5	0.75	0.25	0.7701	0.2299	0.7590	0.2410
6	0.90	0.10	0.8730	0.1270	0.9063	0.0937
7	0.99	0.01	0.9895	0.0105	0.9902	0.0098

Example 3. Random samples consisting of 40 strings and 100 strings were generated using the grammar G_2 given in Example 1. Table 1 shows the actual values of p_{11} and p_{12} and the estimated values of p_{11} and p_{12}, $\hat{p}_{11}$ and $\hat{p}_{12}$, respectively. Since G_2 is proper, all other probabilities are equal to one and, hence, need not be estimated. The estimates were calculated using maximum-likelihood ratio estimation. Thus, $\hat{p}_{11}$ was calculated by dividing the total number of times that rule r_{11} was used in derivations of the M strings in RS by the total number of times that both rules in R_1 (i.e., r_{11} and r_{12}) were used to derive all strings in RS. That is,

$$\hat{p}_{11} = \frac{n_{11}}{n_{11} + n_{12}} = \frac{n_{11}}{n_{11} + M},$$

since $n_{12} = 1$ for each string derived in G_2. Similarly,

$$\hat{p}_{12} = \frac{M}{n_{11} + M} = 1 - \hat{p}_{11}.$$

For example, random sample 2 with M=40 contained 36 occurrences of the string abc and 4 occurrences of the string $a^2b^2c^2$. Hence,

$$\hat{p}_{11} = \frac{4}{4 + 40} = 0.0909.$$

Example 4. Random samples consisting of 40 strings and 100 strings were generated using the unambiguous UPG, G_3, given below:

$$p_{11} : S \to aS$$
$$p_{21} : aS \to aaS$$
$$p_{22} : aS \to aA$$
$$p_{31} : aA \to abA$$
$$p_{41} : bA \to bbA$$
$$p_{42} : bA \to b$$

Table 2 shows the actual values of all production probabilities not equal to one (since G_3 is assumed to be proper) and the corresponding maximum-likelihood ratio estimates for the random samples, with M=40. Data for random samples of 100 strings is given in Table 3.

The percentage errors for random samples of size 40 range from 0 to 100 percent, with the average percentage error equal to 24.63. For random samples of size 100, percentage errors range from 0 to 96 percent, with the average error equal to 8.76.

TABLE 2
G_3 *with* M=40.

Sample	Actual values				Estimated values			
	p_{21}	p_{22}	p_{41}	p_{42}	$\hat{p}_{21}$	$\hat{p}_{22}$	$\hat{p}_{41}$	$\hat{p}_{42}$
1	0.01	0.99	0.20	0.80	0.0000	1.0000	0.2982	0.7018
2	0.10	0.90	0.50	0.50	0.1489	0.8511	0.5604	0.4396
3	0.10	0.90	0.90	0.10	0.0698	0.9302	0.9099	0.0901
4	0.50	0.50	0.01	0.99	0.5000	0.5000	0.0000	1.0000
5	0.50	0.50	0.50	0.50	0.4366	0.5634	0.4444	0.5556
6	0.50	0.50	0.90	0.10	0.4595	0.5405	0.8936	0.1064
7	0.90	0.10	0.01	0.99	0.8773	0.1227	0.0000	1.0000
8	0.90	0.10	0.55	0.45	0.9184	0.0816	0.5294	0.4706
9	0.99	0.01	0.99	0.01	0.9869	0.0131	0.9915	0.0085

TABLE 3
G_3 *with* M=100.

Sample	Actual values				Estimated values			
	p_{21}	p_{22}	p_{41}	p_{42}	$\hat{p}_{21}$	$\hat{p}_{22}$	$\hat{p}_{41}$	$\hat{p}_{42}$
1	0.01	0.99	0.20	0.80	0.0099	0.9901	0.1935	0.8065
2	0.10	0.90	0.50	0.50	0.0991	0.9009	0.5305	0.4695
3	0.10	0.90	0.90	0.10	0.1597	0.8403	0.8924	0.1076
4	0.50	0.50	0.01	0.99	0.5192	0.4808	0.0099	0.9901
5	0.50	0.50	0.50	0.50	0.4444	0.5556	0.5192	0.4808
6	0.50	0.50	0.90	0.10	0.5146	0.4854	0.9136	0.0864
7	0.90	0.10	0.01	0.99	0.8973	0.1027	0.0196	0.9804
8	0.90	0.10	0.55	0.45	0.8957	0.1043	0.6154	0.3846
9	0.99	0.01	0.99	0.01	0.9900	0.0100	0.9903	0.0097

In general, the average error is skewed by large percentage errors in cases where the actual production probabilities are small. Thus, a random sample of at least 100 strings is needed for reasonable estimates. The Law of Large Numbers tells us that even larger sample sizes will provide better estimates, and this is indeed the case.

5. Conclusions. It has been shown that ratio estimators are maximum-likelihood estimators for unambiguous probabilistic grammars. This result is a generalization of the proof for unambiguous context-free probabilistic grammars. The conclusion verifies the intuitive notion that the values of production probabilities can be estimated by comparing the number of times each production is used to parse a given sample of strings.

Several questions remain open. Statistical properties such as means and variances have not been computed for the ratio estimators in the non-context-free case. The author believes that this can be done using Markov chains to describe derivations. Other open problems include the following:

(1) How does ambiguity affect the results obtained in this paper?

(2) Can this method be modified to estimate production probabilities that vary in time?

(3) Given a random sample of strings, can production probability estimation be combined with inference techniques to develop a method for inferring non-context-free probabilistic grammars?

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Compiling—Volume 1: Parsing*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972.

[2] T. L. BOOTH AND R. A. THOMPSON, *Applying probability measures to abstract languages*, IEEE Trans. Comput., 22 (1973), pp. 442-450.

[3] R. CHAUDHURI AND A. N. V. RAO, *Approximating grammar probabilities: Solution of a conjecture*, J. Assoc. Comput. Mach., 33 (1986), pp. 702-705.

[4] K. E. HUMENIK AND R. S. PINKHAM, *Production probability estimators for context-free grammars*, in Proc. 14th Annual ACM Computer Science Conference, Cincinnati, OH, February 1986, pp. 173-181.

[5] ———, *Production probability estimators for context-free grammars*, J. Systems and Software, to appear.

[6] F. J. MARYANSKI AND T. L. BOOTH, *Inference of finite-state probabilistic grammars*, IEEE Trans. Comput., 26 (1977), pp. 521-536.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1057–1086, December 1989

ANALYSIS OF PREFLOW PUSH ALGORITHMS FOR MAXIMUM NETWORK FLOW*

J. CHERIYAN†‡ AND S. N. MAHESHWARI†

Abstract. The class of preflow push algorithms recently introduced by Goldberg and Tarjan for solving the maximum flow problem on a weighted digraph with n vertices and m edges is studied. Goldberg and Tarjan's $O(n^3)$ time bound for the highest distance preflow push algorithm is improved to $O(n^2\sqrt{m})$, and it is shown that this bound is tight by constructing a parametrized worst-case network. It is also shown that the $O(n^3)$ time bound is tight for the FIFO preflow push algorithm, and the $O(n^2m)$ time bound is tight for the LIFO preflow push algorithm. The maximal excess preflow push algorithm is then developed, and it is shown that it performs $O(n^2\sqrt{m})$ pushes and that this bound is tight. Based on this, the authors develop a maximum flow algorithm for the synchronous distributed model of computation that uses $O(n^2\sqrt{m})$ messages and $O(n^2)$ time, thereby improving upon the best previously known algorithms for this model.

Key words. distributed maximum flow algorithm, heuristics, maximum flow problem, parametrized worst-case network, preflow push algorithms

AMS(MOS) subject classifications. 68Q25, 90B10, 68Q10

1. Introduction.

The problem of finding a maximum flow in a network is one of the most important problems in the area of combinatorial optimization. Besides having many practical applications it is also closely related to theoretical topics such as graph connectivity and matchings.

Dinic [D70] has shown that the problem can be solved by repeatedly extracting a layered subnetwork of the given network and solving the simpler problem of finding a blocking flow for the layered subnetwork. All subsequent research has been directed at developing faster algorithms for the blocking flow subproblem. Recently, Goldberg [Go85] has introduced a new maximum flow algorithm, namely, the FIFO preflow push algorithm that does away with the blocking flow subproblem. Goldberg's approach was later generalised by Goldberg and Tarjan [GT88]. They introduced a class of algorithms called *preflow push algorithms.* A preflow push algorithm consists of a general scheme together with a rule for selecting a vertex having flow excess. Different algorithms can be obtained by using different rules. A *push step* is applied to the selected vertex. The most important attribute of a preflow push algorithm is the number of push steps performed by it since this is what determines the time complexity of the algorithm.

In this paper we are interested in studying preflow push algorithms. One of the motivating factors is the intrinsic appeal of simple heuristics that lead to surprisingly good performance. Indeed, a good deal of research in the area of algorithms has focused on giving tight time bounds for algorithms based on simple heuristics that solve important problems. We will give tight time bounds for several preflow push algorithms. Another motivating factor is that, due to their simplicity, preflow push algorithms are likely to dominate asymptotically faster algorithms for networks of moderate size. Furthermore, preflow push algorithms are not dependent on centralized resources and so work very well in distributed models of computation.

* Received by the editors October 14, 1987; accepted for publication (in revised form) December 27, 1988.

† Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India.

‡ Present address, FB10-Informatik, Universität des Saarlandes, 6600 Saarbrücken, Federal Republic of Germany.

In what follows, we will use n to denote the number of vertices of the network for which the maximum flow has to be found, m to denote the number of (directed) edges, and U to denote the largest absolute value of the edge capacities.

Goldberg and Tarjan [GT88] have shown that two of these algorithms, namely, the FIFO preflow push algorithm and the highest distance preflow push algorithm, perform $O(n^3)$ pushes and so achieve an $O(n^3)$ time bound.

We first show that the actual time bound for the highest distance preflow push algorithm is $O(n^2\sqrt{m})$. Although this is $O(n^3)$ for dense graphs, it is $O(n^{2.5})$ for sparse graphs. Our analysis is based on ideas used previously by Cherkasky [Ch77] and Galil [Ga80] to analyze their respective maximum flow algorithms. We then show that this bound is tight by constructing a parametrized worst-case network. Given two numbers n and m such that n is sufficiently large and m is greater than kn, where k is a constant, this network has at most n vertices and m edges, and the highest distance preflow push algorithm performs $\Theta(n^2\sqrt{m})$ pushes on it. Our construction uses techniques introduced by Galil [Ga81] to construct a parametrized worst-case network for many of the earlier maximum flow algorithms. We also show that the bounds of $O(n^3)$ and $O(n^2m)$ are tight for the FIFO and LIFO preflow push algorithms, respectively, by constructing parametrized worst-case networks for these algorithms. We then develop the maximal excess preflow push algorithm that combines the best features of the FIFO preflow push algorithm and the highest distance preflow push algorithm. We show that this algorithm performs $O(n^2\sqrt{m})$ pushes and we also construct a parametrized worst-case network on which this algorithm performs $\Theta(n^2\sqrt{m})$ pushes.

The maximal excess preflow push algorithm leads us to a maximum flow algorithm for the synchronous distributed model of computation that uses at most $O(n^2\sqrt{m})$ messages and $O(n^2)$ time. This improves on the $O(n^3)$ messages and $O(n^2)$ time algorithms of Awerbuch [Aw85b], Goldberg [Go85], and Goldberg and Tarjan [GT88]. Recently, Marberg and Gafni [MG87] have developed a maximum flow algorithm for the asynchronous distributed model of computation that uses $O(n^2\sqrt{m})$ messages and $O(n^2\sqrt{m})$ time. Clearly, this gives an algorithm for the synchronous distributed model of computation that uses $O(n^2\sqrt{m})$ messages and $O(n^2\sqrt{m})$ time. Although our distributed algorithm works only for the synchronous model, note that our time bound is better than that of [MG87] for this model.

We mention other interesting results pertaining to preflow push algorithms. Goldberg and Tarjan [GT88] have improved the FIFO preflow push algorithm to $O(nm \log n^2/m)$ by using the dynamic trees data structure. This is asymptotically the fastest known algorithm. Later, Ahuja and Orlin [AO87] developed an $O(nm + n^2 \log U)$ algorithm that was based on the scaling technique. Subsequently, Ahuja, Orlin, and Tarjan [AOT89] obtained an $O(nm \log (n/m(\log U)^{1/2}+2))$ algorithm that uses scaling and the dynamic trees data structure.

In § 2 we describe preflow push algorithms following Goldberg [Go87] and Goldberg and Tarjan [GT88], and give a few definitions. Section 3 has the $O(n^2\sqrt{m})$ bound for the highest distance preflow push algorithm. Section 4 has the parametrized worst-case network showing that this bound is tight. The $O(n^3)$ time bound for the FIFO preflow push algorithm and the $O(n^2m)$ time bound for the LIFO preflow push algorithm are also shown to be tight in § 4. Section 5 considers the maximal excess preflow push algorithm. The synchronous distributed algorithm is developed in § 6. Section 7 has conclusions.

2. Preflow push algorithms. Let $G(V, E)$ be a digraph with two distinguished vertices, a source s and a sink t, and a positive real-valued capacity $c(v, w)$ on every

edge (v, w). (By an "edge" we mean a directed edge; so (v, w) is the directed edge with tail-vertex v and head-vertex w.) For convenience define $c(v, w) = 0$ if (v, w) is not an edge. A flow on G is a real-valued function f on vertex pairs having the following three properties:

(2.1) *Skew symmetry*: $f(v, w) = -f(w, v)$ for all vertex pairs v, w.

(2.2) *Capacity constraint*: $f(v, w) \leqq c(v, w)$ for all vertex pairs v, w.

(2.3) *Flow conservation*: $\sum_w f(v, w) = 0$ for every vertex v in $V - \{s, t\}$.

The value $|f|$ of a flow is the net flow into the sink, $\sum_v f(v, t)$. The maximum flow problem is that of finding a flow of maximum value.

A preflow push algorithm computes a maximum flow in a given network by manipulating a preflow f on the network. A preflow f is a real-valued function on vertex pairs satisfying (2.1) and (2.2) above, as well as the following weaker form of (2.3):

(2.4) *Nonnegativity constraint*: $\sum_u f(u, v) \geqq 0$ for every vertex v in $V - \{s\}$.

The *flow excess* $e(v)$ of a vertex v is defined to be the net flow into v, $\sum_u f(u, v)$. A vertex v is said to be *active* if v is in $V - \{s, t\}$ and $e(v) > 0$. The *residual capacity* $r(v, w)$ of a vertex pair (v, w) with respect to a preflow f is given by $r(v, w) = c(v, w) - f(v, w)$. The residual graph with respect to a preflow f has vertex set V and has an edge (v, w) if and only if $r(v, w) > 0$.

A preflow push algorithm also maintains a valid distance labeling, where a *valid distance labeling* d is a function from the vertices to the nonnegative integers, such that $d(s) = n$, $d(t) = 0$, and $d(v) \leqq d(w) + 1$ for every edge (v, w) in the residual graph. The intent is that if $d(v) < n$ then $d(v)$ is a lower bound on the actual distance from v to t in the residual graph, and if $d(v) \geqq n$ then $d(v) - n$ is a lower bound on the actual distance to s in the residual graph. In the latter case it will turn out that there is no path from v to t in the residual graph.

A valid distance labeling is called *exact* if $d(v)$ equals the actual distance from v to t (or from v to s) in the residual graph for every vertex v with $d(v) < n$ (or $d(v) \geqq n$). Otherwise, the distance labeling is called *approximate*. The generic preflow push algorithm given below uses an exact distance labeling. This is because the worst-case networks given in §§ 4 and 5 are constructed for preflow push algorithms that use an exact distance labeling.

At each iteration of the generic algorithm an active vertex, say v, with (positive) flow excess is selected, and its excess is sent closer to the sink by making use of the distance labels of v and its neighbours in the residual graph. If the flow excess cannot be sent to vertices with smaller distance labels, then the distance label of v is increased. The algorithm terminates when every vertex in $V - \{s, t\}$ has zero flow excess. An outline of the generic algorithm follows.

```
BEGIN
preprocess: Let the initial preflow f be equal to the edge capacity on each edge emanating
from the source and zero on all other edges. Let d(s) = n and compute d(v) for all
other vertices by doing a backward breadth-first search starting from t.
WHILE there is an active vertex v DO
    BEGIN
    select: Select an active vertex, say v, using a particular rule;
    push: WHILE e(v) > 0 and there is an edge (v, w) with d(v) = d(w) + 1 and
    r(v, w) > 0 DO
        push on edge (v, w):
```

Send min $\{r(v, w), e(v)\}$ units of flow from v to w and update $e(v)$, $e(w)$, $f(v, w)$, and $f(w, v)$ accordingly;
IF there is no edge (v, w) with $d(v) = d(w) + 1$ and $r(v, w) > 0$
THEN
BEGIN
relabel: Replace $d(v)$ by min $\{1 + d(w) | r(v, w) > 0\}$;
Propagate the relabel backward in the residual graph, i.e., if there is a vertex u with $d(u) = d(v) + 1$ and v is the unique successor of u with distance label $d(v)$ then relabel u, and so on;
END;
END;
END.

Note that if the distance label $d(v)$ of a vertex v becomes greater than $n-1$ after a relabel step then there is no path from v to the sink in the residual graph. When this happens the remaining flow excess at v is pushed back along shortest available paths to the source.

The correctness and termination of the generic preflow push algorithm is proved by using the max-flow min-cut theorem [FF56]. (The proofs of the lemmas in this section can be found in [Go87] and [GT88].)

LEMMA 2.1. *The algorithm maintains the invariant that d is a valid distance labeling.*

LEMMA 2.2. *Suppose that the algorithm terminates. Then the preflow f is a maximum flow.*

The termination of the generic algorithm is shown by proving that the number of push steps and relabel steps performed by the algorithm is finite. An edge (v, w) is said to be *saturated* if $r(v, w) = 0$, i.e., if $f(v, w) = c(v, w)$, and an edge (v, w) is said to be *cleared* if $f(v, w) = 0$. A push on edge (v, w) is called a *saturating push* if $r(v, w) = 0$ after the push, otherwise, the push is called a *nonsaturating push.* We say that a push on edge (v, w) is a *nonzeroing push* if this is the first push on (v, w) after either v or w has been relabeled.

LEMMA 2.3. *For any vertex v, an application of a relabel step to v increases $d(v)$. The number of relabel steps is at most $2n-1$ per vertex and at most $2n^2$ overall.*

LEMMA 2.4. *The number of saturating push operations on edges is at most $2nm$. The number of nonzeroing push operations on edges is at most $2nm$. The number of nonsaturating push operations on edges is at most $4n^2m$.*

LEMMA 2.5. *The generic algorithm terminates after $O(n^2m)$ push and relabel steps.*

The generic algorithm can be implemented so that the total time spent in relabel steps is $O(nm)$ and the total time spent in saturating pushes is $O(nm)$. Furthermore, each nonsaturating push can be implemented in $O(1)$ time. When we use Lemma 2.5, it follows that any preflow push algorithm based on the generic algorithm runs in $O(n^2m)$ time.

Several specific algorithms can be obtained from the generic algorithm above depending on the rule used in the select step.

The FIFO *preflow push algorithm* selects all active vertices in the round-robin order. The algorithm maintains all active vertices in a FIFO queue. In the select step it selects the vertex at the front of the queue and applies a push step to it; if this creates flow excesses at any new vertices then these newly active vertices are added to the rear of the queue. Goldberg [Go85] has proved the following lemma.

LEMMA 2.6. *The* FIFO *preflow push algorithm performs $O(n^3)$ pushes, and hence it runs in $O(n^3)$ time.*

In § 4 we will show that the $O(n^3)$ bound is tight for the FIFO algorithm.

The *highest distance preflow push algorithm* selects a vertex having flow excess that is furthest from the sink in each select step. In other words, if we let $d_{\max} = \max\{d(v) | e(v) > 0\}$ then a vertex v with $d(v) = d_{\max}$ and $e(v) > 0$ is selected.

In addition to the above two algorithms, Goldberg and Tarjan also suggested that the scaling technique could be used with preflow push algorithms, and they mentioned two other preflow push algorithms.

The *maximum-value excess preflow push algorithm* selects a vertex having the maximum value of flow excess at each select step.

The LIFO *preflow push algorithm* maintains all active vertices in a stack and selects a vertex for a select step according to the last-in first-out rule. In § 4 we will show that the naive bound of $O(n^2 m)$ is tight for the LIFO algorithm.

Note that when a push step of the generic algorithm terminates (i.e., the inner while loop in the outline above) then the last push, say on edge (v, w), may have been a nonsaturating push. In this case, (v, w) becomes the *current-edge* of v. The importance of the current-edge is that, the next time v is selected for a push step, the current-edge is the first edge on which a push is done, provided that $d(v)$ is still equal to $d(w)+1$.

It is easy to see that at every step of the algorithm the current-edges, at most one edge emanating from each vertex, form a spanning forest of the residual graph. We take each tree in the forest to be rooted at the unique vertex in the tree with minimum distance label. A flow excess at vertex v is called a *maximal excess* if the subtree rooted at v has no other flow excesses.

Occasionally we will regard a flow excess as a "physical entity" originated by either a saturating push or a nonzeroing push. The reason for saying that a nonzeroing push originates a new flow excess is as follows. Suppose a vertex v having flow excess is relabeled and the next push step at v does not saturate any edge. Then we can associate only a nonzeroing push with the resulting flow excess. Each flow excess is identified (i.e., tagged) by the saturating push or nonzeroing push that originated it. When two flow excesses coalesce the resulting flow excess is identified with the constituent that was most recently pushed. We mention that the approach of regarding a flow excess as a physical entity has been used before by Shiloach and Vishkin [SV82], Goldberg [Go85], and others.

We also need the notion of a clock pulse. Consider an implementation of a preflow push algorithm in the synchronous distributed computation model (this model is described in § 6). A clock pulse of a distributed preflow push algorithm consists of all push steps that can be performed simultaneously. For example, during a clock pulse of the distributed FIFO algorithm, a push step is simultaneously performed on each active vertex, and during a clock pulse of the distributed highest distance algorithm, a push step is simultaneously performed on each active vertex v that has $d(v) = d_{\max}$. The notion of a clock pulse carries over to a sequentaial preflow push algorithm as follows. We assume that the algorithm maintains a set of active vertices called SELECTED that is initially empty. At any select step, if the algorithm finds that the set SELECTED is empty, then it uses the rule in the select step to simultaneously insert as many active vertices as possible into the set. At each subsequent select step the algorithm selects any vertex in SELECTED and deletes it from the set. Each pass over the set SELECTED is said to be a *clock pulse* of the preflow push algorithm. The notion of a clock pulse has been used before by Goldberg [Go85] and others.

In the following, we sometimes say that an edge has infinite capacity. This should be interpreted as mU or more, where U is the largest absolute value of the edge capacities.

3. An $O(n^2\sqrt{m})$ bound for the highest distance preflow push algorithm. An example network on which the highest distance preflow push algorithm is superior to the FIFO preflow push algorithm may be constructed as follows (see Fig. 1). The network consists of a path of length $n-2$ together with the source vertex s joined to alternate vertices along the path by edges of unit capacity. The sink t is located at the head of the path, and each edge in the path has infinite capacity. The FIFO algorithm performs $O(n^2)$ pushes on this network since it does $O(n)$ pushes per clock pulse and there are $O(n)$ clock pulses. The highest distance algorithm performs only $O(n)$ pushes since it does only one push per clock pulse. The highest distance algorithm has the desirable feature that it never pushes a nonmaximal flow excess, thereby saving on the total number of pushes.

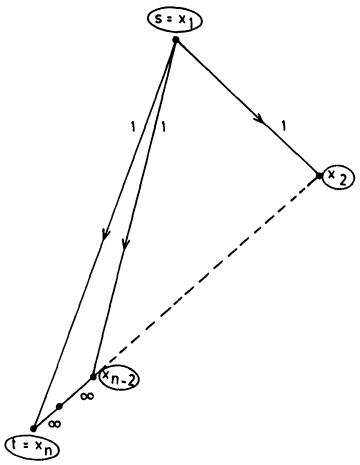


FIG. 1. *Example network on which the highest distance algorithm performs* $\Theta(n)$ *pushes and the* FIFO *algorithm performs* $\Theta(n^2)$ *pushes.*

To determine the number of nonsaturating pushes performed by the highest distance preflow push algorithm, we divide the computation into phases. A *phase* consists of all pushes that occur between two consecutive relabel steps. Note that during a phase the flow excesses that are most distant from the sink are pushed down one level at a time. The $O(n^3)$ bound is easily obtained using the following observation. Any vertex does at most one nonsaturating push per phase, and since there are $O(n^2)$ phases, the total number of nonsaturating pushes is $O(n^3)$.

At any step of the algorithm, let $d_{\max}$ be the distance of the furthest flow excess from the sink. Note that the total increase in $d_{\max}$ over the whole algorithm sums to $O(n^2)$ because each increase is caused by some vertex updating its distance label during a relabel step. The length, l_i, of phase i, is the difference between $d_{\max}$ at the start of the phase and $d_{\max}$ at the end of the phase. Goldberg and Tarjan have shown that the sum of the lengths of all phases $\sum_i l_i$ is $O(n^2)$. This follows because $\sum_i l_i$ is just the total decrease in $d_{\max}$ over the whole algorithm, and hence is bounded by $O(n)$ plus the total increase in $d_{\max}$.

We need the notion of *originating edge* of a maximal excess to improve the time bound below $O(n^3)$. Consider any step of the algorithm. Starting from a maximal excess at a vertex v, we can backtrace along a path of current-edges till we reach an edge, say (x, y), such that the last push along (x, y) was either a saturating push or a nonzeroing push. (If there is more than one such edge then an arbitrary choice can

be made.) In other words, the last push on (x, y) originated a flow excess that is now contained in the flow excess at v. The edge (x, y) is called the originating edge of the flow excess at v. The path of current-edges from (x, y) to v is called a *trajectory.*

During a phase, each nonsaturating push leaves behind a current-edge. At the end of the phase these current-edges constitute a forest, since each vertex has at most one outgoing edge. We partition these current-edges among trajectories that are vertex disjoint, except possibly for the end vertices of some trajectories. We shall account for the nonsaturating pushes by the originating edges of these trajectories.

THEOREM 3.1. *The highest distance preflow push algorithm performs at most* $O(n^2\sqrt{m})$ *nonsaturating pushes, and its time bound is* $O(n^2\sqrt{m})$.

Proof. We partition the nonsaturating pushes into two kinds of pushes and show that over the whole algorithm there are $O(n^2\sqrt{m})$ pushes of each kind.

The nonsaturating pushes that occur along a trajectory within a distace of $n/\sqrt{m}$ from the originating edge of the trajectory are called *short trajectory pushes,* and the remaining pushes that occur along a trajectory at a distance greater than $n/\sqrt{m}$ from the originating edge are called *long trajectory pushes.*

It is easily seen that over the whole algorithm there are $O(n^2\sqrt{m})$ short trajectory pushes because each trajectory starts with either a saturating push or a nonzeroing push and over the whole algorithm the total number of saturating pushes and nonzeroing pushes is $O(nm)$.

The long trajectory pushes are accounted for as follows. Observe that during any phase the trajectories are vertex disjoint and hence the number of "long" trajectories (i.e., longer than $n/\sqrt{m}$) is at most $\sqrt{m}$. Thus the number of long trajectory pushes in a phase of length l_i is at most $\sqrt{m}l_i$. Summed over all phases this is $O(n^2\sqrt{m})$.

The algorithm maintains all vertices having flow excess in a data structure so that it can easily select the highest distance vertex. This can be done at a cost of $O(1)$ per selected vertex by using a list based buckets data structure. The theorem follows. □

4. Parametrized worst-case networks for the FIFO, LIFO, and highest distance preflow push algorithms. Let n and m be two given numbers such that n is sufficiently large and m is greater than $5n$. We construct a worst-case network having at most n vertices and m edges on which the FIFO algorithm performs $\Theta(n^3)$ pushes. We also sketch how this network may be modified to obtain a worst-case network having at most n vertices and m edges on which the LIFO algorithm performs $\Theta(n^2m)$ pushes. Then we construct a worst-case network having at most n vertices and m edges on which the highest distance algorithm performs $\Theta(n^2\sqrt{m})$ pushes.

We will construct the worst-case networks by using gadgets A, B, C, D, F, and G. Each of these gadgets is a one-input one-output gadget, i.e., each gadget has one input vertex and one output vertex. When we draw a figure to illustrate a gadget (Figs. 2–7) then the input vertex of the gadget is the vertex adjacent to s and the output vertex is the vertex adjacent to t. Note that vertices s and t do not belong to the gadget.

Each finite capacity edge that we use in a gadget or in a network is a directed edge, i.e., if edge (v, w) has finite capacity then the vertex pair (w, v) has zero capacity. The direction of a finite capacity edge belonging to a gadget is given by the direction in which that edge is traversed by a shortest path from the input vertex of the gadget to the output vertex through that edge. Each infinite capacity edge (i.e., an edge of capacity at least mU) that we use is an undirected edge, i.e., it represents a pair of oppositely-oriented directed edges, both of which have infinite capacity. If the capacity of an edge is not shown in a figure then that edge has infinite capacity. The direction of a finite capacity edge belonging to a gadget may not be shown in a figure; the

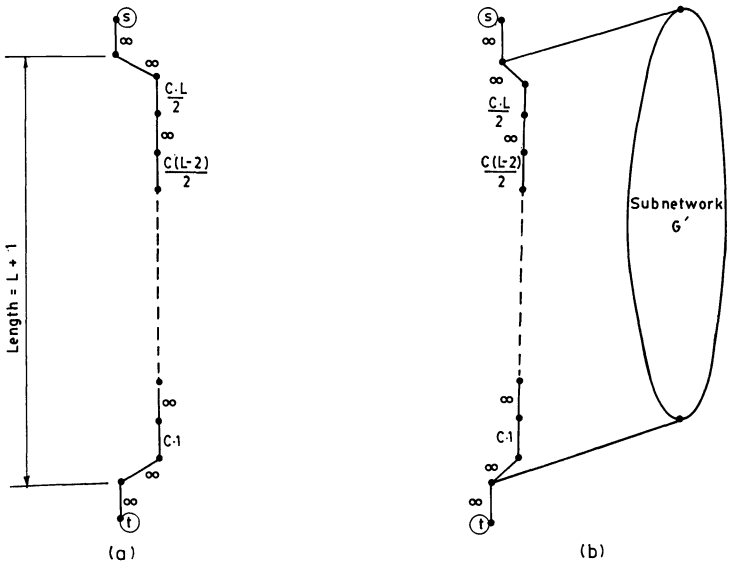


FIG. 2. (a) *Gadget A.* (b) *Gadget A being used by the highest distance algorithm to send a sequence of $L/2$ flow excesses into G'.*

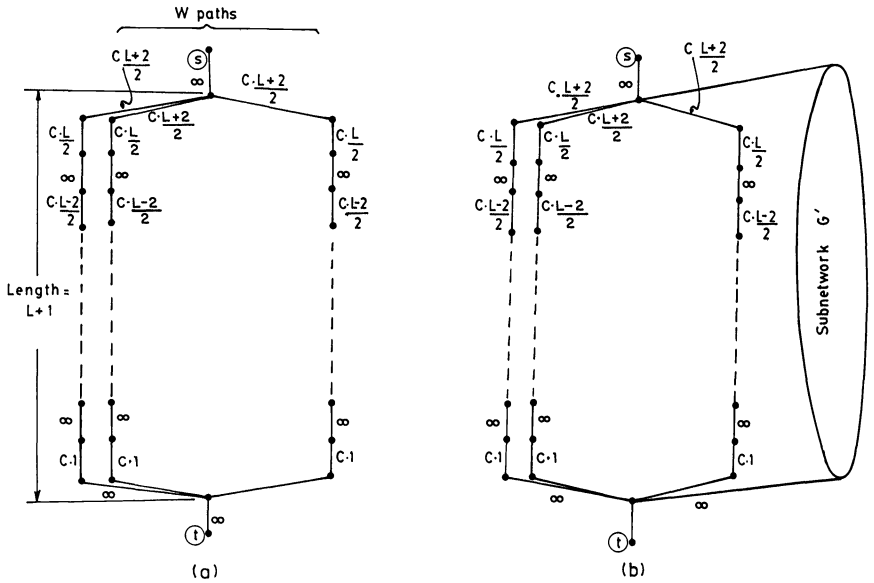


FIG. 3. (a) *Gadget B.* (b) *Gadget B being used by the highest distance algorithm to send a sequence of $WL/2$ flow excesses into G'.*

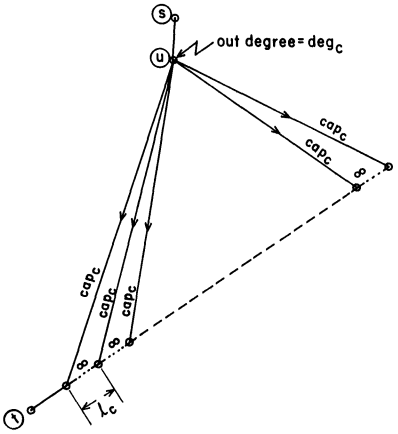


FIG. 4. *Gadget C. The distance label of input vertex u increases by l_c each time a flow excess of value* cap_c *is pushed in at u.*

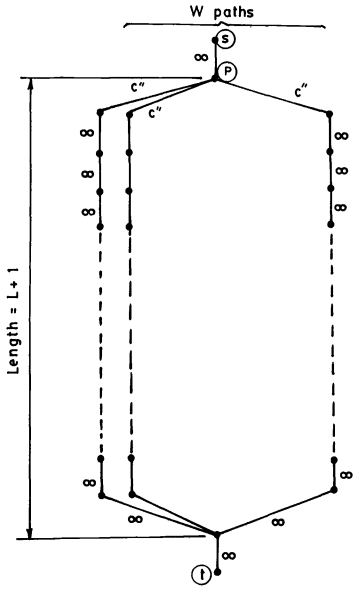


FIG. 5. *Gadget D. WL nonsaturating pushes occur in the gadget each time a flow excess of value c″W is pushed in at p.*

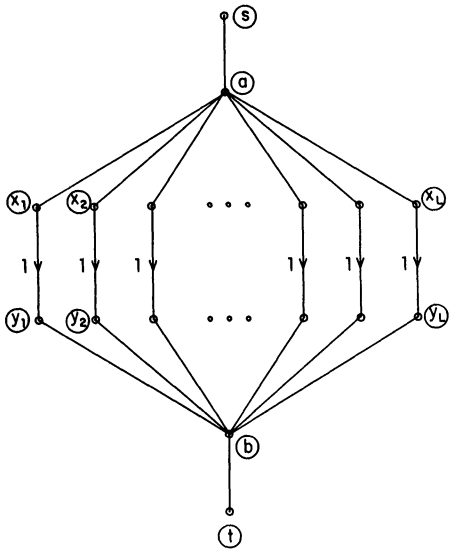


FIG. 6. *Gadget F. Suppose a large flow excess enters the gadget, then leaves it only after a delay of* $2L$ *clock pulses.*

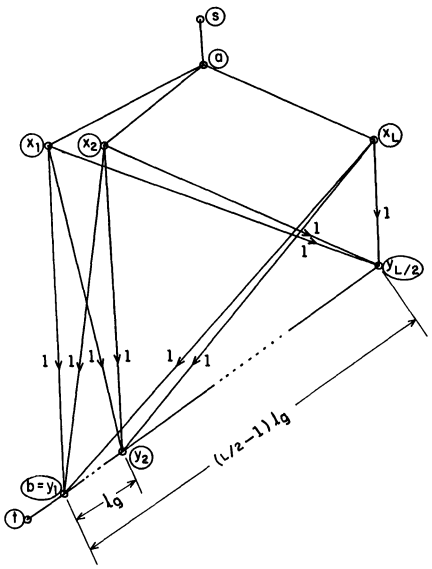


FIG. 7. *Gadget G. Suppose a large flow excess enters the gadget, then leaves it only after a delay of* $2L$ *clock pulses, and this can be repeated a total of* $L/2$ *times.*

direction of such an edge is given by the above convention. We define the *outdegree* of a vertex v to be the number of finite capacity edges with tail-vertex v.

Gadgets A and B are used only in the worst-case network for the highest distance algorithm, and gadgets F and G are used only in the worst-case network for the maximal excess algorithm (§ 5). The values of the parameters used in the gadgets and in the networks are fixed later.

Gadget A (Fig. 2(a)) is a path with $L/2$ finite capacity edges that have telescoping edge capacities. Suppose the highest distance algorithm is run on the network in Fig. 2(b); then a relabel step occurs whenever flow is pushed into an edge of capacity ic, $1 \leqq i \leqq L/2$; it can be seen that a sequence of $L/2$ flow excesses is sent into subnetwork G'.

Putting together W copies of gadget A, we get gadget B (Fig. 3(a)). Suppose the highest distance algorithm is run on the network in Fig. 3(b); then a sequence of $WL/2$ flow excesses is sent into subnetwork G'.

Gadget C (Fig. 4) contains an infinite capacity path of length l_c $(\deg_c - 1)$ and an input vertex u. There are $\deg_c$ edges, each of capacity cap_c, emanating from u. The head-vertices of the edges emanating from u lie in the path, and the head-vertices of two consecutive edges are located at a distance of l_c apart. If a sequence of flow excesses, each of value cap_c, is pushed into u then the distance of u from the sink increases in jumps of l_c.

Gadget D (Fig. 5) contains W vertex disjoint paths, where each path has length $L+1$. The first edge in each path has capacity c'' and the remaining edges have infinite capacity. If a flow excess of value $c''W$ is pushed into vertex p, it gives rise to WL nonsaturating pushes on the collection of paths.

Gadget F is shown in Fig. 6. It consists of two vertices a and b, and L vertex disjoint paths between a and b. Each path is of length three. The middle edge in each path has unit capacity while the other two edges in each path have infinite capacity.

Gadget G (Fig. 7) is constructed as follows. Let $\{x_1, x_2, \cdots, x_L\}$ be a set of L vertices and let $\{y_1, y_2, \cdots, y_{L/2}\}$ be a set of $L/2$ vertices. A complete bipartite graph is constructed on the vertex partition $\{x_1, x_2, \cdots, x_L\}$ and $\{y_1, y_2, \cdots, y_{L/2}\}$. Each edge in the bipartite graph has unit capacity and is directed from x_i to y_j. The input vertex a of gadget G is joined to all the vertices $\{x_1, x_2, \cdots, x_L\}$ by infinite capacity edges. A path of infinite residual capacity is constructed on the vertices $\{y_1, y_2, \cdots, y_{L/2}\}$ such that there is a distance of l_g between two consecutive vertices y_i and y_{i+1}, $1 \leqq i < L/2$, in the path. For convenience, we will refer to the output vertex of gadget G as b (instead of y_1).

When we describe the working of a preflow push algorithm on a network that contains a particular gadget, then we also use the name of that gadget to denote a shortest path from the input vertex of the gadget to the output vertex at a particular step in the running of the algorithm. For example, "Gadget C" initially denotes a path of length one, but after the first edge emanating from the input vertex u is saturated, "Gadget C" denotes a path of length $1+l_c$.

Recall that each push step performed by a preflow push algorithm either moves some flow closer to the sink or moves some flow closer to the source. The latter kind of push step occurs when the distance label of the selected vertex, say v, is greater than n, i.e., all paths leading from v to the sink have been saturated. When we describe the working of a preflow push algorithm on a network, we will completely ignore the pushes that move some flow closer to the source.

For some of the networks described below, we will see that the sequence of steps performed by a particular preflow push algorithm running on the network can be

partitioned into a number of *periods* such that the sequence of steps constituting the first period is similar to the sequence of steps constituting the second period, and so on. This allows us to describe the working of that algorithm on the network just by describing the sequence of steps that occur in the first period and by specifying the total number of periods.

In each of the networks that we construct for either the FIFO algorithm or the maximal excess algorithm, there is only one edge emanating from the source, and this edge has a sufficiently large finite capacity M. When we describe the working of the FIFO algorithm or the maximal excess algorithm on one of these networks, then we will see that, at each step of the algorithm, there is one flow excess whose value is greater than M/2. We will refer to this as "the large flow excess," and, for expository reasons, we will regard it as the same flow excess moving through the network.

4.1. Parametrized worst-case networks for the FIFO and LIFO algorithms. First consider the network in Fig. 8 on which the FIFO algorithm performs $\Theta(n^2)$ pushes.

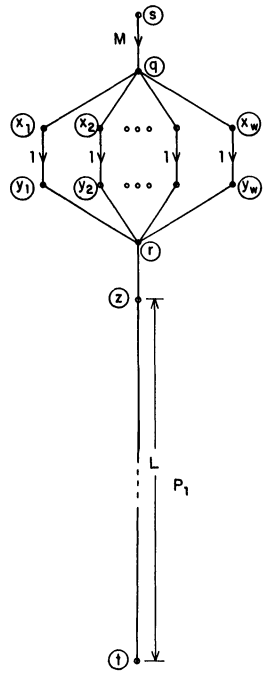


FIG. 8. *Network on which the* FIFO *algorithm performs* $\Theta(n^2)$ *pushes. The parameters are* $L = n/4$ *and* $W = n/4$.

This network contains two sets of vertices $X = \{x_1, x_2, \cdots, x_w\}$ and $Y = \{y_1, y_2, \cdots, y_w\}$ with W vertices each. For every i, $1 \leqq i \leqq W$, vertex x_i has a unit-capacity directed edge (x_i, y_i) to the corresponding vertex y_i. Note that these edges induce a bipartite graph $G'(X, Y, E')$. Each vertex in X is joined to a new vertex q by an infinite capacity edge, and each vertex in Y is joined to a new vertex r by an infinite capacity edge. There is an infinite capacity edge from r to a new vertex z, and z is joined to the sink by a path P_1 of length L with infinite residual capacity.

When the FIFO algorithm is run on this network it first creates a single large flow excess (with value M) at vertex q. The large flow excess then moves from q to vertex

x_1 and performs a saturating push on edge (x_1, y_1), thus creating a unit-capacity flow excess at y_1. (Actually, the large flow excess could move to any vertex x_i and create a unit-capacity flow excess at y_i, but the exact order in which the large flow excess visits the vertices in X does not matter.) The FIFO algorithm then pushes the unit-capacity flow excess at y_1 to the sink along the path $y_1 r z P_1 t$ and this requires $L+1$ nonsaturating pushes. Meanwhile, the large flow excess moves from x_1 to x_2 through vertex q, and then performs a saturating push on edge (x_2, y_2), thus creating a unit-capacity flow excess at y_2. Again the FIFO algorithm requires $L+1$ pushes to move the unit-capacity flow excess from y_2 to the sink. In this manner, the large flow excess visits the vertices $x_1, x_2, \cdots, x_w$, in sequence and saturates all the edges (x_i, y_i), $1 \leqq i \leqq W$, in the bipartite graph, thus creating a unit-capacity flow excess at each vertex y_i. The FIFO algorithm requires $L+1$ pushes to move each unit-capacity flow excess to the sink.

It follows that the FIFO algorithm performs $W(L+1) = \Theta(WL)$ pushes on this network. Taking $W = n/4$ and $L = n/4$, we see that the FIFO algorithm performs $\Theta(n^2)$ pushes on this network.

Let us briefly consider the LIFO preflow push algorithm. It can be seen that the LIFO algorithm also works as described above on the network in Fig. 8, and hence it performs $\Theta(n^2)$ pushes. The number of pushes performed by the LIFO algorithm can be increased to $\Theta(nm)$ by modifying the network in Fig. 8 as follows. We replace the bipartite graph $G'(X, Y, E')$ by the complete bipartite graph $G''(X, Y, E'')$. Each edge in $G''(X, Y, E'')$ has unit capacity and is directed from x_i to y_j. Also, we take $|X| = |Y| = W = \sqrt{m}/4$. It can be seen that the LIFO algorithm performs $\Theta(Lm) = \Theta(nm)$ pushes on this network (by taking $L = n/4$). This concludes the discussion on the LIFO algorithm.

Note that in the network in Fig. 8 a single large flow excess is used to create W unit-capacity flow excesses by saturating all the edges (x_i, y_j) of the bipartite graph. (The FIFO algorithm then performs $\Theta(L) = \Theta(n)$ pushes on each of these unit-capacity flow excesses.) The essential idea for obtaining a worst-case network on which the FIFO algorithm performs $\Theta(n^3)$ pushes is as follows. We clear the edges in the bipartite graph and then use it again to create W unit-capacity flow excesses. This process of creating W unit-capacity flow excesses and then clearing the bipartite graph is repeated a total of $\Theta(n)$ times. We first give an intuitive sketch of how this is done and give the details later (see Figs. 9(a)-(d)). Note that a path P_3 between q and z of length seven has been added to the network in Fig. 8. After all the edges (x_i, y_i) in the bipartite graph have been saturated, the FIFO algorithm moves the large flow excess to vertex r (Fig. 9(b)). When the large flow excess reaches r, it finds that a shortest path from r to the sink is a path $r y_i x_i q P_3 z P_1 t$. Hence, the large flow excess visits the vertices $y_1, y_2, \cdots, y_w$, in sequence and pushes back one unit of flow on each of the edges $(y_1, x_1), (y_2, x_2), \cdots, (y_w, x_w)$ (Fig. 9(c)). This achieves the aim of clearing the edges in the bipartite graph. After this, the FIFO algorithm moves the large flow excess back to q (Fig. 9(d)).

The construction of the complete worst-case network is somewhat complicated because we have to add four copies C_1, C_2, C_3, and C_4 of gadget C and several paths to the network in Fig. 8 to ensure that the large flow excess indeed moves as shown in Fig. 9. We will construct the complete network in four steps and at each step we will indicate the function of the newly added component of the network.

The network in Fig. 10 is obtained from the network in Fig. 8 by replacing the edges (r, z) and (z, r) by gadget C_1 and by adding a path P_3 between q and z of length seven. The input vertex and output vertex of gadget C_1 coincide with r and z,

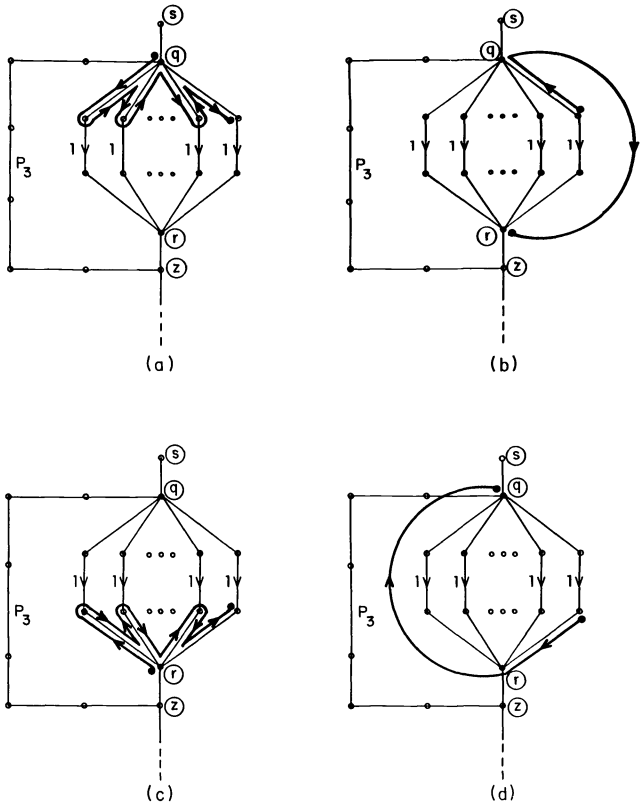


FIG. 9. (a) *A large flow excess saturates the edges* (x_1, y_1), $(x_2, y_2), \cdots, (x_w, y_w)$ *and creates a unit-capacity flow excess at each of* $y_1, y_2, \cdots, y_w$. (b) *After saturating edge* (x_w, y_w) *a large flow excess moves to vertex r.* (c) *A large flow excess clears the edges in the bipartite graph by pushing back one unit of flow on each of the edges* $(y_1, x_1), (y_2, x_2), \cdots, (y_w, x_w)$. (d) *After clearing edge* (y_w, x_w), *a large flow excess moves back to vertex q.*

respectively. The parameters of gadget C_1 are $l_c = l'$, $\text{cap}_c = W = |X| = |Y|$, and $\deg_c = n/\Delta$, where l' and Δ are constants whose values are given later. When the FIFO algorithm is run on the network in Fig. 10, consider the step when the bipartite graph has been saturated and W units of flow have been sent into vertex r. At the next push step, the first edge in gadget C_1 gets saturated and the distance from r to the sink increases from $L+1$ to $L+1+l'$. Gadget C_1 is needed because when the large flow excess reaches r (see Fig. 9(b)), it should find that a path $ry_ix_iqP_3zP_1t$ is shorter than the path $r(\text{gadget } C_1)zP_1t$, and this can be ensured by taking l' to be sufficiently large.

The network in Fig. 11 is obtained from the network in Fig. 10 by adding a path P_2 between q and r of length six, and inserting gadget C_2 into this path such that the input vertex u_2 of gadget C_2 coincides with the neighbour of r in P_2. The output vertex of gadget C_2 coincides with z and the parameters of gadget C_2 are $l_c = l'$, $\text{cap}_c = 1$ and $\deg_c = n/\Delta$. Note that the initial distance label of u_2 equals the initial distance label of r. The path P_2 has infinite residual capacity. The path P_2 is used to move the large flow excess to r as follows. After the large flow excess saturates edge (x_w, y_w) (Fig. 9(a)), it finds that the shortest path from x_w to t is the path $x_wq \cdots u_2$ (gadget $C_2)zP_1t$ and so it moves to vertex u_2. After reaching u_2, the large flow excess saturates the first edge of gadget C_2 thereby increasing the distance label of u_2 by l'. After this, the large

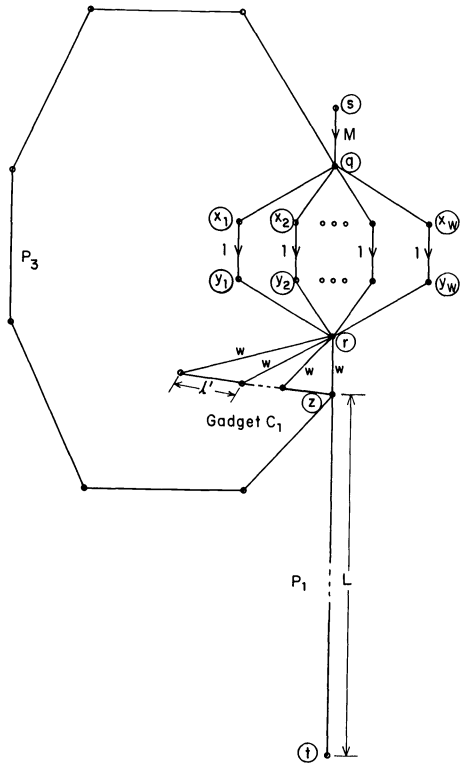


FIG. 10. *Step* 1 *in construction of worst-case network for the* FIFO *algorithm. Gadget* C_1 *and path* P_3 *have been added to the network in Fig.* 8.

flow excess finds that a shortest path from u_2 to the sink is a path $u_2 r y_i x_i q P_3 z P_1 t$ and so it moves to r. Thus the movement of the large flow excess shown in Fig. 9(b) is realized.

The network in Fig. 12 is obtained from the network in Fig. 11 by replacing path P_3 by the union of path P_3' and gadget C_3. P_3' has length six and has infinite residual capacity; the last vertex of P_3' coincides with the input vertex of gadget C_3. The output vertex of gadget C_3 coincides with z and the parameters of gadget C_3 are $l_c = l'$, $\text{cap}_c = W$ and $\deg_c = n/\Delta$. Path P_3' and gadget C_3 are needed to realize the movement of the large flow excess shown in Figs. 9(c)-(d).

Last, the network in Fig. 13 is obtained from the network in Fig. 12 by adding a path P_4 between r and q of length six and inserting gadget C_4 into this path such that the input vertex u_4 of gadget C_4 coincides with the neighbour of q in P_4. The path P_4 has infinite residual capacity. The output vertex of gadget C_4 is joined to z by a path P_5 of length six with infinite residual capacity. The parameters of gadget C_4 are $l_c = l'$, $\text{cap}_c = W$ and $\deg_c = n/\Delta$. The path P_4 is used to move the large flow excess back to q as follows. After the large flow excess saturates edge (y_w, x_w) (Fig. 9(c)), it can be seen that the distance label of u_4 equals the distance label of q, and the shortest path from y_w to t is the path $y_w r \cdots u_4$ (gadget C_4)$P_5 z P_1 t$. Hence, the large flow excess moves to vertex u_4 and saturates the first edge of gadget C_4, thereby increasing the distance label of u_4 by l'. Eventually, the large flow excess reaches q and finds that a shortest path from q to the sink is a path $q x_i y_i r$(gadget C_1)$z P_1 t$. Thus the movement of the large flow excess shown in Fig. 9(d) is realized.

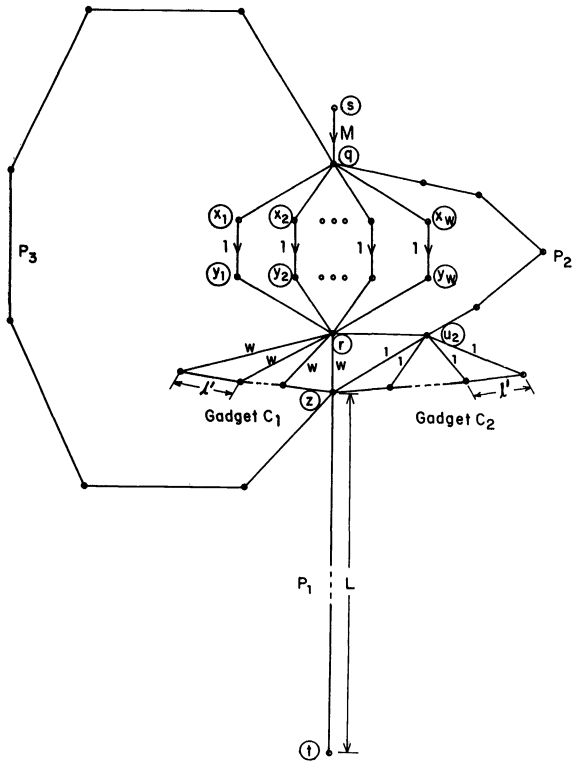


FIG. 11. *Step 2 in construction of worst-case network for the* FIFO *algorithm. Gadget C_2 and path P_2 have been added to the network in Fig.* 10.

The above sequence of steps (shown in Figs. 9(a)-(d)) constitutes one period of the FIFO algorithm running on the network in Fig. 13. The second period of the algorithm proceeds similarly. Note that in each period the FIFO algorithm performs $\Theta(WL)$ nonsaturating pushes. In each period, one edge in each of the gadgets C_1, C_2, C_3, and C_4 gets saturated. Hence, the number of periods over the whole algorithm equals the outdegree of the input vertex in any of these four gadgets, and this equals n/Δ.

We take the parameters to be $W = n/4$, $L = n/4$, $l' = 20$, and $\Delta = 320$. The capacity of the edge emanating from s is taken to be $M > 4(W+1)n/\Delta$. This gives us Theorem 4.1.

THEOREM 4.1. *There is a parametrized worst-case network on which the* FIFO *preflow push algorithm performs $\Theta(n^3)$ pushes.*

A similar construction for the LIFO algorithm gives us Theorem 4.2.

THEOREM 4.2. *There is a parametrized worst-case network on which the* LIFO *Preflow push algorithm performs $\Theta(n^2m)$ pushes.*

4.2. A parametrized worst-case network for the highest distance algorithm. First, consider the network in Fig. 14. This network contains two copies D_1 and D_2 of gadget D and two copies C_1 and C_2, of gadget C. The parameters for gadgets C_1 and C_2 are taken to be $l_c = l'$, $\text{cap}_c = c'$, and $\deg_c = L/2$, where l' is a constant whose value is fixed later. Let the neighbours of vertex P_1 in gadget D_1 be $x_1, x_2, \cdots, x_w$, and let the neighbours of vertex p_2 in gadget D_2 be $y_1, y_2, \cdots, y_w$. We construct a complete bipartite graph on the vertex partition $\{x_1, x_2, \cdots, x_w\}$ and $\{y_1, y_2, \cdots, y_w\}$ such that

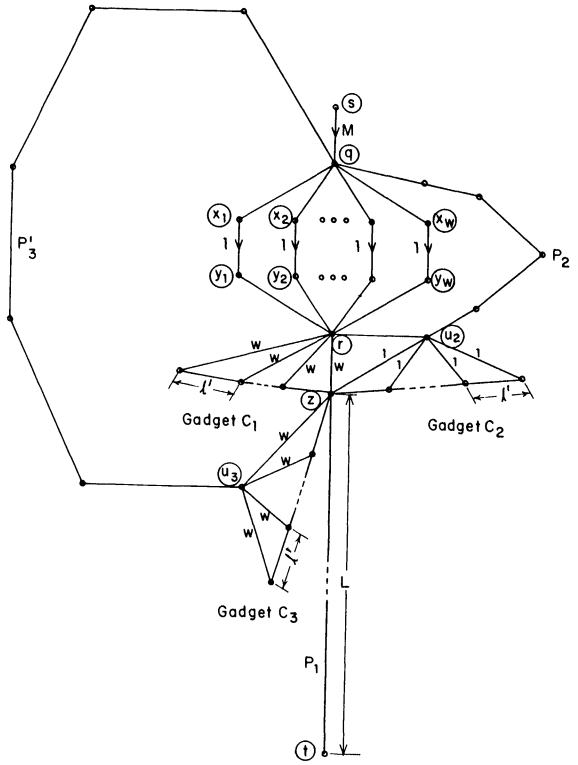


FIG. 12. *Step 3 in construction of worst-case network for the* FIFO *algorithm. Path P_3 in the network in Fig. 11 has been replaced by the union of gadget C_3 and path P'_3.*

each edge in the bipartite graph is directed from x_i to y_j and has capacity equal to c''. Note that the gadget C_1 is located $l'/2$ levels higher (with respect to distance from the sink) than gadget C_2, and gadget D_1 is located $l'/2$ levels higher than gadget D_2.

Consider the working of the highest distance algorithm on the network in Fig. 14. Suppose a sequence of W flow excesses, each of value $c''W$, is pushed into vertex p_1 then each of these flow excesses is routed through one of the vertices x_i and gives rise to WL nonsaturating pushes in gadget D_2. This sequence of flow excesses causes the first edge in gadget C_2 to saturate (we take $c' = c''W^2$ for ensuring this) and thus raises the level of gadget D_2 by l'. Now, suppose a sequence of W flow excesses, each of value $c''W$, is pushed into vertex p_2. Again each of these gives rise to WL nonsaturating pushes in gadget D_1. This process of alternately sending a sequence of flow excesses into p_1 and p_2 can be repeated $L/2$ times, until the gadgets C_1 and C_2 are completely saturated. The total number of nonsaturating pushes is $\Theta((WL)^2)$. Later we will fix the values of the parameters W and L such that $WL = \Theta(n)$; consequently, the total number of nonsaturating pushes is $\Theta(n^2)$.

Figure 15 shows a worst-case network on which the highest distance algorithm performs $\Theta(n^2)$ pushes. This network is obtained by adding two copies B_1 and B_2 of gadget B to the network in Fig. 14. Note that gadget B_1 is located one level higher (with respect to distance from the sink) gadget B_2.

When the highest distance algorithm is run on this network, it first creates a large flow excess at vertex g. The large flow excess then moves to vertex b_2 and saturates all the finite capacity edges emanating from b_2, thereby creating a flow excess of value

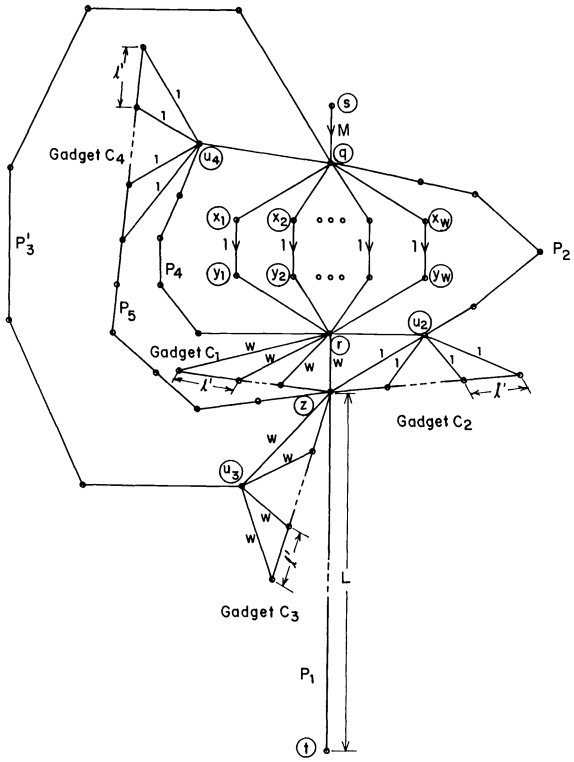


FIG. 13. *Parametrized worst-case network on which* FIFO *algorithm performs* $\Theta(n^3)$ *pushes. Gadget* C_4 *and paths* P_4 *and* P_5 *have been added to the network in Fig.* 12. *The input vertex in each of the gadgets* C_1, C_2, C_3, *and* C_4 *has outdegree* $= n/\Delta$, *where* $\Delta = 320$. *The parameters are* $L = n/4$, $W = n/4$, $l' = 20$, *and the capacity* $M > 4(W+1)n/\Delta$.

$c(L+2)/2$ at each neighbour of b_2 in gadget B_2. The remaining large flow excess at b_2 then moves along the path $b_2p_2gp_1b_1$ to b_1 and saturates all the finite capacity edges emanating from b_1, thereby creating a flow excess of value $c(L+2)/2$ at each neighbour of b_1 in gadget B_1.

At the next select step of the highest distance algorithm, one of the neighbours of b_1 in gadget B_1, say vertex v, is selected. The next push step saturates the unique outgoing edge from v and creates a flow excess with value $cL/2$ at the head-vertex of this edge. Then v gets relabeled and the remaining flow excess at v, with value c, is sent to vertex p_1. From p_1, the flow excess is routed through one of the vertices x_i and gives rise to WL nonsaturating pushes in gadget D_2 (we take $c = c''W$ for ensuring this). Similarly, each remaining neighbour of b_1 in gadget B_1 sends a flow excess with value c to vertex p_1, and each of these flow excesses gives rise to WL nonsaturating pushes in gadget D_2. After this, gadget B_2 sends a sequence of W flow excesses, each with value c, to vertex p_2, and each of these flow excesses gives rise to WL nonsaturating pushes in gadget D_1. This process of gadget B_1 sending a sequence of W flow excesses to p_1 and then gadget B_2 sending a sequence of W flow excesses to p_2 is repeated a total of $L/2$ times, until there are no more flow excesses left in gadgets B_1 and B_2.

Figure 16 shows a worst-case network on which the highest distance algorithm performs $\Theta(n^2\sqrt{m})$ pushes. It is obtained from the network in Fig. 15 by adding two copies C_3 and C_4 of gadget C. These gadgets are added because after gadget B_1 (or B_2) has sent $WL/2$ flow excesses into vertex p_1 (or p_2), all the finite capacity edges in

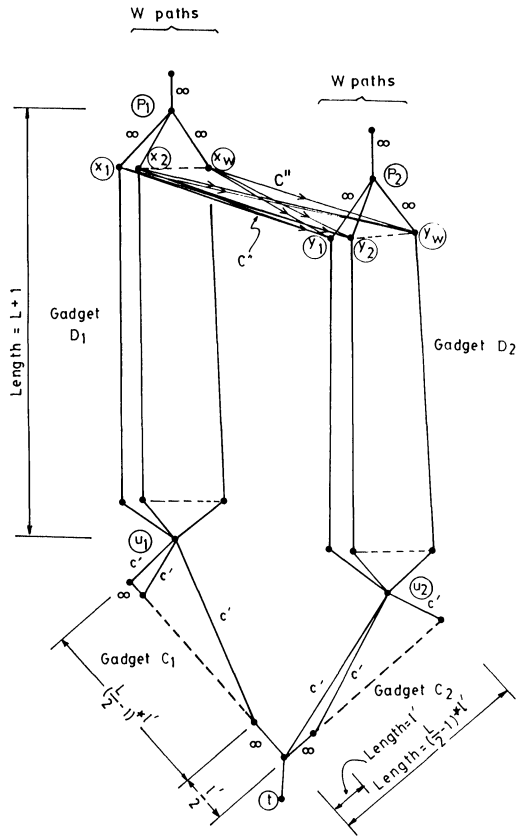


FIG. 14. *Suppose a sequence of W flow excesses, each of value c"W, is pushed in at p_1, then each flow excess causes WL nonsaturating pushes in Gadget D_2. Finally, the current edge of vertex u_2 is saturated and the distance label of u_2 increases by l'. The capacity $c' = c''W^2$.*

gadgets B_1 and B_2 and C_1 and C_2 become saturated. Gadgets C_3 and C_4 are needed to reverse flow through gadgets B_1 and B_2 and C_1 and C_2 to clear all the finite capacity edges in these gadgets. This allows us to reuse gadgets B_1 and B_2, and in fact, gadgets B_1 and B_2 are used repeatedly a total of $\Theta(n/L)$ times.

When we fix the value of l', it will turn out that the length of the longest path from vertex g to vertex z in the network in Fig. 16 is less than $6L+6$. We fix the distance from the output vertex of gadget C_4 to the sink to be $6L+6$. The parameters of gadgets C_3 and C_4 are taken to be $l_c = 12L+12$, $\text{cap}_c = 1$, and $\deg_c = W/12$.

When the highest distance algorithm runs on the network in Fig. 16, it first performs the same sequence of steps as the highest distance algorithm running on the network in Fig. 15. Consider the situation at the end of this sequence of steps. All the flow excess in the network has accumulated at vertex z to form a single large flow excess. The next push step saturates the first edge of gadget C_3. Hence, the distance label of z increases by $12L+12$ and now a path z(gadget B_2)b_2p_2g(gadget C_4) $\cdots$ t becomes a shortest path from z to the sink. It can be seen that the flow is then sent back through gadgets B_2, B_1, C_2 and D_2, and C_1 and D_1 to vertex g. Eventually, all the flow excess in the network (except the flow excess at t) accumulates at vertex g to form a single large flow excess. The next push step saturates the first edge of gadget C_4. Hence, the distance label of g increases by $12L+12$ and now a path gp_2b_2(gadget B_2)z(gadget

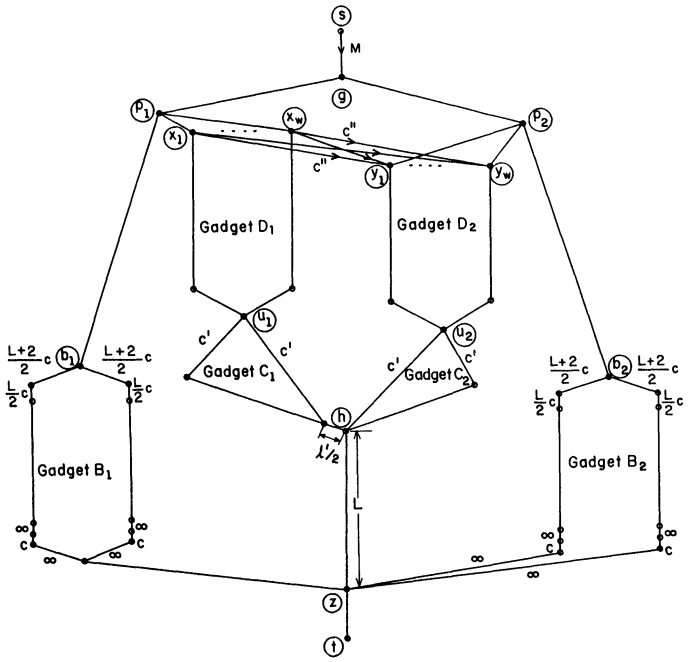


FIG. 15. *Network on which the highest distance algorithm performs* $\Theta(n^2)$ *pushes. Gadgets* B_1 *and* B_2 *have been added to the network in Fig.* 14. *The parameters are* $L = n/\sqrt{m}$, $W = \sqrt{m}/8$, *and* $l' = 8$. *The capacities are* $c'' = 4W^2$, $c = c''W$, $c' = c''W^2$, *and* $M = (L+2)Wc$.

$C_3)t$ becomes a shortest path from g to the sink. The sequence of steps from the start of the algorithm to the push step that saturates the first edge in gadget C_4 constitutes the first period of the highest distance algorithm running on the network in Fig. 16. The second period proceeds similarly. During each period the highest distance algorithm saturates one edge in each of the gadgets C_3 and C_4. Consequently, the number of periods equals the outdegree of vertex g (or z) and this equals $W/12$. When we take the parameters of the network to be $L = n/\sqrt{m}$, $W = \sqrt{m}/8$, and $l' = 8$, it follows that the highest distance algorithm performs $\Theta(n^2\sqrt{m})$ nonsaturating pushes on the network in Fig. 16.

One aspect of the working of the highest distance algorithm on the network in Fig. 16 remains to be discussed. At the beginning of the ith period, let $fsum_i$ denote the sum of the values of the first W flow excesses that are sent into vertex p_1 by gadget B_1. Clearly, $fsum_i$ should be greater than or equal to the capacity of the first edge in gadget C_2. Note that during each period, one unit of flow is pushed to the sink through each of gadgets C_3 and C_4. Consequently, the value of $fsum_i$ decreases by two in each period, i.e., $fsum_i = fsum_{i-1} - 2$, for $i > 1$. The worst-case network has been modified to compensate for this. The modifications are as follows. Let γ be greater than $2W/12$. In gadget C_2, there is one more edge, with capacity γ or more, emanating from the input vertex u_2, i.e., $\deg_c = L/2+1$ for gadget C_2; the capacity of one edge (x_i, y_j) is increased by γW; and the capacity of one edge emanating from vertex b_1 is increased by γ. The capacity of the edge emanating from s is taken to be $M = cW(L+2)+\gamma$. We do not go into further details, but it can be seen that the working of the highest distance algorithm remains essentially as explained above. This gives us the following theorem.

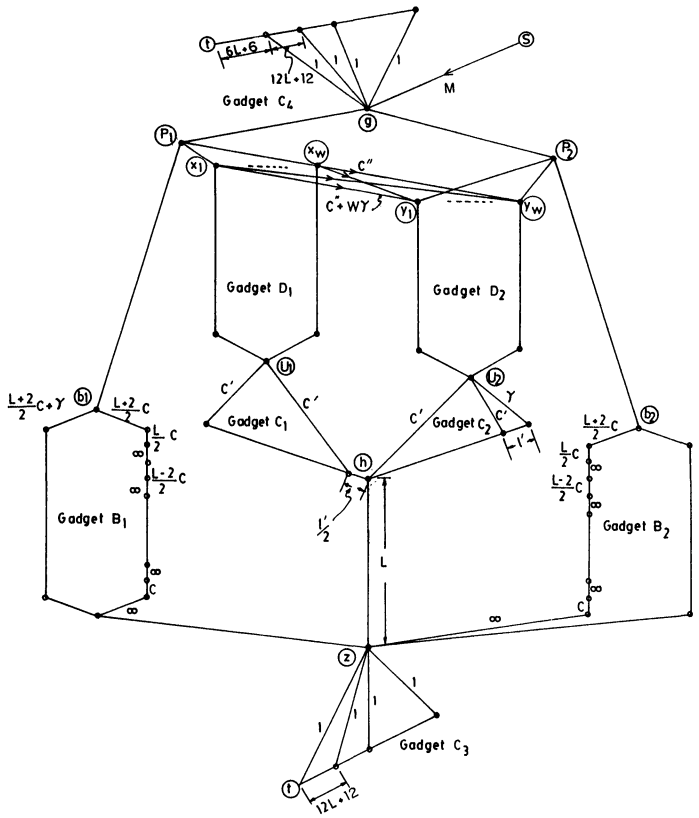


FIG. 16. *Parametrized worst-case network on which the highest distance algorithm performs* $\Theta(n^2\sqrt{m})$ *pushes. Gadgets* C_3 *and* C_4 *have been added to the network in Fig.* 15 *and in gadget* C_2 *there is one more edge emanating from* u_2. *The parameters are* $L = n/\sqrt{m}$, $W = \sqrt{m}/8$, *and* $l' = 8$. *The capacities are* $c'' = 4W^2$, $c = c''W$, $c' = c''W^2$, $\gamma > 2W/12$, *and* $M = (L+2)Wc + \gamma$.

THEOREM 4.3. *There is a parametrized worst-case network on which the highest distance preflow push algorithm performs* $\Theta(n^2\sqrt{m})$ *pushes.*

5. The maximal excess preflow push algorithm. Note that the highest distance algorithm and the FIFO algorithm are able to improve on the naive bound of $O(n^2m)$ for preflow push algorithms because they effectively exploit the mechanism of coalescing flow excesses. In this section we investigate how far the idea of coalescing flow excesses can be carried. Another motivation is to obtain an algorithm that is more suited to the distributed computation model than the highest distance algorithm.

The flow excesses can be coalesced together by pushing as many flow excesses as possible in the round-robin order while still retaining the desirable feature of the highest distance algorithm, namely, a nonmaximal excess is never pushed. This suggests that all active vertices v with maximal excesses (i.e., the current-edge subtree rooted at v contains no other flow excesses) should be selected in the round-robin order. The resulting algorithm is called the *maximal excess preflow push algorithm.*

There are two ways in which a maximal excess located at a vertex v becomes nonmaximal. The first is that some flow excess is pushed into the current-edge subtree rooted at v thereby causing the flow excess at v to become nonmaximal. Second, consider two maximal flow excesses that are moving down the current-edge tree. The one that reaches their nearest common ancestor earlier ceases to be a maximal excess.

Clearly, the maximal excess algorithm requires additional data structure overheads for detecting the above occurrences. We do not consider these overheads in detail but are primarily interested in the number of nonsaturating pushes performed by this algorithm.

The analysis given in Theorem 3.1 does not apply to the maximal excess algorithm because one vertex may do several nonsaturating pushes during a single phase (recall that a phase consists of all pushes between two consecutive relabel steps). However, we can still obtain an $O(n^2\sqrt{m})$ bound on the number of pushes by using an approach similar to that in Theorem 3.1.

THEOREM 5.1. *The maximal excess preflow push algorithm performs at most* $O(n^2\sqrt{m})$ *nonsaturating pushes.*

Proof. We partition the nonsaturating pushes into short trajectory pushes and long trajectory pushes, as in the proof of Theorem 3.1. It is easily seen that over the whole algorithm there are $O(n^2\sqrt{m})$ short trajectory pushes.

The number of long trajectory pushes is shown to be $O(n^2\sqrt{m})$ as follows. We assume that the algorithm operates in clock pulses, where during one clock pulse push steps are simultaneously performed on all maximal excesses. (Note that if the algorithm is implemented using a FIFO queue then a clock pulse corresponds to one pass over the queue.) It is easily seen that there are $O(n^2)$ clock pulses over the whole algorithm because during each clock pulse $d_{\max}$ (defined in § 2) either decreases by one or increases, and the total decrease in $d_{\max}$ over the whole algorithm is $O(n^2)$ and likewise for the total increase in $d_{\max}$.

Now we make the crucial observation that at any clock pulse the trajectories of all maximal excesses are vertex disjoint; otherwise, if the trajectories of two flow excesses overlap then either the two flow excesses have coalesced or one of the flow excesses is nonmaximal. It follows that at any clock pulse the number of long trajectory pushes is at most $\sqrt{m}$ because the number of maximal excesses having long trajectories (i.e., longer than $n/\sqrt{m}$) is at most $\sqrt{m}$. Hence, over the whole algorithm the number of long trajectory pushes is $O(n^2\sqrt{m})$. The theorem follows. □

5.1. A parametrized worst-case network for the maximal excess algorithm. Let n and m be two given numbers such that n is sufficiently large and m is greater than $140n$. We construct a worst-case network having at most n vertices and m edges on which the maximal excess algorithm performs $\Theta(n^2\sqrt{m})$ pushes. This worst-case network is similar to the worst-case network for the highest distance preflow push algorithm except that instead of using gadgets B_1 and B_2 to create flow excesses in gadgets D_1 and D_2, we take a different approach that uses $W-1$ copies of gadget F, and two copies of gadget G, and some additional paths. W and L are parameters whose values will be fixed later such that $WL=\Theta(n)$.

The gadgets D, C, F, and G are described in § 4. In the following, when we use a copy of gadget D we first delete the vertex p; this will not be mentioned explicitly.

We can use k copies of gadget F to construct a network on which the maximal excess algorithm performs $\Theta((k+1)n)$ nonsaturating pushes. Figure 17 illustrates this construction for $k=1$. The network in Fig. 17 contains a copy D_1 of gadget D, a copy F_1 of gadget F, and two new vertices x_1 and x_2. Each of the edges (x_i, y_j), $1 \leqq i \leqq 2$, $1 \leqq j \leqq W$, has unit capacity. The vertex a_1 of gadget F_1 is joined to x_1 by an edge and is joined to x_2 by a path of length four, and the vertex b_1 of gadget F_1 is joined to vertex y_1 by an edge.

The detailed working of the maximal excess algorithm on this network is as follows. A large flow excess is created at vertex x_1. Then the large flow excess performs saturating pushes on the edges (x_1, y_1), (x_1, y_2), $\cdots$, (x_1, y_w), thereby creating unit-capacity flow

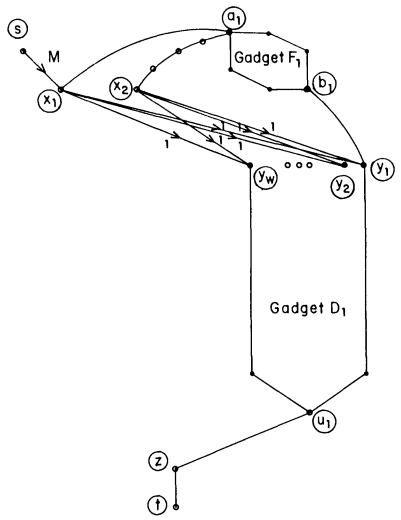


FIG. 17. *Network with one copy of gadget F on which the maximal excess algorithm performs* $\Theta(2WL)$ *pushes. This construction can be extended to give a network with k copies of gadget F on which the maximal excess algorithm performs* $\Theta((k+1)WL)$ *pushes.*

excesses at each of the vertices $y_1, y_2, \cdots, y_w$. The maximal excess algorithm then pushes these unit-capacity flow excesses to the sink and this gives rise to $\Theta(WL) = \Theta(n)$ nonsaturating pushes in gadget D_1. The large flow excess at x_1 should be moved to x_2 but it should reach x_2 only after $\Theta(n)$ nonsaturating pushes have occurred in gadget D_1, i.e., only after all the unit-capacity flow excesses in gadget D_1 have reached the sink. (Actually, when the large flow excess reaches x_2 the unit-capacity flow excess created at y_1 is still in gadget D_1; however, $\Theta(n)$ nonsaturating pushes would have occurred in gadget D_1 since the other unit-capacity flow excesses would have reached the sink.) In other words, there should be a delay of at least L clock pulses before the large flow excess reaches x_2.

Gadget F_1 is used to accomplish this delay. It can be seen that the large flow excess leaves gadget $F_1 2L$ clock pulses after entering the gadget because it saturates each of the unit-capacity edges in the gadget.

After the large flow excess reaches x_2 it again creates a unit-capacity flow excess at each of the vertices $y_1, y_2, \cdots, y_w$, and this gives rise to another $\Theta(n)$ nonsaturating pushes in gadget D_1. This completes the discussion on Fig. 17.

It can be seen that by adding vertices $x_3, x_4, \cdots, x_w$, edges (x_j, y_j) $3 \leqq i \leqq W$, $1 \leqq j \leqq W$, and $W-2$ more copies $F_2, \cdots, F_{w-1}$ of gadget F to the network in Fig. 17 and connecting gadget F_i to vertices x_i, x_{i+1}, and y_i, we obtain a network on which the maximal excess algorithm performs $\Theta(Wn)$ nonsaturating pushes.

The network in Fig. 18 contains two copies D_1 and D_2 of gadget D and $W-1$ copies $F_1, F_2, \cdots, F_{w-1}$ of gadget F, and these gadgets are connected together as shown in Fig. 18. Note that gadget D_2 is located $l'/2$ levels higher (with respect to distance from the sink) than gadget D_1, where l' is a constant whose value is fixed later. A complete bipartite graph is created on the vertex partition $\{x_1, x_2, \cdots, x_w\}$ and $\{y_1, y_2, \cdots, y_w\}$ (i.e., the "tail-vertices of paths" in gadget D_2 and the "tail-vertices of paths" in gadget D_1) by adding unit-capacity edges (x_i, y_j), for all $1 \leqq i \leqq W$ and $1 \leqq j \leqq W$. Also, a path P' between x_w and y_1 of length six and a path P'' between y_w and x_1 of length six are added to the network. Both P' and P'' have infinite residual

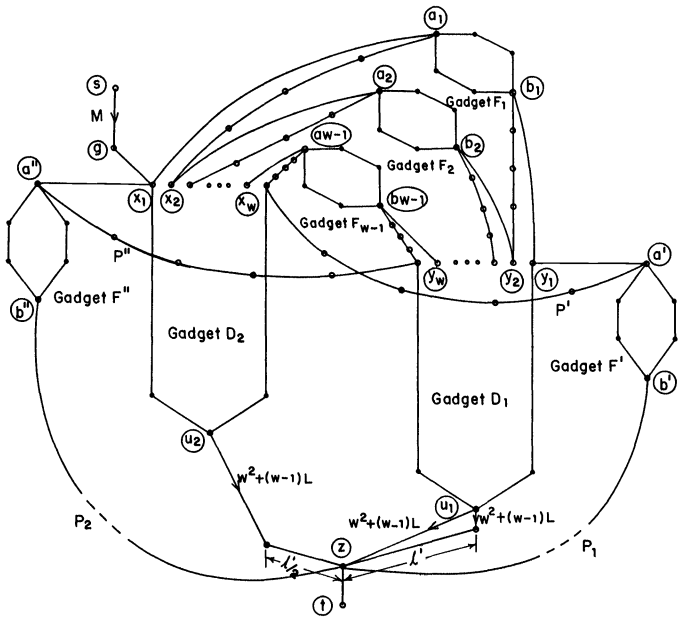


FIG. 18. *Network on which the maximal excess algorithm performs* $\Theta((W)WL)$ *pushes. Edges* (x_i, y_j) *have been omitted for clarity. Path* P' (P'') *and gadget* F' (F'') *are needed to move the large flow excess from* x_w *to* y_1 *(from* y_w *back to* x_1*). The lengths of paths* P_1 *and* P_2 *are* $L-2$ *and* $L-2+l'/2$.

capacity. Gadget F', a copy of gadget F, is inserted into path P' such that vertex a' of gadget F' coincides with the neighbour of y_1 in P'; similarly, gadget F'' is inserted into path P''. The output vertex b' of gadget F' and the output vertex b'' of gadget F'' are joined to vertex z by paths P_1 and P_2 having lengths $L-2$ and $L-2+l'/2$, respectively. The path lengths have been fixed such that $d(a')$ (the initial distance label of a') equals $d(y_1)$ and $d(a'')$ equals $d(x_1)$. Also, each finite capacity edge emanating from u_1 and from u_2 has capacity equal to $W^2+(W-1)L$.

When the maximal excess algorithm runs on this network, it moves the large flow excess to each of the vertices $x_1, x_2, \cdots, x_w$, in sequence, and for each vertex x_i the algorithm performs $\Theta(n)$ nonsaturating pushes in gadget D_1. After moving the large flow excess to x_w and saturating the edges $(x_W, y_1), \cdots, (x_W, y_W)$, the maximal excess algorithm finds that a shortest path from x_w to the sink is a path $x_w \cdots a'$ (gadget $F')b'P_1zt$, and so it moves the large flow excess from x_w along the path P' to gadget F'. The large flow excess is delayed in gadget F' for $2L$ clock pulses since it saturates each of the unit-capacity edges in gadget F'. Consider the step of the algorithm when all the unit-capacity edges in gadget F' have been saturated. All the flow excesses in gadget D_1 have reached the sink before this step, and also the edge (u_1, z) has been saturated. Hence, the maximal excess algorithm finds that a shortest path from a' to the sink is a path $a'y_1x_i$(gadget $D_2)u_2 \cdots zt$, and so it moves the large flow excess to y_1.

The large flow excess then saturates the edges $(y_1, x_1), \cdots, (y_1, x_w)$ and creates unit-capacity flow excesses at vertices $x_1, x_2, \cdots, x_w$, thereby giving rise to $\Theta(n)$ nonsaturating pushes in gadget D_2. The maximal excess algorithm then moves the large flow excess to each of the vertices $y_2, \cdots, y_w$, in sequence and for each y_i the algorithm performs $\Theta(n)$ nonsaturating pushes in gadget D_2.

After moving the large flow excess to y_w and saturating the edges $(y_w, x_1), \cdots, (y_w, x_w)$, the maximal excess algorithm moves the large flow excess from y_w along the path P'' to gadget F''. The large flow excess is then delayed in gadget F'' for $2L$ clock pulses since it saturates each of the unit-capacity edges in gadget F''. Finally, the large flow excess reaches vertex x_1. Before this step of the algorithm all flow excesses in gadget D_2 have reached the sink, and also the edge, with capacity $W^2+(W-1)L$, emanating from u_2 have been saturated.

Let us call the above sequence of steps a *macro-step*, i.e., when the maximal excess algorithm runs on the network in Fig. 18, it first performs a macro-step. Note that the maximal excess algorithm performs $\Theta(2nW)$ nonsaturating pushes during a macro-step.

The network in Fig. 19 is obtained from the network in Fig. 18 by adding two copies C_1 and C_2 of gadget C and replacing gadgets F' and F'' by two copies G' and G'' of gadget G. The parameters for gadgets C_1 and C_2 are taken to be $l_c = l'$, $\text{cap}_c = W^2+(W-1)L$, and $\deg_c = L/2$. For both gadgets G' and G'', the parameter l_g is taken to be l'. The output vertices b' and b'' of gadgets G' and G'' are joined to z by paths P_1 and P_2 of lengths $L-1$ and $L-1+l'/2$, respectively. Note that gadget C_2 is located $l'/2$ levels higher (with respect to distance from the sink) than gadget C_1, and gadget D_2 is located $l'/2$ levels higher than gadget D_1.

When the maximal excess algorithm runs on the network in Fig. 19, it first performs a macro-step. This constitutes one period of the maximal excess algorithm running on the network in Fig. 19. The second period proceeds similarly. Note that during the ith period the maximal excess algorithm saturates one edge in each of the gadgets C_1 and C_2 and also the algorithm saturates all the unit-capacity edges entering vertex y_i' in gadget G' and all the unit-capacity edges entering vertex y_i'' in gadget G'' (recall the

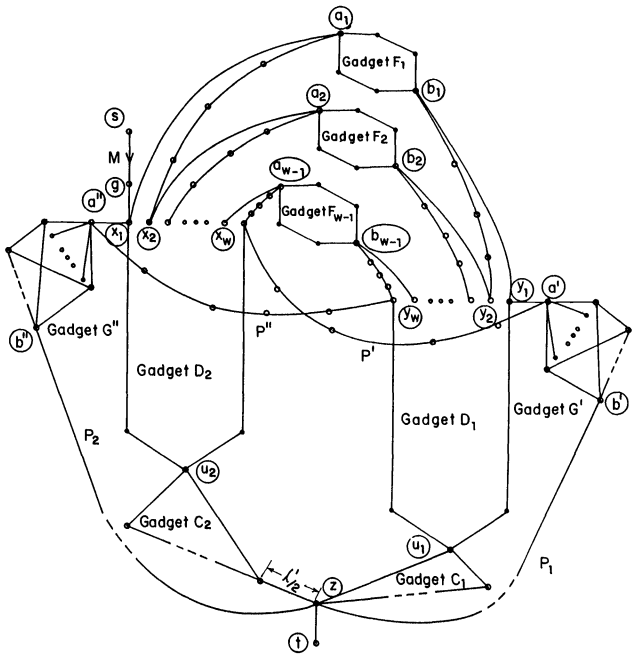


FIG. 19. *Network on which the maximal excess algorithm performs* $\Theta(n^2)$ *pushes. Gadgets* C_1 *and* C_2 *have been added to the network in Fig.* 18, *and gadgets* F' *and* F'' *have been replaced by gadgets* G' *and* G''. *Edges* (x_i, y_j) *have been omitted for clarity. The parameters are* $W = \sqrt{m}/64$, $L = 8n/\sqrt{m}$, *and* $l' = 20$, *and the capacity is* $M > 2WL(W+L)$. *The lengths of paths* P_1 *and* P_2 *are* $L-1$ *and* $L-1+l'/2$.

description of gadget G in § 4). It can be seen that the maximal excess algorithm performs $L/2$ periods, and so it performs $\Theta(nwL) = \Theta(n^2)$ pushes when it runs on the network in Fig. 19.

The $\Theta(n^2\sqrt{m})$ worst-case network for the maximal excess algorithm is shown in Fig. 20. This network is obtained from the network in Fig. 19 by adding two copies C_3 and C_4 of gadget C, and a path P_3 between the input vertex a'' of gadget G'' and vertex z. The gadgets C_3 and C_4 perform the same function as the gadgets C_3 and C_4 in the worst-case network for the highest distance algorithm (§ 4). Consider any shortest path from g to z that contains the longest path through gadget C_2; when we fix the value of l' it will turn out that the length of such a path is less than $11L$. We fix the distance from the output vertex of gadget C_4 to the sink to be $11L$. The parameters of gadgets C_3 and C_4 are taken to be $l_c = 22L$, $\text{cap}_c = 1$, and $\deg_c = W/22$. The path P_3 is needed to move the large flow excess to vertex z (this is discussed below); P_3 has length $L(1+l'/2)+2-l'/2$ and is longer than any longest path from a'' to z through gadget G''. Note that the path between g and x_1 has length four and the path between g and y_1 has length six.

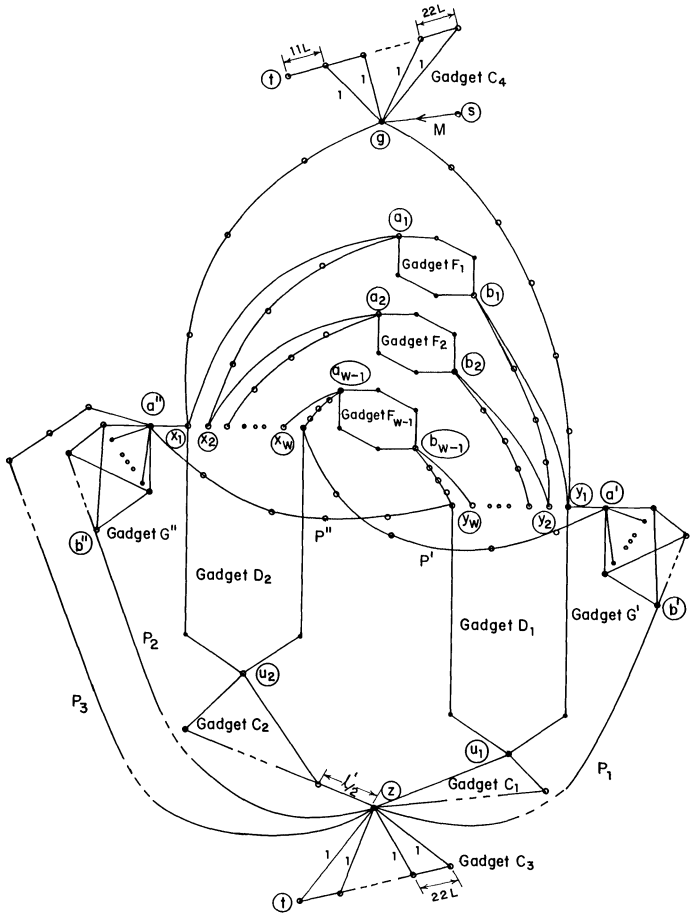


FIG. 20. *Parametrized worst-case network on which the maximal excess algorithm performs* $\Theta(n^2\sqrt{m})$ *pushes. Gadgets* C_3 *and* C_4 *and path* P_3 *have been added to the network in Fig.* 19. *Edges* (x_i, y_i) *have been omitted for clarity. The parameters are* $W = \sqrt{m}/64$, $L = 8n/\sqrt{m}$, $l' = 20$, *and the capacity is* $M > 2WL(W+L)$. *The lengths of paths* P_1, P_2, *and* P_3 *are* $L-1$, $L-1+l'/2$, *and* $L(1+l'/2)+2-l'/2$.

When the maximal excess algorithm runs on the network in Fig. 20, it first performs the same sequence of steps as the maximal excess algorithm running on the network in Fig. 19. At the end of this sequence of steps, all the finite capacity edges in gadgets C_1, C_2, G', and G'' have been saturated, and the large flow excess is at the input vertex a'' of gadget G''. Next, the large flow excess moves along path P_3 to z, and when it reaches z, all the flow excess in the network is located at z. The next push step saturates the first edge of gadget C_3. Hence, the distance label of z increases by $22L$ and now the path z(gadget C_1)(gadget D_1)$y_1 \cdots g$(gadget C_4)$\cdots t$ becomes the shortest path from z to the sink. It can be seen that the flow is then sent back through gadgets C_1 and D_1, C_2 and D_2, P_1 and G', P_2 and G'', and P_3 to vertex g. Eventually, all the flow excess in the network (except the flow excess at t) accumulates at vertex g to form a single large flow excess. The next push step saturates the first edge of gadget C_4. Hence, the distance label of g increases by $22L$ and now a path $g \cdots x_1 y_1$(gadget D_1)(gadget C_1)z(gadget C_3)t becomes a shortest path from g to the sink. The sequence of steps from the start of the algorithm to the push step that saturates the first edge in gadget C_4 constitutes the first period of the maximal excess algorithm running on the network in Fig. 20. The second period proceeds similarly. During each period the maximal excess algorithm saturates one edge in each of the gadgets C_3 and C_4. Consequently, the number of periods equals the outdegree of vertex g (or z) and this equals $W/22$. We take the parameters of the network to be $L = 8n/\sqrt{m}$, $W = \sqrt{m}/64$, and $l' = 20$, and the capacity of the edge emanating from s to be $M > 2WL(W+L)$. It follows that the maximal excess algorithm performs $\Theta(n^2\sqrt{m})$ nonsaturating pushes on the network in Fig. 20. This gives us the following theorem.

THEOREM 5.2. *There is a parameterized worst-case network on which the maximal excess preflow push algorithm performs* $\Theta(n^2\sqrt{m})$ *pushes.*

6. A distributed maximum flow algorithm. In this section, we develop a maximum flow algorithm for the synchronous distributed model of computation that uses at most $O(n^2\sqrt{m})$ messages and $O(n^2)$ time. This is a multiprocessor model with no shared memory. The graph underlying the network is realized as a processor network with one processor located at each vertex and all interprocessor communication occurs along the (bi-directional) edges of the graph. The processors are *synchronized.* The time required by each processor for its local computations is assumed to be negligible. The resource bounds of interest are the number of messages used by the algorithm and the total time taken (i.e., the number of clock pulses used). We will only use messages of length $O((\log n)+(\log U))$ bits.

The distributed maximum flow algorithm is based on the proof of Theorem 5.1. A *push message* is a message used by the algorithm to indicate the pushing of flow along some edge. Each flow excess keeps track of its originating edge and the distance it has moved since leaving the originating edge. A flow excess moves freely as long as its distance from its originating edge does not exceed $n/\sqrt{m}$.

Recall from Theorem 5.1 that, at each clock pulse, the number of long trajectory pushes (i.e., a nonsaturating push that occurs along a trajectory at a distance greater than $n/\sqrt{m}$ from the originating edge of the trajectory) is at most $\sqrt{m}$ because the long trajectories are vertex disjoint and so there are at most $\sqrt{m}$ maximal excesses that have long trajectories. To ensure that long trajectories in the distributed algorithm are vertex disjoint, each flow excess moves in *stages.* During stage i a flow excess moves for a distance of at most $(2^i n/\sqrt{m})$, i.e., it undergoes at most $(2^i n/\sqrt{m})$ nonsaturating pushes. When it has moved this distance, it stops and checks whether its trajectory is

overlapped by the trajectory of some other flow excess. For this purpose, it sends a *probe message* backward along its trajectory all the way to its originating edge. The originating edge echoes back the probe message along the trajectory to the flow excess. Upon receiving the echoed probe message the flow excess enters stage $i+1$ and resumes its movement. Of course, if a probe message is intercepted by a different flow excess, then the probe message is immediately destroyed.

THEOREM 6.1. *The above synchronous distributed algorithm for the maximum flow problem uses at most* $O(n^2\sqrt{m})$ *messages and* $O(n^2)$ *time.*

Proof. First, consider the number of clock pulses used by the distributed algorithm. Let us call a flow excess *dead* if it is waiting for its echoed probe message and its probe message has been intercepted by a different flow excess and destroyed; otherwise, the flow excess is called *alive.* Suppose the distributed algorithm maintains approximate distance labels and does lazy updates of the distance labels, i.e., a vertex v updates its distance label only immediately after a push step at v fails to reduce $e(v)$ to zero. Then a flow excess that is alive at a particular clock pulse was either alive at the preceding clock pulse or it was originated by another alive flow excess at the preceding clock pulse. Let us introduce a *token* into the distributed algorithm; the intention is that, at each clock pulse of the algorithm, the token is held by either an active vertex or by s or by t. Initially, the token is held by an active neighbour of s. At each clock pulse, the token is passed nondeterministically as follows. Suppose the token is held by a vertex v; then either the token is passed to a neighbour w of v such that a push is done on the edge (v, w) or the token continues to be held by v provided $e(v)$ remains greater than zero. It can be shown (using induction on the number of clock pulses) that there exists a token passing sequence such that the token is held by an alive flow excess at each clock pulse, and just before the algorithm terminates, the token is held by a flow excess that moves last. For this token passing sequence, it can be shown that the total distance moved by the token is $O(n^2)$, and the time required by the token to move a distance L is at most $5L$. It follows that the number of clock pulses is $O(n^2)$. The proof that the number of clock pulses for the distributed algorithm with exact distance labels is $O(n^2)$ is left to the interested reader.

Now consider the number of messages used by the algorithm. The number of push messages and probe messages used in stage zero and stage one, summed over all flow excesses originating in the algorithm, is at most $(2nm(11n/\sqrt{m}))$, which is $O(n^2\sqrt{m})$.

CLAIM. The number of stage i push messages at each clock pulse, summed over all i, $i>1$, is $O(\sqrt{m})$.

The $O(n^2\sqrt{m})$ bound on the number of messages would follow from the claim, because the total number of clock pulses is $O(n^2)$ and the number of probe messages is at most four times the number of push messages.

The essential idea for proving the claim is as follows. At each clock pulse, we associate with each stage i, $i>1$, push message a subpath of its trajectory having at least $(2^i-2)n/\sqrt{m}$ vertices such that for distinct stage i push messages the associated subpaths are vertex disjoint.

Let f_1 and f_2 be two flow excesses such that they give rise to two stage i, $i>1$, push messages that are transmitted at the same clock pulse. Suppose that the trajectories of f_1 and f_2 overlap, since otherwise there is nothing to prove. Assume that f_1 was originated earlier. Clearly, at any vertex v common to both trajectories, the (echoed) probe message that initiates stage i for f_1 precedes the push message for f_2. Let tv be the time at which the probe message for f_1 going toward f_1 passes through v, and let dv be the distance from v to the originating edge of f_1.

It can be seen that stage i for f_1 comes to an end at most $((2^i-1)n/\sqrt{m})-dv+(2^i n/\sqrt{m})$ clock pulses after tv. Similarly, stage i for f_2 can start only $2(2^i-1)n/\sqrt{m}$ clock pulses after tv. Thus dv is at most $n/\sqrt{m}$ since stage i push messages for f_1 and f_2 are transmitted at the same clock pulse.

Let p_1 denote f_1's trajectory at the end of stage $(i-1)$, and likewise for p_2. It can be seen that the suffix of p_1 obtained by deleting f_1's trajectory at the end of stage zero is vertex disjoint from the similar suffix of p_2. Hence, the number of stage i, $i>1$, push messages transmitted at any clock pulse is at most $\sqrt{m}/(2^i-2)$, and the claim follows. □

7. Conclusions. We have shown that the highest distance preflow push algorithm improves upon the $O(n^3)$ time bound and have examined several rules for selecting an active vertex in order to apply a push step to it.

We briefly consider the data structure required for implementing the maximal excess preflow push algorithm. The basic operation in this algorithm is to select an active vertex v such that the current-edge subtree rooted at v contains no other flow excesses. The basic operation corresponds to a nearest common ancestor query in a tree that changes in a "structured" way with respect to the distance labeling d. An $O(\log n)$ amortized time bound can be shown for the basic operation using the (self-adjusting) dynamic trees data structure [Ta83]. However, it is not clear that this is the most efficient data structure for this problem.

Recently, Ahuja and Orlin [AO87] have developed an $O(nm+n^2\log U)$ preflow push algorithm based on a novel use of the scaling technique. Their algorithm has $(1+\log U)$ stages. They do not exploit the mechanism of coalescing flow excesses. Rather, they control the coalescing and thereby restrict the value of the maximum flow excess that can be formed in stage i to $U/2^i$. At each stage, their algorithm moves only large enough flow excesses, by doing "restrained greedy" pushes. Subsequently, Safer [Sa88] has given an $O(nm+n^2\log U)$ message and an $O(n^2\log U)$ time distributed algorithm that was based on the above scaling algorithm.

Our work raises some questions. Note that every push on an edge is "greedy," i.e., as much flow as possible is pushed along it. Is there any advantage in doing "restrained greedy" pushes? Another possibility is to study heuristics for choosing the next outgoing edge during a push step. It is an open question whether the naive bound of $O(n^2m)$ is tight for Goldberg and Tarjan's maximum-value excess preflow push algorithm. Our $O(n^2\sqrt{m})$ messages and $O(n^2)$ time synchronous distributed algorithm does not seem to lead to an asynchronous distributed algorithm with comparable resource bounds. The straightforward approach of using Awerbuch's synchronizer [Aw85a] does not work because the synchronizer itself requires $\Theta(n^3)$ messages.

Finally, we comment on the performance of the highest distance preflow push algorithm in computational experiments. The performance of the basic algorithm (given in § 2) on random networks deteriorates due to the high overheads arising from too many relabel steps. Consider a network such that $|S|$ is $\Theta(n)$, where S denotes the set of vertices on the source side of the unique $s-t$ min-cut closest to s. Clearly, the highest distance algorithm (given in § 2) performs $\Theta(n^2)$ relabel steps on this network. Although this is not significant for worst-case networks (since the algorithm performs at least nm push steps on such networks), it does become significant on the average. Hence, an actual implementation should use some strategy to reduce the number of relabel steps. One way is to maintain approximate distance labels and to do periodical updates of the distance labels (in addition to the updates done in the relabel steps) by performing a breadth first search starting from the sink [Go87]. Another way is to

maintain the number of vertices $n(k)$ with distance labels equal to k; whenever $n(k)$ becomes zero for any k then all vertices with distance labels greater than k are disconnected from the sink, and so the algorithm avoids selecting these vertices until a maximum preflow to the sink has been found [AO87].

Acknowledgments. We thank the referees for their comments and also V. M. Malhotra, Sanjiv Kapoor, and Samar Singh for useful discussions.

REFERENCES

[AO87] R. K. AHUJA AND J. B. ORLIN, *A fast and simple algorithm for the maximum flow problem*, Working paper, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[AOT89] R. K. AHUJA, J. B. ORLIN, AND R. E. TARJAN, *Improved time bounds for the maximum flow problem*, SIAM J. Comput., 18 (1989), pp. 939–954.

[Aw85a] B. AWERBUCH, *Complexity of network synchronization*, J. Assoc. Comput. Mach., 32 (1985), pp. 804–823.

[Aw85b] ———, *Reducing complexities of the distributed max-flow and breadth-first-search algorithms by means of network synchronization*, Networks, 15 (1985), pp. 425–437.

[Ch77] B. V. CHERKASKY, *Algorithm of construction of maximal flow in networks with complexity of* $O(V^2\sqrt{E})$ *operations*, Math. Methods Solution Econom. Prob., 7 (1977), pp. 112–125.

[D70] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[FF56] L. R. FORD AND D. R. FULKERSON, *Maximal flows through a network*, IRE Trans. Inform. Theory, 2 (1956), pp. 117–119.

[Ga80] Z. GALIL, *An* $O(V^{5/3}E^{2/3})$ *algorithm for the maximal flow problem*, Acta Inform., 14 (1980), pp. 221–242.

[Ga81] ———, *On the theoretical efficiency of various network flow algorithms*, Theoret. Comput. Sci., 14 (1981), pp. 103–111.

[Go85] A. V. GOLDBERG, *A new max-flow algorithm*, Tech. Report MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.

[Go87] ———, *Efficient graph algorithms for sequential and parallel computers*, Ph.D. thesis MIT/LCS/TR-374, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[GT88] A. V. GOLDBERG AND R. E. TARJAN, *A new approach to the maximum flow problem*, J. Assoc. Comput. Mach., 35 (1988), pp. 921–940.

[MG87] J. M. MARBERG AND E. GAFNI, *An* $O(n^2m^{1/2})$ *distributed max-flow algorithm*, in Proc. Internat. Conference on Parallel Processing, Sartaj K. Sahni, ed., Pennsylvania State University Press, University Park, London, 1987.

[Sa88] H. M. SAFER, *Scaling algorithms for distributed max flow*, Working Paper, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[SV82] Y. SHILOACH AND U. VISHKIN, *An* $O(n^2 \log n)$ *parallel max-flow algorithm*, J. Algorithms, 3 (1982), pp. 128–146.

[Ta83] R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1087–1097, December 1989

POLYNOMIAL SPACE COUNTING PROBLEMS*

RICHARD E. LADNER†

Abstract. The classes of functions $\#PSPACE$ and $\natural PSPACE$ that are analogous to the class $\#P$ are defined. Functions in $\#PSPACE$ count the number of accepting computations of a nondeterministic polynomial space bounded Turing machine, and functions in $\natural PSPACE$ count the number of accepting computations of nondeterministic polynomial space bounded Turing machines that on each computation path make at most a polynomial number of nondeterministic choices. In contrast to what is known about $\#P$, exact characterizations of both $\#PSPACE$ and $\natural PSPACE$ are found. In particular, $\#PSPACE = FPSPACE$ (the class of functions computable in polynomial space) and $\natural PSPACE = FPSPACE(poly)$ (the class of functions computable in polynomial space with output length bounded by a polynomial). Both $\#PSPACE$ and $\natural PSPACE$ can be characterized by counting problems related to alternating Turing machines. Both $\#PSPACE$ and $\natural PSPACE$ have natural complete functions. It is an easy observation that $FP \subseteq \#P \subseteq \natural PSPACE$, where FP is the class of functions computable in polynomial time. Relativization to oracles is considered, as are approximation techniques for obtaining a better understanding of whether either of the above inclusions is proper.

Key words. computational complexity, polynomial space, alternating polynomial time, counting problems

AMS(MOS) subject classification. 68

1. Introduction. In this paper we consider counting problems that arise in nondeterministic polynomial space bounded ($NPSPACE$) computations and in alternating polynomial time bounded ($APTIME$) computations. For example, the class of functions $\#PSPACE$[1] is defined by $f: \Sigma^* \to N \in \#PSPACE$ if and only if there is a nondeterministic Turing machine M that runs in polynomial space with the property that $f(x)$ equals the number of accepting computation paths of M on input x. (N is the set of natural numbers, $\{0, 1, 2, \cdots\}$.) We assume that our polynomial space bounded Turing machines, both deterministic and nondeterministic always halt, avoiding the possibility that $f(x)$ is infinite. Similarly, we define the class of functions $\#APTIME$ by $f \in \#APTIME$ if and only if there is an alternating Turing machine M that runs in polynomial time with the property that $f(x)$ equals the number of accepting computation trees of M on input x. Finally, define $FPSPACE$ to be the class of functions computable in polynomial space. It is worth noting that our $FPSPACE$ functions output only binary strings that represent members of N in a natural way. The order of output, high- or low-order bits first, does not matter because if $f \in FPSPACE$ then $g \in FPSPACE$, where $g(x) = f(x)^R$ ($g(x)$ is the reversal of $f(x)$) for all x. One main result of this paper is the following theorem.

THEOREM 1. $\#PSPACE = \#APTIME = FPSPACE$.

This exact characterization of counting problems in polynomial space contrasts with our current knowledge about $\#P$ defined by Valiant [10], [11]. The class $\#P$ is defined as the class of counting problems in NP, that is, $f \in \#P$ if and only if there is a nondeterministic Turing machine M that runs in polynomial time with the property that $f(x)$ equals the number of accepting computation paths of M on input x. Although $\#P$ is defined in a simple way in terms of $NTIME$ Turing machines it does not

* Received by the editors August 24, 1987; accepted for publication August 18, 1988. This work was supported by National Science Foundation grant DCR-8402565. Part of this research was done at the Mathematical Sciences Research Institute, Berkeley, California.

†Department of Computer Science, University of Washington, Seattle, Washington 98195.

[1] $\#PSPACE$ is pronounced "sharp p space."

seem that, in general, functions in $\#P$ are computable using an oracle in the polynomial time hierarchy. In fact, it is has been conjectured that complete functions in $\#P$ are not computable in polynomial time using an oracle in the polynomial time hierarchy. The best upper bound on $\#P$ is that it is contained in *FPSPACE*. However, it has been shown by Stockmeyer that every function in $\#P$ can be approximated by a function in $F\Delta_3^P$, the class of functions computable in polynomial time using an oracle from Σ_2^P [8]. Currently, we do not have any good characterization of $\#P$. One purpose of this paper is to try to gain more insight into $\#P$.

Because some functions in $\#PSPACE$ have exponential length and all functions in $\#P$ must have polynomial length, it is a simple observation that

$$\#P \neq \#PSPACE.$$

This inequality is a little artificial, leading us to consider restrictions of $\#PSPACE$ that might give us a more interesting comparison between $\#P$ and counting problems in polynomial space. Define the class of functions $\natural PSPACE$[2] by $f \in \natural PSPACE$ if and only if there is a nondeterministic Turing machine M that runs in polynomial space and that makes only a polynomial number of nondeterministic moves (while making potentially exponentially many moves) with the property that $f(x)$ equals the number of accepting computation paths of M on input x. With this restriction the length of $f(x)$ is bounded by a polynomial in $|x|$. Clearly, $\#P \subseteq \natural PSPACE$ and equality is not out of the question. However, $\natural PSPACE$ has some nice characterizations, which leads us to question the possibility that $\#P = \natural PSPACE$.

Let $\natural APTIME$ be the class of functions f such that there is an *APTIME* Turing machine M with the property that $f(x)$ equals the number of equivalence classes of accepting computation trees of M on input x, where two accepting computation trees are equivalent if the initial sequence of existential moves from the initial configuration to the first universal configuration is the same for both. At first glance, the class $\natural APTIME$ appears to be a bit bizarre. However, it turns out to be the natural counting class for alternating computations that corresponds to $\natural PSPACE$. Define $FPSPACE(poly)$ to be the class of functions that is computable in polynomial space and whose outputs are bounded in length by a polynomial. We can characterize $\natural PSPACE$ in Theorem 2.

THEOREM 2. $\natural PSPACE = \natural APTIME = FPSPACE(poly)$.

We define the notion of polynomial-time reducibility between counting problems that make it possible to compare one counting problem to another even though the counting problems may be functions that are exponential in length. We say that a function f is reducible to a function g in polynomial time ($f \leqq^P g$) if there is function h that is computable in polynomial time such that for all x, $f(x) = g(h(x))$. A function k is complete for a class of functions C if (i) $k \in C$ and (ii) for all $f \in C$, $f \leqq^P k$. There are natural counting problems that are complete for each of $\#PSPACE$ and $\natural PSPACE$. For example, let $\#NFA$ be the counting problem defined as follows: if M is a nondeterministic finite automaton, then $\#NFA(M)$ equals the number of input strings not accepted by M. In case the number of input strings not accepted by M is infinite, define $\#NFA(M)$ equal to $k = s^{2^q-1}+1$, where s is the number of input symbols of M and q is the number of states in M. The reason for this choice is that if the number of strings not accepted by M is finite, then the number of such nonaccepted strings must be $<k$. Then,

$$\#NFA \text{ is complete for } \#PSPACE.$$

Another example is $\natural QBF$, which is defined as follows. Let $A = \exists x_1 \forall x_2 \exists x_3 \cdots$

[2] $\natural PSPACE$ is pronounced "natural p space."

$Qx_mB(x_1, x_2, \cdots, x_m)$ be a quantified Boolean formula. Then $\natural QBF(A)$ equals the number of x_1's such that A is true. We have

$$\natural QBF \text{ is complete for } \natural PSPACE.$$

There are more natural complete problems that we describe later in the paper. Complete problems give us some information about the difficulty of $\#P$. For example, it can be shown that $\natural QBF \in \#P$ if and only if $\#P = \natural PSPACE$.

Let FP be the class of functions computable in polynomial time. The inclusions

$$FP \subseteq \#P \subseteq \natural PSPACE$$

are clear, and these inclusions relativize to an arbitrary oracle. It is an open question whether or not one or both of the inclusions are reversible. Using the techniques of Baker, Gill, and Solovay [1] we can construct computable oracles A, B, and C such that $FP^A = \natural PSPACE^A$, $FP^B \neq \#P^B$, and $\#P^C \neq \natural PSPACE^C$. These results indicate that more information about these inclusions may be very difficult to come by.

Stockmeyer has shown that functions in $\#P$ can be approximated by functions computable using an oracle in the polynomial hierarchy [8]. We give a generalization of this result to a class of counting problems defined by *APTIME* Turing machines with a bounded number of alternations. On the other hand, it is a fairly easy observation that if the functions in $\natural PSPACE$ can be approximated using an oracle in the polynomial hierarchy, then *PSPACE* is included in the polynomial hierarchy. This is some evidence that $\natural PSPACE$ is harder than $\#P$.

In § 2 we will clarify the definitions we have made so far and make new ones that will be used later in the paper. In § 3 we give proofs of Theorems 1 and 2. In § 4 we describe some complete problems and describe how they are useful. In § 5 we examine some relativized counting problems. Finally, in § 6 we discuss approximating counting problems.

2. Definitions. Our model of computation is the usual Turing machine. All our Turing machines will either be polynomial time or space bounded. As such, we can assume that our space bounded Turing machines always halt without looping. If the Turing machine has output, then the output is printed on a one-way, write-only output tape, and whatever space bound is put on the Turing machine does not apply to the output tape. Thus, if M is a *PSPACE* Turing machine that computes a function f, then $|f(x)|$ can be as long as $2^{|x|^k}$ for some k. For uniformity we assume that the output of a *PSPACE* Turing machine is a binary representation of a natural number ($N = \{0, 1, 2, \cdots\}$). Recall that *FPSPACE* is the class of functions computable in polynomial space and $FPSPACE(poly)$ are those $f \in FPSPACE$ with the property that there is a constant k such that $|f(x)| \leqq |x|^k$ for all x. We assume that readers are familiar with the polynomial-time hierarchy, $\Sigma_1^P, \Sigma_2^P, \cdots$ [7]. We will also want to consider functions computable in polynomial time FP, and functions computable in polynomial time using an oracle in the polynomial-time hierarchy. Define $F\Delta_k^P$ to be the class of functions computable in polynomial time using an oracle in Σ_{k-1}^P. By definition $FP = F\Delta_1^P$.

If M is a *NPSPACE* Turing machine and if x is an input, then there may be many different computations of M that can lead to the acceptance of x. Define $\#(M, x)$ to be the number of distinct accepting computation paths of M on input x. Naturally, if M does not accept x, then $\#(M, x) = 0$ and *vice versa*. Define $\#PSPACE$ to be the class of functions f such that there is a *NPSPACE* Turing machine M where $f(x) = \#(M, x)$. Note that functions in $\#PSPACE$ can be exponential in length. A natural restriction that limits the length of function in $\#PSPACE$ is to allow only a polynomial

number of nondeterministic moves on any accepting path. Hence, we have the following definition. $\natural PSPACE$ is the class of functions f such that there is a *NPSPACE* Turing machine M that makes only a polynomial number of nondeterministic moves on any computation path and $f(x) = \#(M, x)$. The machine M may still make exponentially many moves on input x, but only a polynomial number of them are nondeterministic, when a choice can be made between alternatives.

We will assume that the reader is familiar with alternating Turing machines [4]. If M is an alternating Turing machine and x is an input, then an accepting computation tree is a tree labeled with configurations satisfying the following conditions: (i) the root is labeled with the initial configuration; (ii) each nonleaf labeled with a universal configuration has a child labeled for each successive configuration; (iii) each nonleaf labeled with an existential configuration has exactly one child, which is labeled with a successive configuration; and (iv) all the leaves are labeled with accepting configuration. Define $\#(M, x)$ to be the number of accepting computation trees of M on input x. Further define $\natural(M, x)$ to be the number of equivalence classes of accepting computation trees, where two accepting computation trees are equivalent if the paths from the initial configuration to the first universal configuration are the same in both. Note that $\#(M, x)$ can have length exponential in $|x|$ while $\natural(M, x)$ can only have length polynomial in $|x|$.

Define $\#APTIME$ to be the class of functions f such that there is an *APTIME* Turing machine M such that $f(x) = \#(M, x)$. Further define $\natural APTIME$ to be the class of functions f such that there is an *APTIME* Turing machine M with $f(x) = \natural(M, x)$. By limiting the number of alternations we can, in a natural way, also define $\#\Sigma_k^P$ and $\natural\Sigma_k^P$. Note that $\natural\Sigma_1^P$ is simply another way of expressing $\#P$. Note further that $\#\Sigma_1^P = \natural\Sigma_1^P$ and $\#\Sigma_2^P = \natural\Sigma_2^P$, but for all $k > 2$, $\#\Sigma_k^P \neq \natural\Sigma_k^P$. This latter fact follows simply from that fact that some functions in $\#\Sigma_k^P$ are exponential in length for each $k > 2$.

3. Equivalence of function classes. In this section we sketch the proof of Theorem 1:

$$\#PSPACE = \#APTIME = FPSPACE.$$

We begin by showing $\#PSPACE \subseteq \#APTIME$. A variation of the standard simulation of a nondeterministic Turing machine that runs in space $s(n)$ by an alternating Turing machine that runs in time $s^2(n)$ [4] has the property that the number of accepting computation paths of the nondeterministic machine equals the number of accepting computation trees of the alternating machine. Let M be a *NPSPACE* Turing machine and let x be an input to M of length n. Without changing the number of accepting paths of M on x, we can pad all computations so they are all the same length $2^{p(n)}$ for some polynomial $p(n)$; we can also assume there is a unique accepting configuration. Consider the following alternating algorithm, *reach*(C, D, k), which accepts if and only if configuration D is reachable from configuration C in *exactly* 2^k steps.

definition *reach*(C, D, k):
begin
  if $k = 0$ then if D is reachable from C in one step then *accept* else *reject* else
    $\bigvee E$ [*reach*$(C, E, k-1) \wedge$ *reach*$(E, D, k-1)$]
end.

The notation $\bigvee E$ means existentially choose a configuration E. The notation $\wedge$ is a binary operator meaning universally choose one of its operands. It can be shown by induction on k that the number of computation paths from C to D of length exactly 2^k equals the number of accepting computation trees of *reach*(C, D, k). The alternating

algorithm that simulates M on input x is simply the call $reach(init, acc, p(n))$, where $init$ is the initial configuration of M on input x and acc is the unique accepting configuration. Hence, the number of accepting computations of M on input x equals the number of accepting computation trees of $reach(init, acc, p(n))$.

To show that $FPSPACE \subseteq \#PSPACE$, let $f \in FPSPACE$; let M be a $PSPACE$ Turing machine that computes f. We define a $NPSPACE$ machine M' with the property that $\#(M', x) = f(x)$ for all inputs x. Let x be fixed. We define a $PSPACE$ subroutine $bit(i)$ that returns the ith bit of $f(x)$, where $bit(0)$ is the lowest-order bit. We can assume without loss of generality that there are $m = 2^{|x|^k}$ bits in $f(x)$. If $f(x)$ is actually a small number, then there will be a large number of 0's as trailing bits. We define a nondeterministic recursive procedure, $check(i)$, which has the property that $\#check(i) = bit(i) + 2 \times \#check(i+1)$, where $\#check(i)$ refers to the number of accepting paths of $check(i)$. Hence, $\#check(0) = f(x)$. The machine M' simply runs $check(0)$.

definition $check(i)$:
begin
  if $i > m$ then *reject* else
    if $bit(i) = 0$ then $check(i+1) \vee check(i+1)$
    else $(bit(i) = 1)$ *accept* $\vee$ $check(i+1) \vee check(i+1)$
end.

To complete the proof, we show $\#APTIME \subseteq FPSPACE$. Let M be an alternating Turing machine that runs in polynomial time and let x be an input. Consider the full computation of tree T of M on input x. Unlike an accepting computation tree, each existential node in T may have more than one child, one child for each choice that can be made. $\#(M, x)$ can be computed, not in polynomial space, in the following way. Construct T and label each node in T with a number starting with the leaves of T. An accepting leaf is labeled with a one and a rejecting leaf is labeled with a zero. If a universal node has k children labeled $a_1, \cdots a_k$, respectively, then label the universal node with $\prod_{i=1}^{k} a_i$. If an existential node has k children labeled $a_1, \cdots, a_k$, respectively, then label the existential node with $\sum_{i=1}^{k} a_i$. The number labeled at the root is $\#(M, x)$.

There are two difficulties in trying to do this calculation of $\#(M, x)$ within polynomial space. The first is that the tree T requires exponential space to store it all, and the second is that the numbers labeled on the nodes of the tree can be exponentially long. Both these difficulties can be overcome by realizing that the whole tree and the number labels do not all have to be written down at once and that arithmetic on exponentially long numbers can be done in polynomial space. To be more specific, if C is a configuration, define $number(C)$ to be the number of accepting subtrees of M on input x when M is started in configuration C. We can define a recursive procedure $bit(i, C)$ that is the value of the ith bit of $number(C)$. Roughly, bit is defined recursively by:

definition $bit(i, C)$:
begin
  if C is accepting then if $i = 0$ then return 1 else return 0 else
  if C is rejecting then return 0 else
  if C is universal with children $D_1, \cdots, D_k$ then
    return the ith bit of $product(number(D_1), \cdots, number(D_k))$ else
  if C is existential with children $D_1, \cdots, D_k$ then
    return the ith bit of $sum(number(D_1), \cdots, number(D_k))$
end.

It is well known that there are procedures for *product* and *sum* that on inputs of length m run in log m space. This can be seen by taking standard logarithmic depth circuits for *product* and *sum* (see [6]) and converting them to logarithmic space Turing machines using a result of Borodin [3]. To compute the ith bit of $product(number(D_1), \cdots, number(D_k))$, run the logarithmic space Turing machine for *product* without actually writing any of the output. Keep a count of the number of output symbols that have been produced so far. When the count reaches i, the ith bit is produced. While *product* is running it may request a bit from its input. At that point a recursive call to *bit* is made. The full arguments of *product* are never written down all at once. A similar computation for *sum* can be made. The depth of recursion is bounded by the height of the computation tree, which is polynomial in $|x|$. The amount of space needed at each level of recursion is logarithmic in the length of $number(C)$, which again is polynomial in $|x|$. Hence, the computation of $bit(i, C)$ can be done in polynomial space. Clearly, the number of accepting subtrees of M on input x can be output by successive calls to $bit(i, init)$, where *init* is the initial configuration.

The proof of Theorem 2,

$$\natural PSPACE = \natural APTIME = FPSPACE(poly),$$

is a bit simpler than the result we have just proven.

We begin by showing that $\natural PSPACE \subseteq \natural APTIME$. Let M be a *PSPACE* Turing machine that makes at most a polynomial number of nondeterministic moves in a computation. There is a *PSPACE* Turing machine M' that takes two inputs x and y with the property that the number of accepting computations of M on input y is exactly the number of distinct x's such that M' accepts the pair (x, y). Furthermore, the length of x is bounded by a polynomial. Essentially, M' on input (x, y) simulates M on input y except when M reaches a nondeterministic move. The ith nondeterministic move is then determined by the ith character of x. By the standard alternating time theorem [4] M' can be simulated by an *APTIME* Turing machine M'' that begins in a universal configuration. Now, consider the alternating machine M''' that on input y first existentially guesses an x, then runs M'' on the input (x, y). We have $\#(M, y) = \natural(M''', y)$.

To show $FPSPACE(poly) \subseteq \natural PSPACE$ we can use the same nondeterministic procedure *check* used to prove $FPSPACE \subseteq \#PSPACE$. In the case that the function is polynomial length bounded, the nondeterministic procedure *check* makes only a polynomial number of nondeterministic moves on any computation path.

Finally, to show that $\natural APTIME \subseteq FPSPACE(poly)$ let M be an *APTIME* Turing machine. We define a new *PSPACE* machine M' that takes pairs (x, y) as input. On input (x, y), M' simulates the initial existential moves of M on input y with these moves determined by the characters of x, reaching a configuration C of M. That is, if M can make at most d moves from an existential configuration, then $x \in \{1, 2, \cdots, d\}^*$. Initially, M' sets a pointer to the first character of x. For each initial existential move of M, M' makes the move determined by the current character of x pointed to and moves the pointer one character to the right. If C is existential and x is exhausted, or C is universal and x is not yet exhausted, then M' rejects. Otherwise, a deterministic polynomial space simulation of the alternating machine M on input y starting in configuration C is used to discover if C is the root of an accepting computation tree of M on input y. If so, M' accepts (x, y). From M' we construct a Turing machine M'' that computes $\natural(M, y)$. The Turing machine M'' simply counts the number of x's

such that M' accepts (x, y). The number of such x's is bounded in length by a polynomial in $|y|$.

4. Complete functions. We would like to be able to classify the hardest problems in $\#PSPACE$ and $\natural PSPACE$ in a way similar to the way that the NP-complete problems are classified as the hardest problems in NP. To this end we define a generalization of the Karp reducibility [5] that extends it to a relation between functions. We say that f is reducible to g in polynomial time ($f \leqq^P g$) if there is function h that is computable in polynomial time such that for all $x, f(x) = g(h(x))$. It is easy to see that $\leqq^P$ is a reflexive and transitive relation between functions. If C is any of the classes of functions we have talked about so far, $FP, \#P, \#\Sigma_k^P, \natural\Sigma_k^P, \#PSPACE$, or $\natural PSPACE$, $g \in C$ and $f \leqq^P g$ then $f \in C$ also. Define a function k to be *complete* for a class of functions C if (i) $k \in C$ and (ii) for all $f \in C, f \leqq^P k$.

Since some functions in $\#PSPACE$ are exponential in length, it is a trivial observation that $\#PSPACE$ cannot equal any of $FP, \#P, \#\Sigma_2^P, \natural\Sigma_k^P$, or $\natural PSPACE$, each of which contains only polynomial length bounded functions. But it is not inconceivable that $\#PSPACE = \#\Sigma_k^P$ for some $k > 2$. The following proposition relates the complexity of complete functions in $\#PSPACE$ to whether $\#\Sigma_k^P = \#PSPACE$ for some k.

PROPOSITION 3. *Let f be complete for $\#PSPACE$. Then, $f \in \#\Sigma_k^P$ if and only if $\#\Sigma_k^P = \#PSPACE$.*

Functions in $\natural PSPACE$ are bounded in length by a polynomial so there is some chance that $\natural PSPACE$ could equal one of $FP, \#P$, or $\natural\Sigma_k^P$ for some k. Knowing the complexity of complete functions in $\natural PSPACE$ enables us to better characterize the class $\natural PSPACE$, and perhaps even $\#P$.

PROPOSITION 4. *Let f be complete for $\natural PSPACE$.*

(1) *$f \in FP$ if and only if $FP = \natural PSPACE$,*

(2) *$f \in \#P$ if and only if $\#P = \natural PSPACE$,*

(3) *$f \in \natural\Sigma_k^P$ if and only if $\natural\Sigma_k^P = \natural PSPACE$, for $k \geqq 2$.*

It is worth noting that if f is complete for $\natural PSPACE$ and $f \in FP$, then $P = PSPACE$.

The first complete counting functions we consider are derived from the *QBF* (quantified Boolean formulas) problem first considered by Stockmeyer and Meyer [9]. A quantified Boolean formula is one of the form

$$A = \exists x_1 \forall x_2 \exists x_3 \cdots Q x_m B(x_1, x_2, \cdots, x_m),$$

where x_i is a vector of Boolean variables for $1 \leqq i \leqq m$, B is a Boolean formula, and $Q = \forall$ if m is even and $Q = \exists$ if m is odd. If A is true, then its truth can be verified by constructing a *verifying tree*, the root of which is labeled with a truth assignment for x_1. The root has $2^{|x_2|}$ children, one for each possible truth assignment of x_2. Each child of the root has exactly one child labeled with a truth assignment for x_3. This process is carried on until all the variables are assigned. Thus, each leaf corresponds to an assignment of all the variables. For each leaf in the verifying tree the formula B must be true for the assignment associated with the leaf. Define $\#QBF$ to be the function that is defined by $\#QBF(A) =$ the number of verifying trees for A. We can also define $\natural QBF(A) =$ the number of x_1's such that there is a verifying tree for A with root labeled x_1.

THEOREM 5. (1) *$\#QBF$ is complete for $\#PSPACE$,*

(2) *$\natural QBF$ is complete for $\natural PSPACE$.*

The key idea in the proof of part (1) of Theorem 5 is that given a *NPSPACE* Turing machine M and an input x a quantified Boolean formula A can be constructed in polynomial time with the property that $\#(M, x) = \#QBF(A)$. This will demonstrate that the function f is defined by $f(x) = \#(M, x)$ satisfies $f \leqq^P \#QBF$. The construction

is a modification of the construction that shows *QBF* is complete for *PSPACE* [9]. To begin with, let us describe the more standard construction and explain why it does not work to get our result. We then modify the construction to get it to work.

Let M be a *NPSPACE* Turing machine and let x be an input of length n. We can assume that on input x, M makes exactly $2^{p(n)}$ moves on every computation path before halting, where $p(n)$ is a polynomial. A configuration of M on input x can be represented by a bit vector u of length $q(n)$ for some polynomial $q(n)$. We define a quantified Boolean formula $reach_k(u_k, v_k)$ that has $2q(n)$ free variables u_k, v_k and has the meaning "the configuration represented by v_k is reachable from the configuration represented by u_k in *exactly* 2^k moves of M." For $k>0$, define

$$reach_k(u_k, v_k) = \exists z_k \, \forall u_{k-1}, v_{k-1}$$
$$[(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_k) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k) \Rightarrow reach_{k-1}(u_{k-1}, v_{k-1})].$$

In this context $u \equiv v$ means that u and v are pointwise equivalent. For $k=0$, $reach_0(u_0, v_0)$ is a quantifier-free formula expressing that v_0 follows from u_0 in exactly one move of M. The formula for $reach_k(u_k, v_k)$ can be easily rewritten so that all the quantifiers are leading quantifiers instead of embedded in the formula. Let c and d be specific bit vectors representing configurations of M. Unfortunately, the number of verifying trees of $reach_k(c, d)$ is in general larger than the number of computation paths of length 2^k from the configuration represented by c to the one represented by d. This is because in the definition of $reach_k(u_k, v_k)$ if $\neg[(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_{k-1}) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k)]$, then the body of $reach_k(u_k, v_k)$ is true no matter what the values are of the rest of the variables. To overcome this difficulty we modify the formula to completely determine the values of the existentially quantified variables in all cases. To do this we define a formula $reach1_k$ that has free variables $u_0, \cdots, u_k, v_0, \cdots, v_k, z_1, \cdots, z_k$. If $k=0$, then $reach1_k = reach_k$. If $k>0$, then

$$reach1_k = [(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_k) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k) \Rightarrow reach1_{k-1}] \wedge$$
$$[\neg[(u_{k-1} \equiv u_k \wedge v_{k-1} \equiv z_k) \vee (u_{k-1} \equiv z_k \wedge v_{k-1} \equiv v_k)] \Rightarrow \textstyle\bigwedge_{i=1}^{k-1} z_i \equiv 0].$$

In this context 0 represents the constant zero vector of length $q(n)$. Now $reach_k$ with free variables u_k, v_k is defined by

$$reach_k(u_k, v_k) = \exists z_k \, \forall u_{k-1}, v_{k-1} \, \exists z_{k-1} \, \forall u_{k-2}, v_{k-2} \cdots \exists z_1 \, \forall u_0, v_0 \, reach1_k.$$

With this definition of $reach_k$ it can be shown that if c and d are bit vectors representing configurations, then the number of verifying trees of $reach_k(c, d)$ equals the number of computation paths of length 2^k from the configuration represented by c to the one represented by d. Hence if *init* is the initial configuration and *acc* is the accepting configuration, then the formula $reach_{p(n)}(init, acc)$ has the number of verifying trees equal to the number of accepting computations of M on input x. Furthermore, the formula $reach_{p(n)}(init, acc)$ can be produced in time polynomial in n.

Part (2) of Theorem 5 can be shown using some of the ideas in the proof of $\natural PSPACE \subseteq \natural APTIME$ combined with the construction just given. Let M be a *PSPACE* Turing machine that makes at most a polynomial number of nondeterministic moves in a computation. There is a *PSPACE* Turing machine M' that takes two inputs x and y with the property that the number of accepting computations of M on input y is exactly the number x's such that M' accepts the pair (x, y). Furthermore, we can assume that for any y, if (x_1, y) and (x_2, y) are accepted by M', then $|x_1| = |x_2|$, $|x_1|$ is bounded by a polynomial, and x_1 is a binary string. From the machine M' we can construct $reach1_k$ just as before. In this case, for a particular input y to M there are

many inputs (x, y) to M'. Let $init(x)$ be the representation of the initial configuration corresponding to (x, y). Consider the following formula:

$$\exists x \, \forall w \, reach_{p(n)}(init(x), acc), \tag{1}$$

where M' runs in time $2^{p(n)}$ and w is a dummy Boolean variable. The number of x's such that there is a verifying tree for (1) is exactly the number of accepting computation paths of M on input y.

It is worth mentioning a couple of other functions that are complete for $\#PSPACE$. If M is a nondeterministic finite automaton, then $\#NFA(M)$ is the number of strings not accepted by M if the number of strings not accepted by M is finite and $\#PSPACE = s^{2^q-1}+1$ (where s is the number of symbols in the input alphabet and q is the number of states of M) otherwise. The number $s^{2^q-1}+1$ is chosen to be just larger than an upper bound on the number of strings which could be in the finite complement of a set accepted by a nondeterministic finite automaton with s input symbols and q states. If E is a regular expression, then $\#RE(E)$ is the number of strings not in the language defined by E if the number of string not in the language defined by E is finite and $\#RE(E) = s^{2^l-1}+1$ (where s is the number of symbols in the alphabet of E and l is the length of E) otherwise. The number $s^{2^l-1}+1$ is chosen to be just larger than an upper bound on the number of strings which could be in the finite complement of a language defined by a regular expression with s alphabet symbols and length l. Both $\#NFA$ and $\#RE$ are complete for $\#PSPACE$. It would be interesting to find more natural functions that are complete for either $\#PSPACE$ or $\natural PSPACE$.

5. Relativization. A good characterization of the complexity of $\#P$ is not known. We do have some interesting characterizations of $\natural PSPACE$ but for all we know $\natural PSPACE = FP$. The following inclusions are all we know for sure:

$$FP \subseteq \#P = \natural\Sigma_i^P \subseteq \natural\Sigma_2^P \subseteq \cdots \subseteq \natural PSPACE.$$

This sequence of inclusions also relativizes to any oracle so we have

$$FP^A \subseteq \#P^A = \natural\Sigma_1^{P,A} \subseteq \natural\Sigma_2^{P,A} \subseteq \cdots \subseteq \natural PSPACE^A.$$

In the definition of $\natural PSPACE^A$ we only allow strings of polynomial length to be written on the oracle tape. This makes the comparisons between the different classes of functions more fair. The following theorem gives the possible relationships between FP, $\#P$, and $\natural PSPACE$.

THEOREM 6. *There are computable oracles A, B, and C such that*:

(1) $FPA^A = \natural PSPACE^A$,

(2) $FP^B \neq \#P^B$,

(3) $\#P^C \neq \natural PSPACE^C$.

This result indicates it may be very difficult to separate FP from $\#P$ and $\#P$ from $\natural PSPACE$.

To show part (1) of Theorem 6, let A be any $PSPACE$-complete set. Suppose $f \in \natural PSPACE^A$. Since A is a member of $PSPACE$, it can be easily seen that $f \in FPSPACE(poly)$. The language $L = \{\langle x, i, b\rangle | i\text{th bit of } f(x) \text{ is } b\}$ is clearly a member of $PSPACE$. Since A is complete for $PSPACE$, L is polynomial time reducible to A. Hence, $f \in FP^A$ because, given x, the bits of $f(x)$ are encoded in the set L which can be computed in polynomial time using the set A as an oracle.

Part (2) of Theorem 6 holds for any B such that $P^B \neq NP^B$ [1]. Suppose $FP^B = \#P^B$. Let M be a $NPTIME$ oracle Turing machine that accepts L using oracle B. Let f be the function in $\#P^B$ defined by M. Thus, f is also a member of FP^B by assumption.

To check if $x \in L$ in polynomial time, compute $f(x)$ in polynomial time using the oracle B. We have $x \in L$ if and only if $f(x) > 0$. Hence, $P^B = NP^B$.

Part (3) of Theorem 6 holds for any C such that $NP^C \neq PSPACE^C$. Such sets C exist from the results of Baker and Selman [2] and Yao [12]. Suppose $\#P^C = \natural PSPACE^C$. Let M be a *PSPACE* oracle Turing machine which accepts L using oracle C. Let f be the function in $\natural PSPACE^C$ defined by M thinking of M as a nondeterministic machine. By assumption, $f \in \#P^C$. Let M' be the *NPTIME* oracle Turing machine which on oracle C defines f. The machine M' also accepts the set L. Hence, $NP^C = PSPACE^C$.

The proof technique of parts (2) and (3) of Theorem 6 combined with the result of Yao that the polynomial hierarchy can be separated using an oracle [12] can be used to show Theorem 7.

THEOREM 7. *There is an oracle A such that*

$$FP^A \subset \natural\Sigma_1^{P,A} \subset \natural\Sigma_2^{P,A} \subset \cdots \subset \natural PSPACE^A.$$

The notation $X \subset Y$ means that X is a proper subset of Y.

6. Approximation. Stockmeyer has shown that functions in $\#P$ can be approximated by functions in the polynomial hierarchy, namely by functions by $F\Delta_3^P$ [8]. This result can be generalized to functions in $\natural\Sigma_k^P$. We say that f is approximated by g if there is a constant $c \geqq 1$ such that for all $x, f(x)/c \leqq g(x) \leqq cf(x)$.

THEOREM 8. *For $k \geqq 2$, every function in $\natural\Sigma_k^P$ can be approximated by a function in $F\Delta_{k+1}^P$.*

A complete proof of Theorem 8 will not be given here since it follows immediately from an observation about a proof of Stockmeyer [8, Thm. 3.1, p. 858]. In Stockmeyer's proof a predicate of the form $z \in Acc_M(x)$, meaning "z is an accepting computation of M on input x," must be evaluated. In this context M is a *NPTIME* Turing machine so that this predicate can be checked in polynomial time. In our context this predicate would mean that "z is an initial sequence of configurations, the last of which is a universal configuration and all but the last of which are existential configurations, in an accepting computation tree of M on input x." In our context M is an *APTIME* Turing machine that starts in an existential configuration and makes at most $k-1$ alternations before halting. That is, M is a machine that defines a member of the class $\natural\Sigma_k^P$. Hence, in our context the predicate $z \in Acc_M(x)$ is computable in Π_{k-1}^P. Examining Stockmeyer's proof, we find this observation guarantees that a function in $F\Delta_{k+1}^P$ can approximate a function in $\natural\Sigma_k^P$.

Finally it is easy to show that functions in $\natural PSPACE$ are approximated by functions in the polynomial hierarchy if and only if *PSPACE* is contained in the polynomial hierarchy. To see this note that if $\{0, 1\}$-valued function f is approximated by g, then g essentially exactly computes f. This is evidence that the functions in $\natural PSPACE$ are harder to compute than those in $\#P$ or in $\natural\Sigma_k^P$ for any k.

Acknowledgments. The author thanks Sam Buss for the stimulating conversations that inspired some of these results. Thanks also go to Larry Stockmeyer for pointing out an easy way of finding oracles satisfying parts (2) and (3) of Theorem 6 and to an anonymous referee for pointing out an easy way to satisfy part (1) of Theorem 6. Also, the author thanks both referees for many useful suggestions.

REFERENCES

[1] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the* P = ?NP *question*, SIAM J. Comput., 4 (1975), pp. 431-442.

[2] T. BAKER AND A. SELMAN, *A second step toward the polynomial hierarchy*, Theoretical Computer Science, 8 (1979), pp. 177-187.
[3] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733-744.
[4] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.
[5] R. M. KARP, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85-104.
[6] J. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.
[7] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.
[8] ———, *On approximation algorithms for #P*, SIAM J. Comput., 14 (1985), pp. 849-861.
[9] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: Preliminary report*, in Proc. 5th Annual ACM Symposium on Theory of Computing, 1973, pp. 1-9.
[10] L. G. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.
[11] ———, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410-421.
[12] A. C. YAO, *Separating the polynomial-time hierarchy by oracles*, in 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 1-5.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1098-1127, December 1989

003

SCHEDULING ARITHMETIC AND LOAD OPERATIONS IN PARALLEL WITH NO SPILLING*

DAVID BERNSTEIN†, JEFFREY M. JAFFE‡, AND MICHAEL RODEH§

Abstract. A machine model in which load operations can be performed in parallel with arithmetic operations by two separate functional units is considered. For this model, the evaluation of a set of expression trees is discussed. A dynamic programming algorithm for producing an approximate solution is described and analyzed. For binary trees its worse-case cost is at most $\min(1.091, 1+(2\log n)/n)$ times the optimal cost.

Key words. scheduling algorithms, code generation, parallel functional units

AMS(MOS) subject classifications. 68N20, 68Q25

1. Introduction. Optimal code generation for arithmetic expressions is the subject of many theoretical studies. In most previous papers the underlying machine model was assumed to be sequential. For sequential machines, linear algorithms have been designed that produce optimal code given a set of expression trees [N67], [Re69], [SU70]. Via dynamic programming wide classes of such machines may be handled [AJ76]. By contrast, generating optimal code for directed acyclic graphs is intractable [AJUa77], [BS76].

Modern computers are no longer sequential. Their design usually includes a certain degree of parallelism, most commonly by offering a number of functional units that may operate simultaneously—some units handle memory access while others manipulate (in registers) the data thus retrieved (for example, [Ru78]). We assume a machine model in which memory operations can be performed in parallel with arithmetic operations by two separate functional units. The parallelism of this kind appears in many existing computers, beginning with processors of RISC architecture [H84] to today's supercomputers, such as Cray-1 [Ru78] and Fujitsu VP-200 [HB84].

Allowing parallelism in a machine model introduces additional complications into code generation problems. In their seminal work [AJ76] Aho and Johnson proved that for a wide range of sequential machines an optimal evaluation of an expression tree can always be chosen from a class of *normal form* evaluations. This observation allowed them to use a linear time dynamic programming algorithm to provide an optimal solution to certain code generation problems. It turns out that for the proposed model of a parallel machine the Normal Form Theorem of [AJ76] does not hold, i.e., in an optimal evaluation of an expression tree it may be necessary to oscillate back and forth during the evaluation of various subtrees of certain expression trees.

This oscillatory phenomena was first investigated in [AJUb77] for sequential machines with multiregister operations, leaving open the questions of the existence of a polynomial time optimal algorithm and an efficient approximation algorithm. We continue in this direction learning more about the properties of bouncing (oscillating) evaluations. In a previous work [BJPR85], an efficient algorithm was developed to optimally schedule the operations of a forest of expression trees assuming that the number of registers in the parallel machine is unbounded. In this paper we consider

* Received by the editors July 20, 1987; accepted for publication (in revised form) December 28, 1988.
† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. This work was done while the author was a graduate student at the Technion, Israel Institute of Technology, Haifa 3200, Israel.
‡ IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.
§ IBM Israel Scientific Centre, Technion City, Haifa 32000, Israel.

a more realistic model in which the number of registers is finite, but is sufficiently large to allow generating code with no stores.

As in [AJUb77], in this work the question of existence of a polynomial time optimal algorithm is not answered. Our main result is an approximation algorithm that schedules some of the load operations of a binary expression forest in parallel with the arithmetic operations so that the completion time is at most 9.1 percent worse than the optimal. The proposed algorithm is an extension of the dynamic programming scheme of Aho and Johnson [AJ76] making it suitable to our parallel machine. Performing tedious analysis of bouncing evaluations, we prove that for binary trees, if the difference between the completion time of the evaluation order generated by the algorithm and the shortest schedule is d then the expression tree must have at least 2^d vertices. This observation leads to the result that the ratio between the completion time of the generated and the shortest schedule tends to 1 as the size of the tree grows. Moreover, when a given expression tree has a small number of vertices ($n \leqq 16$), or the number of unary arithmetic operations in a tree is at most three, or a machine has only four registers or fewer, the proposed algorithm turns to be optimal. The algorithm can be implemented in such a way that its time complexity is bounded by $O(n \min(\log n, R))$, where R is the number of machine registers.

The rest of the paper is organized as follows. In § 2 we define the machine model, describe how an expression tree is computed on such machines, and present the dynamic programming scheme. In §§ 3–6 we perform full analysis of the algorithm. Then we conclude with the extensions and open problems. Note that to shorten the paper, some of the proofs are omitted; however, the reader is referred to [BJR87] for the full version of the paper.

2. Preliminaries.

2.1. The machine model. Our machine has R general purpose registers $r_1, \cdots, r_R$ and an unbounded number of memory cells $mem_1, mem_2, \cdots$. It supports the following operations, the execution times of which are all equal (in § 7 a generalization is considered):

(1) Load operation: $(mem_i) \to (r_j)$
(2) Store operation: $(r_i) \to (mem_j)$
(3) Unary arithmetic operation: $A((r_i)) \to (r_i)$
(4) Binary arithmetic operation: $A((r_i), (r_j)) \to (r_i)$.

The major parallel feature of the machine is that it can execute a load operation concurrently with an arithmetic operation. The only restriction on the registers participating in parallel instructions is that the result of the arithmetic operation should be directed to a register different from that of the load operation. As was mentioned in the Introduction, we assume that the given expression forests can be computed without storing intermediate results.

2.2. Computation forests and their evaluation. Following [AJ76] and [SU70], computations are represented by rooted forests. An example of a forest with one tree is given in Fig. 1. Throughout the paper we will use the tree of Fig. 1 as a running example.

The leaves $L(F)$ of a computation forest F represent load operations, and the internal vertices $A(F)$ represent arithmetic operations. Let $l(F) = |L(F)|$, $a(F) = |A(F)|$, $n(F) = l(F) + a(F)$. In the rest of the paper we use the following notation. Capital letters denote aggregates (sequences, evaluations, trees, forests), while lower case letters usually denote simple objects. A capital letter with a subscript denotes a particular element of an aggregate (a vertex of a tree, an operation of an evaluation, etc.) while an aggregate with a superscript designates an instance of this aggregate.

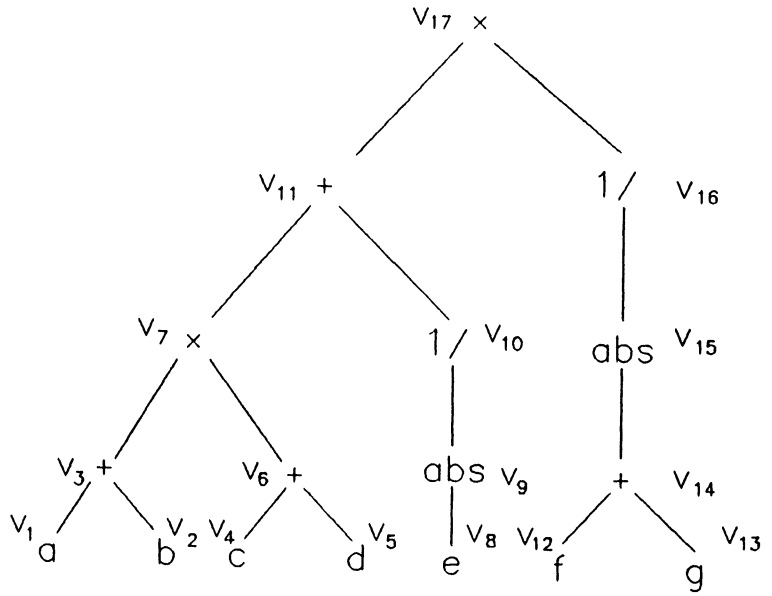


FIG. 1. *An expression tree for* $((a+b)\times(c+d)+1/\text{abs}(e))\times(1/\text{abs}(f+g))$.

However, we denote by F_v the subtree of F rooted at v. Also, let $l_v = l(F_v)$, $a_v = a(F_v)$, and $n_v = n(F_v)$.

A *sequential evaluation* $S = S_1, \cdots, S_n$ of a forest F is a linear ordering of the vertices of F such that if $i<j$ then S_i and S_j obey the precedence relation:

$$S_j \notin F_{S_i}.$$

For example, $S = 1\text{-}17$ (i is a shorthand for v_i and i-j stands for $i, i+1, \cdots, j$) is a legal evaluation of the tree of Fig. 1. S_i is the element of F computed (or loaded into memory) at *time slot i.* The *competion time* $c(S)$ of S is $|S| = n(F)$. Let S^1 and S^2 be two sequential evaluations of disjoint forests. The *concatenation* $S = S^1|S^2$ of S^1 and S^2 is simply a sequential evaluation S resulting from placing the first element of S^2 after the last element of S^1. Formally, for all $1 \leqq i \leqq |S^1|$, $S_i = S^1_i$, and for all $|S^1| < i \leqq |S^1|+|S^2|$, $S_i = S^2_{i-|S^1|}$.

A *parallel evaluation* $PQ = PQ_1, \cdots, PQ_{|PQ|}$ of a forest F is a sequence of pairs $PQ_i = (p_i, q_i)$ where $p_i \in L(F) \cup \{\text{NOP}\}$ (NOP stands for No OPeration) and $q_i \in A(F) \cup \{\text{NOP}\}$. If $p_i \neq \text{NOP}$ then p_i is loaded into memory at time slot i; otherwise i is an *empty load slot.* Similarly, if $q_i \neq \text{NOP}$ then q_i is computed at time slot i; otherwise i is an *empty arithmetic slot.* If both $p_i = \text{NOP}$ and $q_i = \text{NOP}$ then i is an *empty slot.* Thus, in the parallel evaluation PQ^1 of Fig. 2, time slots 7, 8, 10–13 are empty load slots while time slots 1, 2, 4 are empty arithmetic slots. For every $i<j$, PQ_i and PQ_j obey the *precedence relations*:

$$q_i \neq \text{NOP} \rightarrow p_j, \qquad q_j \notin F_{q_i}.$$

For a parallel evaluation PQ, the completion time $c(PQ) = |PQ|$.

The *width* $\rho_i(E)$ *of an evaluation* E (sequential or parallel) of a forest F *at time slot* i is the number of *useful intermediate results* (*registers*) of E which are available at time slot i. An intermediate result is *useful* at time slot i if the vertex representing it is either a root or a vertex that has already been computed at time slot i, and its parent has not been computed at time slot i. The *width* $\rho(E)$ *of an evaluation* E

i		1	2	3	4	5	6	7	8	9	10	11	12	13
PQ^1	P	1	2	4	5	8	12			13				
	Q			3		6	7	9	10	11	14	15	16	17
PQ^2	P	1	2	4	5	8	12	13						
	Q			3		6	7	9	10	11	14	15	16	17
PQ^3	P	1	2	4	5	8	12			13				
	Q				3	6	7	9	10	11	14	15	16	17
PQ^4	P	1	2	4	5	8	12	13						
	Q				3	6	7	9	10	11	14	15	16	17
PQ^5	P	8	1	2	4	5	12	13						
	Q			9	10	3	6	7	11	14	15	16	17	
PQ^6	P	8	12	13	1	2	4	5						
	Q		9	10	14	15	16	3	6	7	11	17		

FIG. 2. *Six parallel evaluations of the tree of Fig.* 1.

(sequential or parallel) of a forest F is the maximal value of $\rho_i(E)$ for all i. In Fig. 2, $\rho_i(PQ^1)=1$ for $i=1, 13$; $\rho_i(PQ^1)=2$ for $i=2, 3, 10\text{-}12$; and $\rho_i(PQ^1)=3$ for $4\leqq i\leqq 9$. Therefore, $\rho(PQ^1)=3$.

Any sequential evaluation may be viewed as a parallel evaluation of the same width by mapping every load operation p into a pair (p, NOP) and every arithmetic operation q into a pair (NOP, q). The other direction is also true: Given a parallel evaluation PQ, it can be transformed without increasing the width into a sequential evaluation by breaking every pair $PQ_i=(p_i, q_i)$ for which $p_i \neq \text{NOP}$ and $q_i \neq \text{NOP}$ into two pairs $(\text{NOP}, q_i)(p_i, \text{NOP})$ and then deleting all the NOPs from all pairs to yield a sequential evaluation. Transforming the parallel evaluation PQ^1 of Fig. 2 into a sequential, we obtain the sequential evaluation 1-6, 8, 7, 12, 9-11, 13-17.

Let E and E' be two (sequential or parallel) evaluations of a forest F. E and E' are *compatible* if the subsequences of load operations and the subsequences of arithmetic operations of E and E' are identical. For a given forest F let

$$\rho \min(E) = \min_{E'} [\rho(E') \mid E' \text{ is an evaluation of } F \text{ that is compatible with } E].$$

Also, let

$$\mu(F) = \min_{E} [\rho(E) \mid E \text{ is an evaluation of } F].$$

In other words, $\rho \min(E)$ is the minimal number of registers sufficient to compute F in an order compatible with E, and $\mu(F)$ is the minimal number of registers sufficient to compute F. In Fig. 2, $\rho \min(PQ^1)=3$, $\rho \min(PQ^5)=4$, while $\rho \min(PQ^6)=5$. For the tree T of Fig. 1, $\mu=3$. In the sequel, we will apply μ to a fixed tree T and use the following shorthand: $\mu=\mu(T)$ and $\mu_v=\mu(T_v)$.

2.3. The DP scheme. For the sequential case, the problem of finding an evaluation of minimum completion time is trivial, while the problem of finding an evaluation of minimum width is of interest. This problem has been solved efficiently by the *labelling*

algorithm of Sethi and Ullman [SU70]. Their algorithm labels the vertices of an expression tree with a minimal number of registers needed to compute the subtrees rooted at these vertices without store operations. In the following lemma we formalize their result without proof.

LEMMA 2.1. *Let T be a binary tree rooted at* root. *Then*

$$\mu = \begin{cases} 1 & \text{root } \textit{is the only vertex of } T, \\ \mu_u & \text{root } \textit{has one child } u, \\ \min(\max(\mu_u, \mu_w + 1), \max(\mu_u + 1, \mu_w)) & \text{root } \textit{has two children } u \textit{ and } w. \end{cases}$$ □

Before we start the discussion on parallel machines, we present a generic scheme derived from the dynamic programming algorithm of Aho and Johnson [AJ76]. For every vertex v of the input tree T the scheme computes $S_v[1], \cdots, S_v[R]$. If T_v cannot be computed with r registers, then $S_v[r] = \Lambda$, where Λ denotes the undefined evaluation sequence. Let $\Lambda | S = \Lambda$ and $S | \Lambda = \Lambda$ for every evaluation sequence S. Also, let Γ be a real-valued function defined over the set of evaluations (sequential or parallel) including the undefined evaluation sequence, so that $\Gamma(S) = \infty$ if and only if $S = \Lambda$.

THE DP SCHEME.
Visit the vertices of the tree in *postorder.* **For** every vertex v **do**;
if v is a leaf **then**
  for every $1 \leqq r \leqq R$ **do** $S_v[r] = (v)$
else if v has one child u **then**
  for every $1 \leqq r \leqq R$ **do** $S_v[r] = S_u[r]|(v)$
else /*v has two children */ **begin**
  let u, w be the children of v;
  $S_v[1] = \Lambda$;
  for every $2 \leqq r \leqq R$ **do**
L1: **if** $\Gamma(S_u[r]|S_w[r-1]|(v)) > \Gamma(S_w[r]|S_u[r-1]|(v))$ **then** $S_v[r] = S_w[r]|S_u[r-1]|(v)$
    else $S_v[r] = S_u[r]|S_w[r-1]|(v)$;
end;

All the decisions concerning the order of evaluation (line L1 in the DP scheme) depend on Γ. Given a specific function f, DP_f will stand for the algorithm obtained from the DP scheme by substituting f for Γ. In particular, consider DP_ρ and apply it to an input tree T. Let r be the smallest integer for which $S_{\mathrm{root}}[r] \neq \Lambda$. Then $r = \mu$ [AJ76]. For the tree of Fig. 1, DP_ρ yields $S_{\mathrm{root}}[r] = \Lambda$ for $r = 1, 2$ and $S_{\mathrm{root}}[r] = 1\text{-}17$ for $r \geqq 3$.

In the following theorem we state the correctness of the DP scheme. Notice that the proof (omitted here) is independent of Γ, and the only property that Γ must have is: $\Gamma(S) = \infty$ if and only if $S = \Lambda$.

THEOREM 2.2. *Let T be a binary tree and let v be a vertex of T.*
(i) *If $S_v[r] \neq \Lambda$, then $\rho(S_v[r]) \leqq r$.*
(ii) *For all $r \geqq \mu$, $S_{\mathrm{root}}[r] \neq \Lambda$.* □

In the following section a mapping P_r is developed that maps sequential to parallel evaluations whose width is bounded by r. $|P_r|$ is a function that, given a sequential computation S, finds the parallel completion time of $P_r(S)$. Assume that $\mathrm{DP}_{|P_r|}$ has been applied to a tree T. $P_R(S_{\mathrm{root}}[R])$ will be shown to be either undefined or use at most R registers. In addition it will be shown to be an efficient parallel evaluation that approximates the optimal one quite well.

2.4. From sequential to parallel evalution sequences. To develop the theory of parallel evaluations, let $P_r(S)$ be a shortest parallel evaluation of F compatible with

a sequential evaluation S that computes F such that $\rho(P_r(S)) \leqq r$. If no such evaluation exists, as $r < \rho \min(S)$, then $P_r(S) = \Lambda$. To fully describe $P_r(S)$ let L and A be the subsequences of S that consist of all the load and all the arithmetic operations of S, respectively.

THE P_r TRANSFORMATION.

```
i := 1;
j := 1;
PQ := ( );
while i ≦ |L| or j ≦ |A| do
  if |L| < i then for k := j to |A| do PQ := PQ, (NOP, A_k)
  else if |A| < j then for k := i to |L| do begin
    if ρ(PQ) = r then return (Λ);
    PQ := PQ, (L_k, NOP);
  end
  else if L_i ∈ F_{A_j} then
    if ρ(PQ) = r then return(Λ)
    else do begin
      PQ := PQ, (L_i, NOP);
      i := i+1;
    end
L1: else if ρ(PQ, (L_i, A_j)) ≦ r then begin
    PQ := PQ, (L_i, A_j);
    i := i+1;
    j := j+1;
  end
  else begin
    PQ := PQ, (NOP, A_j);
    j := j+1;
  end;
return(PQ);
```

Consider the evaluation $S = 1\text{-}17$ of the tree of Fig. 1. Then $P_1(S)$, $P_2(S) = \Lambda$, $P_3(S) = PQ^1$, and for $r \geqq 4$, $P_r(S) = PQ^2$ where PQ^1 and PQ^2 appear in Fig. 2. Given a sequential evaluation S, $P_r(S)$ is a shortest parallel evaluation compatible with S with the width less than or equal to r. We state that in the following lemma.

LEMMA 2.3. *Let PQ be a parallel evaluation of a forest F that is compatible with a sequential evaluation S. Then* $|P_{\rho(PQ)}(S)| \leqq |PQ|$. □

In § 3, additional properties of P_r are derived.

2.5. The analysis of the algorithm—An overview. The rest of the paper is devoted to the analysis of $\mathrm{DP}_{|P_r|}$. The DP scheme and the P_r transformation described above are all we need to implement $\mathrm{DP}_{|P_r|}$. However, in order to analyze the algorithm, several new concepts must be defined:

(1) In § 2.6, we define a class of *canonical* (sequential and parallel) evaluations. Informally, in canonical evaluations the arithmetic operations are scheduled as soon as possible after their operands have been computed. The evaluations produced by the DP scheme are always canonical. Also, we prove that an optimal evaluation can always be found among canonical evaluations. Therefore, in the rest of the paper, we restrict ourselves to considering canonical evaluations only.

(2) In § 3, several properties of parallel (canonical) evaluations are developed that enable us to define the *concatenation* of parallel evaluations. Intuitively, this concatenation operator allows to compute $|P_r(S_1|S_2)|$ from $|P_r(S_1)|$ and $|P_{r-1}(S_2)|$ in constant time. By employing the concatenation operator, the $DP_{|P_r|}$ algorithm is transformed into a dynamic programming algorithm (DPA) whose time complexity is linear. DPA is easier to analyze.

(3) Section 4 is devoted to the analysis of DPA. To this end, a subclass of canonical evaluations, called *normal form* (or simply *normal*) evaluations is defined. Evaluations produced by the DP scheme are always normal, and we prove that DPA produces shortest normal parallel evaluations.

(4) It turns out that certain trees do not have normal optimal evaluations. In § 5, we consider *bouncing* (nonnormal) evaluations and develop the conditions under which optimal evaluations necessarily bounce. Using these conditions, we prove that DPA is optimal when the number of registers is at most four and the number of vertices is at most 16.

(5) Finally, by performing elaborate analysis of DPA, we show in § 6 that the evaluations produced by DPA can be worse than the optimal by at most a factor of 1.091.

2.6. Canonical evaluations. Let S be a sequential evaluation of a forest F. Let $S_{j_1} \cdots S_{j_l}$ and $S_{i_1} \cdots S_{i_a}$ be the load and the arithmetic operations of S, respectively. For every $1 \leqq k \leqq l$, having S_{j_k} in a register enables a (possible empty) sequence B^k of arithmetic operations to be carried out that could not be computed without it. The *canonical evaluation derived from* S is $(S_{j_1})|B^1| \cdots |(S_{j_l})|B^l$. S is a *canonical evaluation* if it is identical to the canonical evaluation derived from S. Clearly, the canonical evaluation derived from S obeys the precedence constraints of F. Note that canonical evaluations are fully determined by the order among the load operations. Also, note that the arithmetic operations in a sequential evaluation S need not occupy the same positions in the canonical evaluation derived from it.

LEMMA 2.4. *Let $\bar{S}$ be the canonical evaluation derived from S. Then $\rho(\bar{S}) \leqq \rho(S)$.* □

Consider the evaluation $S = 1, 2, 4\text{-}6, 3, 7\text{-}17$ of the tree of Fig. 1 that has $\rho(S) = 4$. The canonical evaluation $\bar{S}$ derived from S is given by 1-17 and $\rho(\bar{S}) = 3$. Also, by Lemma 2.4, for every canonical evaluation S, $\rho \min(S) = \rho(S)$.

The notion of canonical evaluations may be extended to parallel evaluations. Let $PQ = (p_1, q_1) \cdots (p_{|PQ|}, q_{|PQ|})$ be a parallel evaluation of a forest F. Let $P = p_{j_1} \cdots p_{j_l}$ and $Q = q_{i_1} \cdots q_{i_a}$ be the non-NOP elements of PQ. As in the case of sequential evaluations, for every $1 \leqq k \leqq l$, having p_{j_k} in a register enables a (possibly empty) sequence B^k of arithmetic operations to be carried out that could not be computed without it. The concatenation $B = B^1|\cdots|B^l$ of $B^1, \cdots, B^l$ is a linear ordering of $A(F)$. Let $B = B_1, \cdots, B_a$. The *canonical parallel evaluation* $\overline{PQ}$ *derived from* PQ is obtained from PQ by replacing q_{i_k} with B_k for $1 \leqq k \leqq a$. We see that the arithmetic operations are permuted, but they occupy the same time slots (unlike the situation in the sequential case). The next lemma shows that canonical parallel evaluations fulfill the precedence relations.

LEMMA 2.5. *Let PQ be a parallel evaluation of a forest F. Then so is the canonical evaluation $\overline{PQ}$ derived from PQ.* □

In § 3, we show that for every parallel evaluation PQ compatible with a sequential evaluation S, there exists a parallel evaluation PQ', compatible with the canonical evaluation $\bar{S}$ derived from S, such that $c(PQ') \leqq c(PQ)$ and $\rho(PQ') \leqq \rho(PQ)$. Since PQ' is canonical, we will be interested only in canonical parallel evaluations.

3. Properties of parallel evaluations.

3.1. Basic properties. A binary arithmetic operation reduces by one the number of useful intermediate results in a parallel evaluation PQ while a unary arithmetic operation does not change it. Thus, if $p_i \neq \text{NOP}$ and q_i is a unary operation then $\rho_i(PQ) = \rho_{i-1}(PQ) + 1$ while if q_i is binary then $\rho_i(PQ) = \rho_{i-1}(PQ)$.

A time slot i is identified as a *load* (*arithmetic*) *hole* if in time slot i no load (arithmetic) operation is performed, while load (arithmetic) operations occur both before and after i. Consider the evaluation PQ^1 of Fig. 2. The time slots 7 and 8 are empty load holes while time slot 4 is an empty arithmetic hole.

Intuitively, the P_r procedure tries to execute the load and arithmetic operations of S as soon as possible without violating the precedence and the register constraints. Certain properties of the resultant parallel evaluations are derived from this observation. First, note that PQ has no empty slots. The following lemmas give additional properties of P_r.

LEMMA 3.1. *Let S be a sequential evaluation of a binary forest F and let $PQ = P_r(S) \neq \Lambda$. Then, all the load holes of PQ are occupied by unary arithmetic operations.* □

For example, consider evaluation PQ^1 of Fig. 2. The only empty load holes of PQ^1 are at time slots 7 and 8, and they are occupied by unary arithmetic operations that compute vertices 9 and 10 of the tree, respectively.

For sequential machines, ρ_i does not have to be a monotonic function of i; it increases as a result of several consecutive load operations and decreases when several binary arithmetic operations are performed in a row. This also may be true for parallel evaluations. However, if we concentrate on parallel evaluations generated by the P_r procedure, then ρ_i is a convex function of i. This is because if $\rho_i > \rho_{i+1}$ then a load operation could be scheduled at time slot $i+1$. Only when all the load operations have been scheduled may ρ_i decrease, but then it cannot increase again. Formally we get the following.

LEMMA 3.2. *Let the last load operation of an $PQ = P_r(S)$ be at time slot i. Then, $\rho_1(PQ) \leqq \cdots \leqq \rho_i(PQ) \geqq \rho_{i+1}(PQ) \geqq \cdots \rho_{|PQ|}(PQ)$.* □

COROLLARY 3.3. *Let S be a sequential evaluation of a binary forest F with k trees, and let $PQ = P_r(S)$. The number of binary arithmetic operations performed after the last load operation of PQ is $\rho(PQ) - k$. (However, there may be a few more unary operations.)* □

LEMMA 3.4. *Let $PQ = P_r(S)$, as above. Let the last arithmetic hole of PQ be at time slot i, and the first load hole at time slot j. Then, $j > i$.* □

The following lemma shows that the order of evaluation S of F determines the completion time of $P_r(S)$, independently of r (as long as $r \geqq \rho \min(S)$).

LEMMA 3.5. *Let S be a sequential evaluation of a forest F, and let $r_1, r_2 \geqq \rho \min(S)$. Then $|P_{r_1}(S)| = |P_{r_2}(S)|$.* □

Consider the tree T of Fig. 1. Computing T in the natural order $S = 1\text{-}17$, results in $P_3(S) = PQ^1$ and $P_4(S) = PQ^2$ where PQ^1 and PQ^2 appear in Fig. 2. Note that $c(PQ^1) = c(PQ^2) = 13$.

COROLLARY 3.6. *Let PQ be a parallel evaluation compatible with a sequential evaluation S. Let $\bar{S}$ be the canonical evaluation derived from S. Then*

(i) $|P_{\rho(\bar{S})}(\bar{S}) \leqq |PQ|$.

(ii) $\rho(P_{\rho(\bar{S})}(\bar{S})) \leqq \rho(PQ)$. □

Let S be a sequential (canonical) evaluation of a binary forest F. We define a (canonical) *ρ-efficient* evaluation PQ compatible with S to be $P_{\rho(S)}(S)$. As a consequence of Corollary 3.6, in the rest of the paper we consider canonical ρ-efficient evaluations only.

3.2. Consecutive evaluations. Let S be a sequential evaluation of a binary forest F and let $PQ = P_r(S)$. An evaluation PQ' that is identical to PQ, except that it has no arithmetic holes, is called *r-consecutive evaluation.* By Lemmas 3.2 and 3.4, we can iteratively delay all the arithmetic operations prior to the last arithmetic hole and leave all other operations in their original place to transform every parallel evaluation resulted from the P_r procedure into an r-consecutive evaluation of the same width and completion time. Let $P_r^*(S)$ denote the r-consecutive evaluation obtained from $P_r(S)$ in this fashion. Then, $\rho(P_r^*(S)) = \rho(P_r(S))$, and $|P_r^*(S)| = |P_r(S)|$. For example, PQ^3 of Fig. 2 is a 3-consecutive evaluation that corresponds to $PQ^1 = P_3(S)$ where $S = 1\text{-}17$. Also, PQ^4 is a 4-consecutive evaluation that corresponds to $PQ^2 = P_4(S)$. Since for all S and for every $r \geqq \rho \min(S)$, $|P_r^*(S)| = |P_r(S)|$, the properties of the algorithm $\mathrm{DP}_{|P_r^*|}$ are identical to those of $\mathrm{DP}_{|P_r|}$.

Given a parallel evaluation PQ and a sufficiently large r (e.g., $r > |PQ|$), an r-consecutive evaluation PQ' compatible with PQ will have no load holes. Such an evaluation PQ' is called ∞-*consecutive*, or simply *consecutive.* Thus, PQ^4 shown in Fig. 2 is the consecutive evaluation of the tree of Fig. 1 compatible with $S = 1\text{-}17$.

In line L1, the $\mathrm{DP}_{|P_r^*|}$ algorithm computes $|P_r^*(S_u[r]|S_w[r-1]|(v))|$ to make a decision concerning the order of evaluation. It turns out that $|P_r^*(S_u[r]|S_w[r-1]|(v))|$ can be computed directly from $|P_r^*(S_u[r])|$ and $|P_{r-1}^*(S_w[r-1])|$ rather than by applying P_r^* to $S_u[r]|S_w[r-1]|(v)$. Since, by Lemma 3.5, the cost of r-consecutive evaluations is independent of how large r can be, we can use P_∞^* rather than P_r^* and compute $|P_\infty^*(S_u[r]|S_w[r-1]|(v))|$ from $|P_\infty^*(S_u[r])|$ and $|P_\infty^*(S_w[r-1])|$. In the sequel, some properties of consecutive evaluations are developed. Then, using the concatenation of consecutive evaluations that will be defined, we develop a linear algorithm called DPA derived from the $\mathrm{DP}_{|P_\infty^*|}$ algorithm. DPA is easier to analyze than $\mathrm{DP}_{|P_r^*|}$.

Let F be a forest of k trees. Let PQ be a consecutive evaluation of F. A consecutive evaluation PQ is presented pictorially in Fig. 3(a). It is characterized by the following parameters:

(1) $c = |PQ|$ is the completion time of PQ;

(2) h is the *head*—the number of load operations that take place before the first arithmetic operation. Note that h is equal to the number of empty arithmetic slots in PQ. Using $a = a(F)$ we get

$$h = c - a; \tag{1}$$

(3) t is the *tail*—the number of arithmetic operations that take place after the last load operation. Note that t is equal to the number of empty load slots in PQ. Using $l = l(F)$ we get

$$t = c - l. \tag{2}$$

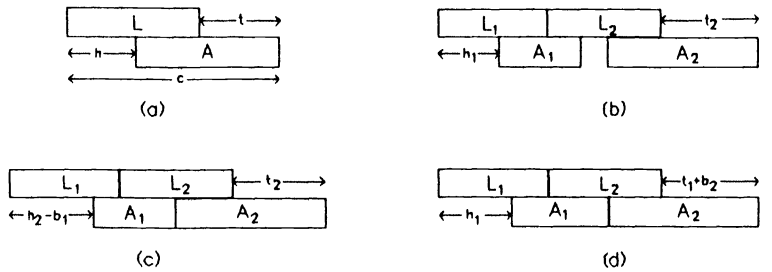


FIG. 3. *Pictorial presentation and concatenation of consecutive evaluations.*

The c, h, t parameters are a convenient, partial characterization of consecutive evaluations. To completely characterize consecutive evaluations that compute a forest F, both S and F are needed. In the sequel we present the c, h, t parameters of a consecutive evaluation PQ as a triple $\langle c, h, t\rangle$. For example, the consecutive evaluation of a single load operation is characterized by $\langle 1, 1, 0\rangle$, while a single arithmetic operation is characterized by $\langle 1, 0, 1\rangle$. Also, we assume that if $PQ = \Lambda$ then $c = \infty$, $h = \infty$, and $t = \infty$.

Since we assumed that there are no arithmetic leaves in expression trees, at least one load operation must be performed before a first arithmetic operation can be carried out. Therefore, in the rest of the paper we assume that $h \geqq 1$.

Below some key properties of the c, h, t parameters are derived. Let PQ be a consecutive evaluation which computes a forest F with k trees.

COROLLARY 3.7. $h(PQ) \geqq \rho(PQ)$. □

COROLLARY 3.8. $t(PQ) \geqq \rho(PQ) - k$. □

Another important concept is that of a *balance*. The *balance* of a consecutive evaluation PQ is defined by

$$b = t - h. \tag{3}$$

Substituting (1) and (2) into (3) we get $b = a - l$. Therefore, all forests with l load operations and a arithmetic operations have the same balance.

Let T be a binary tree. Since the number of arithmetic operations in T is at least one less than the number of load operations $a - l \geqq -1$. Therefore, for every binary tree T, $b \geqq u - 1$. Moreover, if T is a *full binary tree* (all the internal vertices of T have exactly two children) then $a = l - 1$, and $b = -1$.

Let PQ^1 and PQ^2 be two consecutive evaluations compatible with sequential evaluations S^1 and S^2 of F^1 and F^2, respectively. The *concatenation* $PQ^{1,2} = PQ^1 | PQ^2$ of PQ^1 *and* PQ^2 is defined as the consecutive evaluation of $F^1 \cup F^2$ compatible with $S^1 | S^2$. $PQ^{1,2}$ is obtained by shifting PQ^2 toward PQ^1, until either the load operations meet, as in Fig. 3(b) (and then shifting the arithmetic operations of PQ^1 to the right until they meet the arithmetic operations of PQ^2; see Fig. 3(c)), or until the arithmetic operations meet (and then shifting the load operations of PQ^2 to the left until they meet the load operations of PQ^1; see Fig. 3(d)). It turns out that to compute the c, h, and t parameters of $PQ^{1,2}$ only the c, h and t parameters of PQ^1 and PQ^2 are needed.

We use the concatenation operator defined above to present the following linear version of the $DP_{|P^*_\infty|}$ algorithm (DPA). Note that all the decisions in the DPA algorithm (line L1 below) are based solely on the c, h, and t parameters of consecutive evaluations.

THE DPA ALGORITHM.

Visit the vertices of the tree in *postorder*. **For** every vertex v **do**;
if v is a leaf **then**
  for every $1 \leqq r \leqq R$ **do** $PQ_v[r] = ((v, \text{NOP}))$
else if v has one child u **then**
  for every $1 \leqq r \leqq R$ **do** $PQ_v[r] = PQ_u[r] | ((\text{NOP}, v))$
else /* v has two children */ **begin**
  let u, w be the children of v;
  $PQ_v[1] = \Lambda$;
  for every $2 \leqq r \leqq R$ **do**
L1: **if** $c(PQ_u[r] | PQ_w[r-1] | ((\text{NOP}, v))) > c(PQ_w[r] | PQ_u[r-1] | ((\text{NOP}, v)))$
    then $PQ_v[r] = PQ_w[r] | PQ_u[r-1] | ((\text{NOP}, v))$
    else $PQ_v[r] = PQ_u[r] | PQ_w[r-1] | ((\text{NOP}, v))$
end;

It can be shown that the concatenation of two consecutive evaluations can be computed

in a constant time. Therefore, the complexity of DPA is linear in nR since every vertex is visited only once, and the time spent there is bounded by $O(R)$. Moreover, for large values of R ($R \geqq \log n$), DPA can be implemented in such a way that its complexity is bounded by $O(n \min(\log n, R))$. An example that shows how DPA processes the tree of Fig. 1 is shown in the Appendix.

4. Properties of DPA.

4.1. Properties of the concatenation operation. Below, certain properties of the concatenation of parallel consecutive evaluations are derived. The proofs of Lemmas 4.1–4.3 are essentially the same as those of [BJPR85] and are omitted here.

Let PQ^i, $i = 1, 2, \cdots$ be arbitrary consecutive evaluations with $\langle c_i, h_i, b_i \rangle$ parameters. Also, let $b_i = t_i - h_i$. Since $b = a - l$, we get

$$b_{1,2} = b_1 + b_2. \tag{4}$$

LEMMA 4.1. *Let* $PQ^{1,2} = PQ^1 | PQ^2$. *Then*

$$h_{1,2} = \max(h_1, h_2 - b_1). \tag{5}$$ □

The following lemma shows that the concatenation operator is associative. This will allow us to omit parenthesis where convenient.

LEMMA 4.2. (the associativity property). $(PQ^1 | PQ^2) | PQ^3 = PQ^1 | (PQ^2 | PQ^3)$. □

Lemma 4.3 below shows that if some subevaluation in the middle of a larger evaluation is locally improved, then the total cost may only decrease.

LEMMA 4.3 (the monotonicity property). *If* $b_2 = b_4$ *and* $h_2 \leqq h_4$ *then* $h_{1,2,3} \leqq h_{1,4,3}$. □

4.2. Normal form evaluations. A sequential canonical evaluation S of a tree T may be viewed as a visiting sequence of the vertices of T. S is a *normal form evaluation* (or simply *normal evaluation*) if for every $v \in T$ all the vertices of T_v are visited in a row such that v is the last one to be visited. A parallel evaluation is *normal* if it is compatible with a sequential normal evaluation S. For example, in Fig. 2, PQ^1 and PQ^5 are normal, while PQ^6 is not normal. Note that normal evaluations are always canonical. Also note that for all v and r, $PQ_v[r]$ is either normal or undefined.

THEOREM 4.4. $PQ_{root}[r]$ *has the shortest completion time among all normal evaluations that use at most* r *registers.*

Proof. The proof is by induction on $n = |T|$.

Basis. For a tree with one vertex the theorem trivially holds.

Induction hypothesis. Assume that for all trees with less than n vertices the theorem holds.

Induction step. Let T be a tree with n vertices rooted at *root*.

Case 1. *Root* has one immediate descendant u. By the induction hypothesis, $PQ_u[r]$ has the smallest completion time among all normal evaluations of T_u that use at most r registers. Therefore, so does $PQ_{\text{root}}[r] = PQ_u[r] | ((\text{NOP}, v))$.

Case 2. *Root* has two immediate descendants u and w. Consider the set of all normal evaluations of T that use at most r registers. They can be divided into two disjoint subsets $S^{u,w}$ and $S^{w,u}$, depending on the order in which they compute T. The evaluations of $S^{u,w}$ compute T_u first, while the evaluations of $S^{w,u}$ compute T_w first. Let $PQ = PQ^u | PQ^w | ((\text{NOP}, v))$ be an arbitrary evaluation in $S^{u,w}$ that uses at most r registers. PQ^u cannot use more than r registers. Therefore, by the induction hypothesis, $c(PQ_u[r]) \leqq c(PQ^u)$. Similarly, PQ^w cannot use more than $r-1$ registers. Again, by the induction hypothesis, $c(PQ_w[r-1]) \leqq c(PQ^w)$. By Lemma 4.3,

$$c(PQ_u[r] | PQ_w[r-1]) \leqq c(PQ_u[r] | PQ^w) \leqq c(PQ^u | PQ^w).$$

Therefore, $PQ_u[r]|PQ_w[r-1]|((\text{NOP}, v))$ has the smallest cost among all the evaluations in $S^{u,w}$. Similarly all the evaluations in $S^{w,u}$ have their completion time bounded below by $c(PQ_w[r]|PQ_u[r-1]|((\text{NOP}, v)))$. Since $c(PQ_{\text{root}}[r]) = \min(c(PQ_u[r]|PQ_w[r-1]|((\text{NOP}, v))), c(PQ_w[r]|PQ_u[r-1]|((\text{NOP}, v))))$, the theorem follows. □

The following lemma proves that $c(PQ_{\text{root}}[r])$ is a nonincreasing function of r, i.e., by increasing the number of registers it is possible only to reduce the completion time of the evaluation produced by DPA. Using (1) we can prove this result by showing that $h(PQ_{\text{root}}[r])$ never increases when r increases.

LEMMA 4.5. *Let T be a binary tree rooted at root. For every $r \geqq 2$, $h(PQ_{root}[r]) \leqq h(PQ_{root}[r-1])$.*

Proof. The proof is by induction on $n = |T|$.

Bases. For a tree with one vertex and for $r \geqq 2$, $h(PQ_{root}[r])$, $h(PQ_{root}[r-1]) = 1$. Therefore, the lemma trivially holds.

Induction hypothesis. Assume that for all trees with less than n vertices and for every $r \geqq 2$, $h(PQ_{root}[r]) \leqq h(PQ_{root}[r-1])$.

Induction step. Let T be a tree with n vertices.

Case 1. *Root* has one immediate descendant u. For every $r \geqq 1$, $h(PQ_{root}[r]) = h(PQ_u[r]|((\text{NOP}, root))$. Therefore, for every $r \geqq 2$, $h(PQ_{root}[r]) = h(PQ_u[r])$, and $h(PQ_{root}[r-1]) = h(PQ_u[r-1])$. By the induction hypothesis, for every $r \geqq 2$, $h(PQ_u[r]) \leqq h(PQ_u[r-1])$. Therefore, for every $r \geqq 2$, $h(PQ_{root}[r]) \leqq h(PQ_{root}[r-1])$.

Case 2. *Root* has two immediate descendants u and w. For $r = 2$, the lemma holds as $h(PQ_{root}[2]) \leqq h(PQ_{root}[1]) = \infty$. Therefore, assume that $r \geqq 3$. By the induction hypothesis,

$$\text{for all } r \geqq 2: \quad h(PQ_u[r]) \leqq h(PQ_u[r-1])$$

and

$$\text{for all } r \geqq 2: \quad h(PQ_w[r]) \leqq h(PQ_w[r-1]).$$

Therefore, by (5),

$$\begin{aligned} h(PQ_u[r-1]|PQ_w[r-2]|((\text{NOP}, root))) &= \max(h(PQ_u[r-1]), h(PQ_w[r-2]) - b_u) \\ &\geqq \max(h(PQ_u[r]), h(PQ_w[r-1]) - b_u) \\ &= h(PQ_u[r]|PQ_w[r-1]|((\text{NOP}, root))). \end{aligned}$$

Similarly,

$$h(PQ_w[r-1]|PQ_u[r-2]|((\text{NOP}, root))) \geqq h(PQ_w[r]|PQ_u[r-1]|((\text{NOP}, root))).$$

Substitution of these inequalities in the condition of line L1 of DPA yields

$$\begin{aligned} h(PQ_{root}[r]) &= \min(h(PQ_u[r]|PQ_w[r-1]|((\text{NOP}, root))), \\ &\quad h(PQ_w[r]|PQ_u[r-1]|((\text{NOP}, root)))) \\ &\leqq \min(h(PQ_u[r-1]|PQ_w[r-2]|((\text{NOP}, root))), \\ &\quad h(PQ_w[r-1]|PQ_u[r-2]|((\text{NOP}, root)))) \\ &= h(PQ_{root}[r-1]). \end{aligned}$$

□

The following lemma proves that DPA finds the minimal completion time (optimal) evaluation for a *full binary tree*—a tree in which all the internal vertices have exactly two children.

LEMMA 4.6. *Let T be a full binary tree rooted at root. For every $r \geqq \mu$, $h(PQ_{root}[r]) = \mu$.*

Proof. Consider the consecutive evaluation $PQ_{root}[\mu]$ of T. By Theorem 2.2, $PQ_{root}[\mu] \neq \Lambda$, and $\rho(PQ_{root}[\mu]) \leqq \mu$. Therefore, $\rho(PQ_{root}[\mu]) = \mu$. Let i and j be the time slots in which the first arithmetic operation and the last load operation of $PQ_{root}[\mu]$ are performed. Clearly, $\rho_{i-1}(PQ_{root}[\mu]) = h(PQ_{root}[\mu])$. By Lemma 3.3,, $\rho(PQ_{root}[\mu]) = \rho_j(PQ_{root}[\mu])$. But, since all the arithmetic operations of T are binary, for all $i \leqq k \leqq j$, $\rho_k(PQ_{root}[\mu]) = \rho_{i-1}(PQ_{root}[\mu])$. Therefore, $\mu = h(PQ_{root}[\mu])$.

For $r > \mu$, by Lemma 4.5, $h(PQ_{root}[r]) \leqq h(PQ_{root}[\mu])$. If for some r, $h(PQ_{root}[r]) < h(PQ_{root}[\mu])$ then the same argument shows that $\rho(PQ_{root}[r]) < \mu$, contradicting the definition of μ. Therefore, for every $r \geqq \mu$, $h(PQ_{root}[\mu]) = \mu$. □

LEMMA 4.7. *Let v be a binary vertex with two children u and w. If $h(PQ_v[\mu_v]) > h(PQ_u[\mu_u]), h(PQ_w[\mu_w])$, then T_v is a full binary tree.*

Proof. Without loss of generality, we assume that $\mu_u \geqq \mu_w$.

Case 1. $\mu_u = \mu_w$. By Lemma 2.1, $\mu_v = \mu_u + 1 = \mu_w + 1$. By line L1 of DPA,

$$h(PQ_v[\mu_v]) \leqq h(PQ_\mu[\mu_v] | PQ_w[\mu_v - 1]) = \max(h(PQ_u[\mu_v]), h(PQ_w[\mu_v - 1]) - b_u). \tag{6}$$

By Lemma 4.5, $h(PQ_u[\mu_u + 1]) \leqq h(PQ_u[\mu_u])$. Therefore, $h(PQ_v[\mu_v]) > h(PQ_u[\mu_v])$. Since $\mu_v - 1 = \mu_w$, $h(PQ_v[\mu_v]) > h(PQ_w[\mu_v - 1])$. Therefore, $b_u = -1$. Similarly, $b_w = -1$. Thus, T_v is a full binary tree.

Case 2. $\mu_v > \mu_w$.

We claim that this case cannot happen. Consider (6) again. By Lemma 2.1, $\mu_v = \mu_u$. Therefore, $h(PQ_v[\mu_v]) > h(PQ_u[\mu_v])$. Also, since $\mu_v - 1 \geqq \mu_w$, by Lemma 4.5, $h(PQ_v[\mu_v]) > h(PQ_w[\mu_v - 1])$. Therefore, $b_u = -1$, and T_u is a full binary tree. By Lemma 4.6, $h(PQ_u[\mu_u]) = \mu_u$. Also, by Theorem 2.2(i), $\rho(PQ_w[\mu_v - 1]) \leqq \mu_v - 1 = \mu_u - 1$, and by Corollary 3.7, $h(PQ_w[\mu_v - 1]) \leqq \mu_u - 1$. Therefore, by (6), $h(PQ_v[\mu_v]) \leqq \mu_u$, which would imply $h(PQ_v[\mu_v]) \leqq h(PQ_u[\mu_u])$ contradicting the assumption of the lemma. □

Let *bop* and *uop* be the number of binary and unary arithmetic operations of T, respectively. The following lemma establishes an upper bound on μ as a function of *bop*.

LEMMA 4.8. $bop \geqq 2^{\mu-1} - 1$.

Proof. The proof is by induction on $n = |T|$.

Basis. For a tree with one vertex $bop = 0$, and $\mu = 1$, and the lemma holds.

Induction hypothesis. Assume that for all trees with less than n vertices the lemma holds.

Induction step. Let T be a tree with n vertices rooted at *root*.

Case 1. *Root* has one immediate descendant u. By the induction hypothesis,

$$bop_u \geqq 2^{\mu_u - 1} - 1.$$

Since $bop = bop_u$ and by Lemma 2.1, $\mu = \mu_u$, the lemma holds.

Case 2. *Root* has two immediate descendants u and w. By the induction hypothesis,

$$bop_u \geqq 2^{\mu_u - 1} - 1 \quad \text{and} \quad bop_w \geqq 2^{\mu_w - 1} - 1.$$

Without loss of generality we assume that $\mu_u \geqq \mu_w$. If $\mu_u > \mu_w$ then, by Lemma 2.1, $\mu = \mu_u$. Therefore, $bop > bop_u \geqq 2^{\mu_u - 1} - 1 = 2^{\mu - 1} - 1$. If $\mu_u = \mu_w$ then, by Lemma 2.1, $\mu = \mu_u + 1$. Since $bop = bop_u + bop_w + 1$, we apply the induction hypothesis twice to get $bop \geqq 2^{\mu_u - 1} + 2^{\mu_w - 1} - 1$. By $\mu_u = \mu_w = \mu - 1$, we get $bop \geqq 2^{\mu_u} - 1 = 2^{\mu - 1} - 1$. □

COROLLARY 4.9. $h(PQ_{root}[\mu]) \leqq \lfloor \log(n - uop + 1) \rfloor$.

Proof. For arbitrary binary trees $l = (n - uop + 1)/2 = bop + 1$. Thus, by Lemma 4.8,

$$\lfloor \log (n - uop + 1) \rfloor = \lfloor \log 2(bop + 1) \rfloor \geqq \mu.$$

By Corollary 3.7, $h(PQ_{root}[\mu]) \leqq \mu$, and the statement is proved. □

COROLLARY 4.10. $a \geqq 2^{h(PQ_{root}[\mu])-1} + uop - 1$.

Proof. Since $a = bop + uop$, by Lemma 4.8, $a \geqq 2^{\mu-1} + uop - 1$. By Corollary 3.7, $h(PQ_{root}[\mu]) \leqq \mu$, and the statement is proved. □

5. Bouncing evaluations. In § 4 we have discussed the properties of normal evaluations, and have proved that DPA finds a normal parallel evaluation of minimum completion time. In this section we discuss arbitrary canonical ρ-efficient evaluations. By Corollary 3.6, no other parallel evaluations need be considered. In the sequel, whenever we mention sequential or parallel evaluations, we mean canonical sequential evaluations or canonical parallel ρ-efficient evaluations. Also, in the rest of the paper we use the following shorthand: $h(S) = h(P^*_\infty(S))$, $t(S) = t(P^*_\infty(S))$, and $b(S) = b(P^*_\infty(S))$. We start with some properties of ρ.

5.1. Properties of ρ. Let S be a sequential evaluation of a binary tree T, and let S^1 and S^2 be subevaluations of S such that $S = S^1 | S^2$. In the following lemmas we relate $\rho(S^1)$ and $\rho(S^2)$ to $\rho(S)$.

LEMMA 5.1. *Let $S = S^1 | S^2$, and let k be the number of useful results computed by S^1.*

(i) $\rho(S^1), \rho(S^2) \leqq \rho(S)$.

(ii) *If the results computed by S^1 are not used in S^2 then* $\rho(S) = \max(\rho(S^1), \rho(S^2) + k)$.

Proof. (i) $\rho(S^1) = \max_{1 \leqq i \leqq |S^1|} \rho_i(S) \leqq \rho(S) = \rho(S)$. Similarly, $\rho(S^2) \leqq \max_{|S^1| < i \leqq |S|} \rho_i(S) \leqq \rho(S)$. Therefore, (i) holds.

(ii) During the computation of S^2, k registers hold the results computed by S^1. Since they are not used in S^2, $\max_{|S^1| < i \leqq |S|} \rho_i(S) = \rho(S^2) + k$, and (ii) follows. □

LEMMA 5.2. *Let $S = S^1 | S^2$, and let k be the number of useful results computed by S^1 (some of them may be used in S^2). Then $h(S^2) \leqq \rho(S) - k$.*

Proof. Let $PQ = P_{\rho(S)}(S)$. Consider the time slot i of PQ at which the last arithmetic operation of S^1 is performed. Let the number of the load operations of S^2 that have been performed no later than i be h', and let the number of the arithmetic operations of S^2 be a.

CLAIM. $h' \geqq h(S^2)$.

Proof of Claim. Assume, by contradiction, that $h' < h(S^2)$. Consider the parallel evaluation PQ^2 that is compatible with S^2 in which all the operations are executed exactly in the same time slots as in PQ except that the first h' loads are consecutive at time slot $1, 2, \cdots, h'$ of PQ^2. PQ^2 is a legal parallel evaluation, and $|PQ^2| = h' + a < h(S^2) + a = |P^*_\infty(S^2)|$. This contradicts Lemma 3.5, and the claim follows. □

Now consider PQ again. At the end of slot i, k registers hold the results of S^1, and h' registers are loaded with elements needed to compute S^2. Therefore, $h' + k \leqq \rho(PQ)$. Since PQ is a ρ-efficient evaluation $\rho(PQ) = \rho(S)$, and the lemma follows. □

5.2. Properties of bouncing sequential evaluations. Let S be a sequential evaluation of a binary tree T. If S is not a normal evaluation then there exists at least one subtree T_v of T such that S starts to evaluate it, and without finishing the computation *bounces* to a disjoint subtree T_u of T. In this case we say that S *bounces* at v, and v is a *bouncing operation.* An operation computing $v \in T$ is *normal* in S if for every $u \in T_v$, the vertices of T_u are visited by S in a row and u is the last one to be visited. Note that if $v \in T$ is a bouncing operation of S then all the ancestors of v in T are not normal, but they are not necessarily bouncing. For example, consider the sequential evaluation $S = 8\text{–}10$,

12-16, 1-7, 11, 17 of the tree of Fig. 1 that is compatible with PQ^6 of Fig. 2. In S, v_7, v_{10}, and v_{16} are normal, v_{11} is bouncing, while v_{17} is neither normal nor bouncing. Also, note that all the operations that compute the leaves of a tree (load operations) are normal. If v has one child u then v is executed immediately after u in every canonical evaluation of T. In this case v is a normal operation if and only if u is normal. On the other hand, if v has two children u and w then there exist canonical evaluations in which u and w are normal while v is bouncing.

Let v be a bouncing operation of a sequential evaluation S whose children are normal. Then v has two children u and w. Assume that u is computed first, then v is executed immediately after w. Projecting S on the vertices of T_u and T_w yields subevaluations S^1 and S^2 of S, respectively. Let the vertices of T computed between S^1 and S^2, belong to some subforest F^v of T. We say that T_u, T_w and F^v *are induced by* v and S. Note that the operations of F^v may use some of the results that are computed in S before T_u, but since S is a canonical evaluation, for each such operation at least one of its two operands must be computed after T_u. In the following lemma, certain properties of T_u, T_w, and F^v are derived.

LEMMA 5.3. *Let S be a sequential evaluation of a tree T. Let v be a bouncing operation of S whose children are normal, and let T_u, T_w and F^v be induced by v and S. Let S^u be the projection of S on the vertices of T_u. Also, let m and k be the number of useful results that are computed by S before T_u and T_w, respectively. Let S' be a sequential evaluation of T that is identical to S except that F^v is computed after v. Let S'' be the sequential evaluation of T that is identical to S except that F^v is computed before T_u.*

(i) *If $k \geqq m+1$, then $\rho(S') \leqq \rho(S)$.*

(ii) *If $k < m+1$, then $\rho(S'') \leqq \rho(S)$.*

(iii) *If $\rho(S^u) \leqq \rho(S) - k + 1$, then $\rho(S'') \leqq \rho(S)$.*

Proof. (i) $\rho_i(S')$ may be greater than $\rho_i(S)$ only in time slots i in which T_w, v, and F^v are computed. Since T_u, T_w, and v construct a single subtree of T, during the computation of F^v the number of useful results in S and in S' is identical. Since $k \geqq m+1$, during the computation of T_w and v the number of useful results in S' is not greater than that in S.

(ii) Consider the part of S'' that computes F^v and T_u. During the computation of F^v the width cannot increase since T_u is computed after F^v in S'', while in S it is computed before F^v, and its result occupies one additional register. When T_u is computed in S, m registers hold previously computed results, while when T_u is computed in S'', only $k-1$ registers hold previously computed results. Since $m > k-1$, (ii) holds.

(iii) As in (ii), during the computation of F^v the width cannot increase since T_u is computed after F^v in S'', while in S it is computed before F^v, and its result occupies one additional register. When T_u is computed in S'', $k-1$ registers hold previously computed results. By the assumption, $\rho(S^u) \leqq \rho(S) - k + 1$. Therefore, by Lemma 5.1, (ii), (iii) holds. □

5.3. Properties of optimal parallel evaluations. The notion of bouncing evaluations can be extended to parallel evaluations. A parallel evaluation PQ of T is *bouncing* if it is compatible with a sequential bouncing evaluation S of T. This definition is proper since we are dealing with canonical evaluations only, and there is exactly one sequential canonical evaluation compatible with a parallel canonical evaluation.

Next let us develop some properties of optimal evaluations and derive the conditions under which a optiml evaluation may not be normal. Without loss of generality we consider only optimal bouncing ρ-efficient evaluations with a *minimal number of*

bouncing operations. Let PQ be such an optimal evaluation of a binary tree T compatible with a sequential evaluation S. Let T_v be a subtree of T normal in S. Let S^v be the projection of S on the vertices of T_v. Consider the parallel evaluation $PQ_v[\rho(S^v)]$ generated by DPA for T_v. By Theorem 2.2, $\rho(PQ_v[\rho(S^v)]) \leqq \rho(S^v)$, and by Theorem 4.4, $|PQ_v[\rho(S^v)]| \leqq |P^*_\infty(S^v)|$. Therefore, we can assume that for every $v \in T$, if v is normal in S, then T_v is computed in S in the same order of evaluation as the one generated by DPA.

To analyze bouncing evaluations, we permute them so as to get normal evaluations without increasing the number of registers used, and then assess the ratio between the completion time of the resultant normal evaluation and the completion time of the optimal bouncing evaluation.

LEMMA 5.4. *Let PQ be an optimal evaluation of a binary tree T compatible with a sequential evaluation S. Let v be a bouncing operation of S whose children are normal, and let T_u, T_w, and F^v be induced by v and S. Also, let m and k be the number of useful results computed by S before T_u and T_w, respectively, and assume that $k \geqq m+1$. Let S^1, S^2, and S^3 be the projection of S on the vertices of T_u, T_w, and F^v, respectively, and let PQ^1, PQ^2, and PQ^3 be the corresponding compatible consecutive evaluations.*

(i) $b_1, b_3 \geqq 1$ *and* $h_1 < h_3 < h_2$.

(ii) *Let $\bar{S}$ be the prefix of S computed before F^v. Then* $\rho(\bar{S}) \leqq \rho(S) - 1$.

(iii) $\rho(S^2) \leqq \rho(S) - 1$, $\rho(S^3) \leqq \rho(S) - 1$.

Proof. Let S' be a sequential evaluation of T identical to S except that F^v is computed after v. Since v is a normal operation in S', the number of bouncing operations of S' is strictly smaller than that of S. Let PQ' be a ρ-efficient evaluation compatible with S'. By Lemma 5.3(i), $\rho(S') \leqq \rho(S)$. We will use the fact that $|PQ'| > |PQ|$, since otherwise PQ could not have been an optimal evaluation with a minimal number of bouncing operations. Let $PQ^4 = PQ^2|((\text{NOP}, v))$. Therefore, $h_4 = h_2$ and $b_4 = b_2 + 1$.

Proof of (i) We compare $c_{1,3,4}$ with $c_{1,4,3}$. By Lemma 3.5, $|P^*_\infty(S)| = |PQ|$ and $|P^*_\infty(S')| = |PQ'|$. By Lemma 4.3, if $c_{1,4,3} \leqq c_{1,3,4}$ then $|P^*_\infty(S')| \leqq |P^*_\infty(S)|$ and therefore, $|PQ'| \leqq |PQ|$. Since this is impossible we assume that $c_{1,3,4} < c_{1,4,3}$, i.e., $h_{1,3,4} < h_{1,4,3}$. By (5), this leads to $h_{3,4} < h_{4,3}$.

CLAIM 1. $h_3 < h_4$.

Proof of Claim 1. Since T_w is a binary tree, $b_2 \geqq -1$. Therefore, $b_4 \geqq 0$, and by (5),

$$h_{4,3} = \max(h_4, h_3 - b_4) \leqq \max(h_4, h_3).$$

Also, by (5),

$$h_{3,4} = \max(h_3, h_4 - b_3).$$

If $h_3 \geqq h_4$ then

$$h_{4,3} \leqq h_3 \leqq \max(h_3, h_4 - b_3) = h_{3,4},$$

contradicting $h_{3,4} < h_{4,3}$, and Claim 1 is proved. □

CLAIM 2. $b_3 \geqq 1$.

Proof of Claim 2. If $b_3 \leqq 0$ then $h_{3,4} = \max(h_3, h_4 - b_3) \geqq h_4$. By Claim 1, $h_{4,3} \leqq \max(h_4, h_3) = h_4$. This leads to a contradiction to $h_{3,4} < h_{4,3}$. Therefore, Claim 2 is proved. □

CLAIM 3. $b_1 \geqq 1$ and $h_1 < h_3$.

Proof of Claim 3. The proof is by case analysis.

Case 3.1. $\rho(S^1) > \rho(S) - k + 1$.

We show that this leads to a contradiction. By Corollary 3.8, $t_1 \geqq \rho(S^1) - 1 \geqq \rho(S) - k + 1$. Let $\hat{S}$ be the prefix of S that ends just before v. Then, by Lemma 5.1(i),

$\rho(\hat{S}) \leqq \rho(S)$. Also by applying Lemma 5.2 to $\hat{S}$, $h_2 \leqq \rho(\hat{S}) - k \leqq \rho(S) - k$. Consider again the evaluation PQ'. By Lemma 5.3(i), $\rho(S') \leqq \rho(S)$. By (5),

$$h_{1,4,3} = \max(h_1, h_4 - b_1, h_3 - b_1 - b_4).$$

Since $h_4 - b_1 = h_2 + h_1 - t_1 \leqq (\rho(S) - k) + h_1 - (\rho(S) - k + 1) = h_1 - 1$ and $b_4 \geqq 0$, we get $h_{1,4,3} \leqq \max(h_1, h_3 - b_1)$. Also by (5),

$$h_{1,3,4} = \max(h_1, h_3 - b_1, h_4 - b_1 - b_3) \geqq \max(h_1, h_3 - b_1).$$

Thus, $h_{1,4,3} \leqq h_{1,3,4}$. By Lemmas 4.3 and 3.5, this leads to $|PQ'| \leqq |PQ|$, which is impossible.

Case 3.2. $\rho(S') \leqq \rho(S) - k + 1$.

By (5),

$$h_{1,3} = \max(h_1, h_3 - b_1) \quad \text{and} \quad h_{3,1} = \max(h_3, h_1 - b_3).$$

By Claim 2, $b_3 \geqq 1$. Therefore, $h_{3,1} \leqq \max(h_3, h_1)$. If $h_1 \geqq h_3$ then $h_{3,1} \leqq h_1 \leqq h_{1,3}$. Also, if $b_1 \leqq 0$ then $h_{1,3} \geqq \max(h_1, h_3) \geqq h_{3,1}$. In either case this leads to $h_{3,1} \leqq h_{1,3}$. Consider a sequential evaluation S'' that is identical to S except that F^v is computed before T_u. S'' has less bouncing operations than S. Let PQ'' be the ρ-efficient evaluation compatible with S''. By Lemma 5.3(ii), $\rho(S'') \leqq \rho(S)$. Since $h_{3,1} \leqq h_{1,3}$, by Lemmas 4.3 and 3.5, $|PQ''| \leqq |PQ|$ which is impossible. This completes the proof of Claim 3. □

Combining this with $h_2 = h_4$, (i) is proved.

Proof of (ii) By contradiction, assume that $\rho(\bar{S}) = \rho(S)$. Let $\overline{PQ} = P^*_\infty(\bar{S})$. We will show that this implies $h(\overline{PQ}|PQ^4|PQ^3) \leqq h(\overline{PQ}|PQ^3|PQ^4)$, leading to $|PQ'| \leqq |PQ|$. By (5),

$$\begin{aligned} h(\overline{PQ}|PQ^3|PQ^4) &= \max(h(\overline{PQ}), h_3 - b(\overline{PQ}), h_4 - b(\overline{PQ}) - b_3) \\ &\geqq \max(h(\overline{PQ}), h_3 - b(\overline{PQ})) \end{aligned}$$

and

$$h(\overline{PQ}|PQ^4|PQ^3) = \max(h(\overline{PQ}), h_4 - b(\overline{PQ}), h_3 - (\overline{PQ}) - b_4).$$

By Lemma 5.1(i) and (ii), $\rho(PQ^2) \leqq \rho(S) - 1$. Therefore, by Corollary 3.7, $h_4 = h_2 \leqq \rho(S) - 1$. By Corollary 3.8, $t(\overline{PQ}) \geqq \rho(S) - 1$. Thus, $h_4 - b(\overline{PQ}) = h_4 - t(\overline{PQ}) + h(\overline{PQ}) \leqq h(\overline{PQ})$. Therefore, $h(\overline{PQ}|PQ^4|PQ^3) = \max(h(\overline{PQ}), h_3 - b(\overline{PQ}) - b_4)$. Since $b_4 = b_2 + 1 \geqq 0$, $h(\overline{PQ}|PQ^4|PQ^3) \leqq h(\overline{PQ}|PQ^3|PQ^4)$.

Proof of (iii). Since u is not used in S^3, by Lemma 5.1(i) and (ii), $\rho(S^3) \leqq \rho(S) - 1$. Also, since the results of F^v are not used in T_w, $\rho(S^2) \leqq \rho(S) - 1$. □

THEOREM 5.5. DPA *produces an optimal evaluation for* T *if one of the following conditions is fulfilled*:

(i) *The number* uop *of unary arithmetic operations is at most three.*

(ii) *The number* R *of machine registers is at most four.*

(iii) T *has at most* 16 *vertices.*

Proof. (i) Let PQ be an optimal parallel evaluation of a tree T compatible with a sequential evaluation S having a minimal number of bouncing operations. By Theorem 4.4, we can assume that the number of bouncing operations in PQ is greater than zero. Let v be the leftmost bouncing operation of PQ, and let u and w be the normal children of v. Let T_u, T_w, and F^v be induced by v and S. Let the number of useful results computed in F^v be k. Let the projection of S on the vertices of T_u, T_w, and F^v be S^1, S^2, and S^3, respectively, and let PQ^1, PQ^2, and PQ^3 be the corresponding compatible consecutive evaluations. Since v is the leftmost bouncing operation of PQ, all the operations of F^v are normal, and therefore, we can apply Lemma 5.4(i) to PQ to get

$$b_1, b_3 \geqq 1 \quad \text{and} \quad h_1 < h_3 < h_2.$$

As $b_1, b_3 \geqq 1$, T_u and F^v must have at least two unary operations each. These unary operations are also unary in T. Therefore, $uop \geqq 4$.

(ii) By $h_2 > h_3 > h_1 \geqq 1$, $h_3 \geqq 2$ and $h_2 \geqq 3$. By applying Lemma 5.2 to $S^1|S^3$ and S^2, we get

$$h_2 \leqq \rho((S^1|S^3)|S^2) - k - 1.$$

Applying Lemma 5.1(i) twice we get $\rho(S) \geqq \rho((S^1|S^3)|S^2)$. Combining with $k \geqq 1$ and $h_2 \geqq 3$, we get $\rho(S) \geqq 5$, i.e., at least five registers are needed to compute S.

(iii) Since u and w are normal in S, $PQ^1 = PQ_u[\rho(S^1)]$ and $PQ^2 = PQ_w[\rho(S^2)]$. By Corollary 4.9,

$$h(PQ^1) \leqq \lfloor \log(n_u - uop_u + 1) \rfloor \quad \text{and} \quad h(PQ^2) \leqq \lfloor \log(n_w - uop_w + 1) \rfloor.$$

As $uop_u \geqq 2$ and $h_1 \geqq 1$, we get $n_u \geqq 3$. Also, since $h_2 \geqq 3$, $n_w - uop_w + 1 \geqq 8$. Therefore, $n_w \geqq 7$. Since $h_3 \geqq 2$ and $b_3 \geqq 1$, $t_3 \geqq 3$. Therefore, there are at least two loads and three arithmetic operations in PQ^3. Thus, $n(F^v) \geqq 5$. Taking into account v and the root of T (which is different than v as F^v and T_v are disjoint), we get $n \geqq 17$.

We see that PQ has one bouncing operation only if $uop \geqq 4$, $R \geqq 5$, and $n \geqq 17$. Otherwise there are no bouncing operations, and by Theorem 4.4, DPA yields an optimal schedule. □

The bounds on n, uop and R proved in Theorem 5.5 are tight. Consider, for example, the tree T of Fig. 1. It has $n = 17$ and $uop = 4$. The normal evaluation of T computed by DPA for $r \geqq 4$ is PQ^5 of Fig. 2 and its completion time is 12. PQ^6 of Fig. 2 is an optimal evaluation of T, its completion time is 11 and the number of registers that it uses is five.

LEMMA 5.6. *Let PQ be an optimal evaluation of T that is compatible with a sequential evaluation S. Let T' be a full binary subtree of T. Then, T' is normal in PQ.*

Proof. By contradiction, assume that T' is not normal in PQ. Let v be the leftmost bouncing operation of T'. Then, the children of v are normal. Let T_u, T_w, and F^v be induced by v and S. Also, let m and k be the number of useful results computed by S before T_u and T_w, respectively.

Case 1. $k \geqq m + 1$.

Then, we can apply Lemma 5.4(i) and get $b_1 \geqq 1$. Thus, $uop_u \geqq 2$, but this is impossible since T' is a full binary tree.

Case 2. $k < m + 1$.

Let S^1 and S^3 be the projection of S on the vertices of T_u and F^v, respectively, and let PQ^1 and PQ^3 be the corresponding compatible consecutive evaluations.

CLAIM 1. $b_3 \geqq 1$.

Proof of Claim 1. Let s be the number of results computed in S before T_u and used in F^v. Thus, F^v generates $k - 1 - (m - s)$ useful results. Since the balance of every tree in F^v is at least -1, we get $b_3 \geqq (-1)(k - 1 - (m - s)) = m - s - k + 1$. Since S is canonical, for each result computed before T_u and used in F^v a unary operation is generated in F^v. Since there are s such results, we get $b_3 \geqq m - k + 1 \geqq 1$. □

Now we complete the proof of Case 2. Consider a sequential evaluation S'' that is identical to S except that F^v is computed before T_u. Since v is normal in S'', the number of bouncing operations of S'' is strictly smaller than that of S. By Lemma 5.3(iii), $\rho(S'') \leqq \rho(S)$. Let PQ'' be the ρ-efficient evaluation that is compatible with S''. We will use the fact that $|PQ''| > |PQ|$, since otherwise PQ could not have been an optimal evaluation with a minimal number of bouncing operations. By (5),

$$h_{1,3} = \max(h_1, h_3 - b_1) \quad \text{and} \quad h_{3,1} = \max(h_3, h_1 - b_3).$$

By Claim 1, $b_3 \geq 1$. Therefore, $h_{3,1} \leq \max(h_3, h_1)$. If $b_1 \leq 0$ then $h_{1,3} \geq \max(h_1, h_3) \geq h_{3,1}$. However, if $h_{3,1} \leq h_{1,3}$ then by Lemmas 4.3 and 3.5, $|PQ''| \leq |PQ|$, which is impossible. Therefore, $b_1 \geq 1$. Thus, $uop_u \geq 2$, but this is impossible since T' is a full binary tree. □

Let T be a fixed binary tree, let $v \in T$, and let $T^0 = T_v$. By iteratively deleting the vertices $parent(v) = v_1, v_2, \cdots, root$ on the path from v to the root of T, we create subtrees $T^1, T^2, \cdots$ of T in addition to T^0 (some of these trees may be empty). We denote the forest composed of $T^1, T^2, \cdots$ by $F(v)$. Let S^i be a sequential evaluation of T^i, $i \geq 0$, and assume that for $i \geq 1$, S^i is normal, while S^0 may contain bouncing operations. For a given sequential evaluation S^0, let $\Omega(S^0)$ be the set of all possible evaluations of the form $S^0|S^1|(v_1)|S^2|(v_2)|\cdots|(root)$. Note that if S^0 is a normal evaluation of T_v, then all $S \in \Omega(S^0)$ are normal. Applying Lemma 5.1(ii) several times, we get

$$\rho(S) = \max(\rho(S^0), 1 + \max_{i \geq 1} \rho(S')). \tag{7}$$

Since for every tree T there exists a normal evaluation that computes T with no more than μ registers (Theorem 2.2), there exists $S' \in \Omega(S^0)$ such that

$$\rho(S') = \max(\rho(S^0), 1 + \max_{X \in F(v)} \mu(X)). \tag{8}$$

Let $PQ' = P^*_\infty(S^i)$. Since $P^*_\infty(S') = P^*_\infty(S^0)|P^*_\infty(S^1)|(\text{NOP}, v_1)|\cdots|(\text{NOP}, root)$, by (5),

$$h(S') = \max(h_0, \max_{i \geq 0}(h_{i+1} - b_{v_i})). \tag{9}$$

Note that for all i, $b_{v_i} \geq b_v$.

LEMMA 5.7. *Let PQ be an optimal evaluation of T having a minimal number of bouncing operations, and let S be a compatible sequential evaluation. Let $v \in T$ be a normal vertex in S, and let S^v be the projection of S on the vertices of T_v. If there exists a bouncing operation in S that is executed later than v, then there exists a normal evaluation $\hat{S} \in \Omega(S^v)$ such that $\rho(\hat{S}) \leq \rho(S)$.*

Proof. Applying Lemma 5.1 twice we show that $\rho(S^v) \leq \rho(S)$. Therefore, by (8), to prove the lemma it suffices to show that for every tree $X \in F(v)$, $\mu(X) \leq \rho(S) - 1$. Assume, by contradiction, that there exists a tree $T_u \in F(v)$ such that $\mu_u \geq \rho(S)$. By Lemma 2.1, $\mu \geq \mu_u$, implying $\mu_u \leq \rho(S)$. Therefore, we assume that $\mu_u = \rho(S)$.

Let S^0 be the projection of S on the vertices of T_u. By (8), there exists a sequential evaluation $S' \in \Omega(S^0)$ such that

$$\rho(S') = \max(\rho(S^0), 1 + \max_{X \in F(u)} \mu(X)).$$

Let PQ' be the ρ-efficient evaluation compatible with S'. We wish to prove the following:

(i) The number of bouncing operations in S' is strictly smaller than that of S.

(ii) $\rho(S') \leq \rho(S)$.

(iii) $|P^*_\infty(S')| \leq |P^*_\infty(S)|$.

Since, by Lemma 3.5, $|P^*_\infty(S')| \leq |P^*_\infty(S)|$ implies that $|PQ'| \leq |PQ|$, this will contradict the assumption that PQ is an optimal evaluation with a minimal number of bouncing operations.

Proof of (i). First note that the only bouncing operations of S' are those of S^0 and that u does not bounce. Furthermore, all the operations of S^0 that are bouncing in S', are also bouncing in S. Therefore, if u is bouncing in S then (i) follows. If u is not bouncing in S then, by Lemma 5.1, T_u is the first subtree of T computed by S. Therefore, T_v is computed in S later than T_u. Since there exists a bouncing operation in S computed later than T_v, there exists a bouncing operation computed later than T_u. Since this operation is not bouncing in S', (i) follows.

Proof of (ii). Since $\rho(S^0)=\rho(S)$, we only have to prove that for all $X\in F(u)$, $\mu(X)\leqq\rho(S)-1$. If, by contradiction, there exists $T_w\in F(u)$ such that $\mu_w=\rho(S)$, then T must contain two disjoint subtrees T_u and T_w such that $\mu_u=\mu_w=\rho(S)$. By Lemma 2.1, this leads to $\mu\geqq\mu_u+1=\rho(S)+1$, which is impossible.

Proof of (iii). Let i be the earliest time slot in which some $S_i\notin T_u$ is computed. (If $S_1\notin T_u$ then $i=1$.) Let $\bar{S}=S_1,\cdots,S_{i-1}$, and let $\underline{S}=S_i^0,\cdots,S_{|S^0|}^0$. (If $i=1$, then $\bar{S}$ is empty.) Note that $\bar{S}=S_1^0,\cdots,S_{i-1}^0$.

CLAIM 1. $\rho(\bar{S})=\mu_u$.

Proof of Claim 1. As $\mu_u=\rho(S)$, $\rho(\bar{S})$ cannot be larger than μ_u. Assume, by contradiction, that $\rho(\bar{S})<\mu_u$. By the choice of i, $\bar{S}$ computes a (possible empty) part of T_u. Since $\rho(S^0)=\mu_u$ and $\rho(\bar{S})<\mu_u$, there exists $j\geqq i$ such that $\rho_j(S_0)=\mu_u$. In time slot i, $S_i\notin T_u$ is computed. Therefore, by Lemma 5.1(ii), there exists $j>i$ such that $\rho_j(S)=\mu_u+1$, which is impossible. □

CLAIM 2. $\rho(\underline{S})<\mu_u$.

Proof of Claim 2. If $\rho(\underline{S})=\mu_u$ then since $S_i\notin T_u$ is computed, by Lemma 5.1(ii), there exists $j>i$ such that $\rho_j(S)=\mu_u+1$, which is impossible. □

CLAIM 3. $h(S)\geqq h(S^0)$.

Proof of Claim 3. Since $\bar{S}$ is a prefix of S, by (5), $h(S)\geqq h(\bar{S})$. Also, $P_\infty^*(S^0)=P_\infty^*(\bar{S})|P_\infty^*(\underline{S})$. Therefore, by (5),

$$h(S^0)=\max\,(h(\bar{S}),\,h(\underline{S})-b(\bar{S}))=\max\,(h(\bar{S}),\,h(\underline{S})-t(\bar{S})+h(\bar{S})).$$

By Claim 1, $\rho(\bar{S})=\mu_u$. Therefore, by Corollary 3.8, $t(\bar{S})\geqq\rho(\bar{S})-1=\mu_u-1$. By Claim 2, $\rho(\underline{S})<\mu_u$. Therefore, by Corollary 3.7, $h(\underline{S})\leqq\rho(\underline{S})\leqq\mu_u-1$. Therefore, $h(\underline{S})\leqq t(\bar{S})$ yielding $h(S^0)=h(\bar{S})$, and Claim 3 follows. □

Now we complete the proof of (iii). Let S^i be the projection of S' on the vertices of $T^i\in F(u)$. Since $S'\in\Omega(S^0)$, S' has the form $S^0|S^1|(u_1)|S^2|\cdots|(root)$. Therefore,

$$P_\infty^*(S')=P_\infty^*(S^0)|P_\infty^*(S^0)|P_\infty^*(S^1)|(\mathrm{NOP},\,u_1)|P_\infty^*(S^2)|\cdots|(\mathrm{NOP},\,root).$$

Note that, by the associativity property of concatenation (Lemma 4.2), the parentheses are not needed here. Let $h_i=h(S^i)$, $t_i=t(S^i)$, and $b_i=b(S^i)$. By Corollary 3.7, $h_i\leqq\rho(S^i)$. By the choice of S', $\rho(S^i)=\mu(T^i)$. As in (ii) above, for every subtree $X\in F(u)$, $\mu(X)\leqq\rho(S)-1$. Therefore, for all $i\geqq1$, $h_i\leqq\rho(S)-1$. Also, by Corollary 3.8, $t_0\geqq\rho(S^0)-1=\rho(S)-1$. By (5),

$$h_{0,1}=\max\,(h_0,\,h_1-b_0)=\max\,(h_0,\,h_1-t_0+h_0).$$

Since $t_0\geqq h_1$, $h_{0,1}=h_0$. Also, since $\rho(S^0|S^1)=\rho(S)$, by Corollary 3.8, $t_{0,1}\geqq\rho(S)-1$. We can continue this argument with $S^0|S^1$ and u_1 and iteratively with the rest of S' to show that $h(S')=h_0$. Since, by Claim 3, $h(S)\geqq h_0$, we get $h(S')\leqq h(S)$, and (iii) is proved. □

LEMMA 5.8. *For $v\in T$ let S^v be a normal evaluation of T_v. Then there exists a normal evaluation $S'\in\Omega(S^v)$ of T such that $\rho(S')\leqq\max\,(\rho(S^v),1+\mu)$.*

Proof. By (8), there exists $S'\in\Omega(S^v)$ such that

$$\rho(S')=\max\,(\rho(S^v),1+\max_{X\in F(v)}\mu(X)).$$

Since for every $X\in F(v)$, $\mu(X)\leqq\mu$, the lemma holds. □

6. Worse-case analysis of DPA. As proved in § 5, DPA does not always find optimal evaluations. Let T be a fixed binary tree rooted at *root*, and assume that DPA is not optimal for T. Let PQ be a fixed optimal ρ-efficient evaluation of T that has a minimal number (greater than zero) of bouncing operations, and let S be a compatible

sequential evaluation. Let $copt = |PQ|$ and $hopt = h(S)$. Also, let $cdpa = c(PQ_{root}[\rho(S)])$ and $hdpa = h(PQ_{root}[\rho(S)])$. In this section we derive upper bounds on

$$B = \frac{cdpa}{copt} = \frac{hdpa + a}{hopt + a} = 1 + \frac{hdpa - hopt}{hopt + a}. \tag{10}$$

6.1. Proof of 1.091 bound. It follows from (10) that if we increase *hdpa*, or decrease *hopt*, or decrease *a*, an upper bound for B is obtained. By Lemma 4.5, $hdpa \leqq h(PQ_{root}[\mu])$. Therefore, by Corollary 4.9, $hdpa \leqq \lfloor \log(n - uop + 1) \rfloor$. Since DPA is not optimal, we get $hopt < hdpa$. By Theorem 5.5 this implies that $uop \geqq 4$. Therefore, $hdpa < \log n$, and $a = (n + uop - 1)/2 > n/2$. Substituting into (10) and using $hopt \geqq 1$, we get

$$B < \frac{\log n + n/2}{1 + n/2} < 1 + \frac{2 \log n}{n}.$$

It follows that as n grows, B converges to one. To analyze (10) more precisely, let $hmax = \max_v h(PQ_v[\mu_v])$. By definition, $hmax \geqq h(PQ_{root}[\rho(S)]) = hdpa$.

LEMMA 6.1. $hmax \geqq hdpa + 1$.

Proof. By contradiction, assume that $hmax = hdpa$. Let v be the leftmost bouncing operation of S. Let T_u, T_w, and F^v be induced by v and S. Let the projection of S on the vertices of T_u and T_w be S^u and S^w, respectively. Let $PQ^u = P^*_\infty(S^u)$ and $PQ^w = P^*_\infty(S^w)$. By Lemma 5.4(i), $b_u \geqq 1$ and $h_u \leqq h_w - 2$. Since we can assume that every normal subtree of T is computed in S in the order of evaluation determined by DPA, $h_w \leqq hmax = hdpa$. Therefore, $h_u \leqq hdpa - 2$. By Lemma 5.7, there exists a sequential evaluation $S' \in \Omega(S^u)$ such that $\rho(S') \leqq \rho(S)$. Since S^u is a normal evaluation of T_u, S' is a normal evaluation of T. Let S^i be the projection of S' on the vertices of $T^i \in F(u)$, and let $PQ^i = P^*_\infty(S^i)$. Since S^i is a normal evaluation, $h_i \leqq hmax$. By (9), using $h_0 = h_u \leqq hdpa - 2$, $b_{v_i} \geqq b_u \geqq 1$, and $h_i \leqq hmax = hdpa$, we get $h(S') \leqq hdpa - 1$. Since $\rho(S') \leqq \rho(S)$, this contradicts Theorem 4.4 and completes the proof. □

Consider the vertex v of T at which $h(PQ_v[\mu_v]) = hmax$. We say that T_v is a *maximal subtree of* T if there does not exists a vertex $w \in T_v$ other than v such that $h(PQ_w[\mu_w]) = hmax$. By Lemma 4.7, a maximal subtree of T is a full binary tree. Thus, by Lemma 5.6, maximal subtrees of T are normal in S. In the sequel we denote maximal subtrees by *TMAX*. In the next lemma we establish the lower bound on the number a of the arithmetic operations of T as a function of *hmax*.

LEMMA 6.2. $a \geqq 2^{hmax-1} + 6$.

Proof. Let *TMAX* be a maximal subtree of T. Then, by Lemma 4.7, *TMAX* is a full binary tree. Also, by Corollary 4.10, $a(TMAX) \geqq 2^{hmax-1} - 1$. Let v be the leftmost bouncing operation of S. Let T_u, T_w, and F^v be induced by v and S. Note that by Lemma 5.6, $v \notin TMAX$ since v is bouncing. Also, the root of T cannot belong to *TMAX* as T cannot be a full binary tree. Let the projection of S on the vertices of T_u, T_w, and F^v be S^1, S^2, and S^3, respectively. Let $PQ^1 = P^*_\infty(S^1)$, $PQ^2 = P^*_\infty(S^2)$, and $PQ^3 = P^*_\infty(S^3)$. By Lemma 5.4(i), $b_1, b_3 \geqq 1$ and $h_2 > h_3 > h_1$. Therefore, $h_3 \geqq 2$ and $h_2 \geqq 3$. Since $b_1 \geqq 1$, $a_u \geqq 2$. By Corollary 4.10, $h_2 \geqq 3$ implies $a_w \geqq 3$. Since $h_3 \geqq 2$ and $b_3 \geqq 1$, $t_3 \geqq 3$. Therefore, $a(F^v) \geqq 3$. Since *TMAX* may belong to either T_u, T_w, or F^v, taking into account v and the root of T, we get

$$a \geqq a(TMAX) + a_u + a_w + a(F^v) + 2 - \max(a_u, a_w, a(F^v)) \geqq 2^{hmax-1} + 6. \qquad \square$$

By substituting the result of Lemma 6.2 into (10) we obtain

$$B \leqq 1 + (hdpa - hopt)/(hopt + 2^{hmax-1} + 6). \tag{11}$$

Assume that $hmax \geqq hdpa+2$. B achieves the largest value when $hmax = hdpa+2$. Substituting this into (11) results in

$$B \leqq 1+\frac{hdpa-hopt}{hopt+2^{hdpa+1}+6}$$

where the maximum value of B is 1.087 when $hopt = 1$ and $hdpa = 3$. In what follows we assume that $hmax = hdpa+1$, and thus (11) reduces to

$$B \leqq 1+\frac{hdpa-hopt}{hopt+2^{hdpa}+6}. \tag{12}$$

Assume that $hopt \geqq 2$. The largest value of B is achieved for $hopt = 2$. Substituting this into (12), we obtain

$$B \leqq 1+\frac{hdpa-2}{2^{hdpa}+8}$$

and B is bounded by 1.083. Therefore, hereafter we assume that $hopt = 1$. Substituting this into (12), we obtain

$$B \leqq 1+\frac{hdpa-1}{2^{hdpa}+7}.$$

For $hdpa \geqq 6$ and $hdpa = 2$, $B \leqq 1.091$. Therefore, in the sequel we can assume $3 \leqq hdpa \leqq 5$. We summarize the assumptions that will be used in the rest of this section:

(A1) $hopt = 1$.

(A2) $hmax = hdpa+1$.

(A3) $3 \leqq hdpa \leqq 5$.

LEMMA 6.3. *Let $u \in T$ be a normal vertex in S. Let S^u be the projection of S on the vertices of T_u. If S has at least one bouncing operation computed after u and $h(S^u) \leqq hdpa-1$ then $b_u \leqq 1$.*

Proof. Assume, by contradiction, that such vertex u exists, but $b_u \geqq 2$. Since there is at least one bouncing operation in S performed after u, by Lemma 5.7, there exists a normal sequential evaluation $S' \in \Omega(S^u)$ such that $\rho(S') \leqq \rho(S)$. By (9), using $h_0 = h(S^u) \leqq hdpa-1$, $b_{v_i} \geqq b_u \geqq 2$ and $h_i \leqq hmax = hdpa+1$, we get $h(S') \leqq hdpa-1$, contradicting Theorem 4.4. □

LEMMA 6.4. *Let TMAX be a maximal subtree of T. If TMAX is computed in S before the leftmost bouncing operation then*:

(i) $uop \geqq 6$.

(ii) $\rho(S) \geqq 3+\mu(TMAX)$.

(iii) *There exist two disjoint subtrees T_x and T_y of T computed before TMAX such that*:

(a) $b_x, b_y = 1$.

(b) *Let S^x and S^y be the projection of S on the vertices of T_x and T_y, respectively. Then $h(S^x)$, $h(S^y) \leqq hdpa-1$.*

Proof. *TMAX* cannot be the first tree to be computed in S since otherwise, $hopt \geqq hmax$ would contradict assumption (A1). Let $\bar{S}$ be the prefix of S ending just

before *TMAX*. Let *SMAX* be the projection of S on the vertices of *TMAX*. $h(P^*_\infty(\bar{S})|P^*_\infty(SMAX)) \leqq hopt = 1$. However, by (5), $h(P^*_\infty(\bar{S})|P^*_\infty(SMAX)) \geqq h(SMAX) - b(\bar{S})$, leading to $1 \geqq hmax - b(\bar{S})$. Therefore, $b(\bar{S}) \geqq hmax - 1 \geqq 3$. Thus, there are at least four unary operators before *TMAX* in S. Let these unary operators be $u_1, u_2, \cdots$ and assume that they appear in the same order as in S. For every vertex u_i there exists a unique subtree T^i of T rooted at $root_i$ such that $u_i \in T^i$ and $parent(root_i)$ is not scheduled before *TMAX*. Let $\bar{T}^1, \bar{T}^2, \cdots, \bar{T}^k$ be the subsequence of $T^1, T^2, \cdots$ such that $b(\bar{T}^i) > 0$. Let $\bar{S}^i$ be the projection of S on $\bar{T}^i$ and let $\underline{S}^i$ be the prefix of S that ends just before $\bar{T}^i$.

CLAIM 1. $\bar{T}^1$ and $\bar{T}^2$ exist and $b(\bar{T}^1) = b(\bar{T}^2) = 1$.

Proof of Claim 1. We repeat the following argument for $j = 1$ and $j = 2$. If $h(\bar{S}^j) \leqq hdpa - 1$ then the claim follows from Lemma 6.3. Otherwise, $h(\bar{S}^j) \geqq hdpa$. Since by (A3), $hdpa \geqq 3$, $\bar{S}^j$ cannot be a prefix of S. Therefore, $\underline{S}^j$ is not empty. By (5), $h(P^*_\infty(\underline{S}^j)|P^*_\infty(\bar{S}^j)) \geqq h(\bar{S}^j) - b(\underline{S}^j)$. (A1) implies

$$1 \geqq h(\bar{S}^j) - b(\underline{S}^j) \geqq hdpa - b(\underline{S}^j). \tag{13}$$

Since $\bar{T}^1$ is the leftmost tree with positive balance and $\underline{S}^1$ precedes $\bar{T}^1$, $b(\underline{S}^1) = 0$, and a contradiction to (A3) is obtained. Therefore, $b(\bar{T}^1) = 1$. Now we can plug $b(\underline{S}^2) = b(\bar{S}^1) = 1$ into (13) to obtain a contradiction to (A3). Therefore, $b(\bar{T}^2) = 1$. □

Note that $\bar{T}^1$ and $\bar{T}^2$ fulfill the conditions of (iii), and therefore, this proves (iii).

CLAIM 2. $\bar{T}^3$ exists.

Proof of Claim 2. By Claim 1, $\bar{T}^1$ and $\bar{T}^2$ contribute a total balance of 2. Since $b(\bar{S}) \geqq 3$, a third tree of positive balance must exist. □

Now we complete the proof of (i) and (ii). Since $\bar{T}^1$, $\bar{T}^2$, and $\bar{T}^3$ are disjoint trees with positive balance, each has two unary operations, proving (i). Also, at least three registers must hold the results computed those trees before *TMAX* is computed, proving (ii). □

LEMMA 6.5. *Let TMAX be a maximal subtree of T. If* $B > 1.091$ *then* $\mu = \mu(TMAX)$.

Proof. Assume, by contradiction, that $\mu > \mu(TMAX)$. Then, by Lemma 2.1, there exists at least one subtree T_x of T that is disjoint of *TMAX* and such that $\mu_x \geqq \mu(TMAX)$. By Lemma 4.8, $bop(TMAX)$, $bop_x \geqq 2^{\mu(TMAX)-1} - 1$. Since by Lemma 4.7, *TMAX* is a full binary tree, by Lemma 4.6, $hmax = \mu(TMAX)$. Also, by (A2), $hmax = hdpa + 1$. Therefore, $bop(TMAX)$, $bop_x \geqq 2^{hdpa} - 1$. We proceed by case analyses.

Case 1. $hdpa = 3$. By (10), to achieve $B > 1.091$ we need $a \leqq 20$. $bop(TMAX)$, $bop_x \geqq 7$. Let v be the leftmost bouncing operation of S. Since by Lemma 4.7, *TMAX* is a full binary tree, $v \notin TMAX$. If *TMAX* is computed before v then by Lemma 6.4(i), $uop \geqq 6$, leading to $a \geqq 21$. Therefore, we may assume that *TMAX* is computed after v.

Let T_u, T_w, and F^v be induced by v and S. Let S^1, S^2, and S^3 be the projection of S on the vertices of T_u, T_w, and F^v, respectively. Let $PQ^1 = P^*_\infty(S^1)$, $PQ^2 = P^*_\infty(S^2)$, and $PQ^3 = P^*_\infty(S^3)$. Since $b_1, b_3 \geqq 1$, we get $uop_u \geqq 2$ and $uop(F^v) \geqq 2$. Since the root of T belongs neither to *TMAX* nor T_x, we get

$$a \geqq bop(TMAX) + bop_x + 1 + uop_u + uop(F^v) \geqq 19. \tag{14}$$

We want to show that T has at least two more arithmetic vertices, contradicting $a \leqq 20$. First, consider $a(F^v)$. By Lemma 5.4(i), $h_1 < h_3$ and $b_3 \geqq 1$. Therefore, $h_3 \geqq 2$ and $t_3 \geqq 3$, implying $a(F^v) \geqq 3$. Second, consider a_w. Using Lemma 5.4(i) again we get $h_2 \geqq 3$ and by Corollary 4.10, $a_w \geqq 3$.

CLAIM 1. $v \in T_x$.

Proof of Claim 1. Otherwise, v is not counted in the above 19 vertices, and we have 20 vertices already. Since $a_w \geqq 3$ and *TMAX* is computed after v, $w \in T_x$. But then only two operations of F^v have been counted in (14). Since $a(F^v) \geqq 3$, we would get $a \geqq 21$. □

CLAIM 2. F^v is a subforest of T_x.

Proof of Claim 2. By Claim 1, F^v cannot contain T_x. Therefore, we have to show that T_x and F^v cannot be disjoint. If they are disjoint, then by $a(F^v) \geqq 3$, one of the vertices of F^v is not counted in (14), and we get $a \geqq 20$. However, T now contains three disjoint subforests, T_x, *TMAX* and F^v. The root of T can combine at most two of them, and we need at least one additional vertex that has not been counted before to build a single tree out of T_x, *TMAX*, and F^v, proving Claim 2. □

Consider $F(u)$. It consists of at least three trees, namely, T_w, F^v, and *TMAX*. Therefore, there are at least three binary vertices of T that do not belong to any of the trees of $F(u)$. By Lemma 5.7, there exists a normal evaluation $S' \in \Omega(S^1)$ such that $\rho(S') \leqq \rho(S)$.

CLAIM 3. Let $T_y \in F(u)$ be such that *TMAX* is not a subtree of T_y. Then $\mu_y < hmax$.

Proof of Claim 3. Otherwise, by Lemma 4.8, $bop_y \geqq 7$. Therefore, counting three for the binary vertices of T that belong to the path from u to the root of T, we get $a \geqq bop_y + bop(TMAX) + 3 + uop_u + uop(F^v) \geqq 21$. □

Now we complete the proof of Case 1. Remember that T_x and *TMAX* are disjoint. By Claim 1, $v \in T_x$, and therefore, $u \in T_x$. Since S' starts by evaluating T_u, the computation of T_x in S' is completed before *TMAX* even starts. By Claim 1, $v \in T_x$, and by Claim 2, F^v is a subforest of T_x. Therefore, $b_x \geqq b_u + b(F^v) \geqq 2$. Also, since by Lemma 5.4(i), $h_1 < h_3 < h_2 \leqq hmax$, by (A2), we get $h_1 \leqq hdpa - 1$. Also, by Claim 3, for every $T_y \in F(u)$ other than the one containing *TMAX*, $\mu_y \leqq hmax - 1 = hdpa$. Therefore, by Corollary 3.7, for every evaluation S^y of T_y, $h(S^y) \leqq \mu_y \leqq hdpa$. Summarizing the information needed to apply (9), we get the following:

(a) For T_u, $h_1 \leqq hdpa - 1$.

(b) For every $T^i \in F(u)$ such that *TMAX* is not a subtree of T^i, $h_i \leqq hdpa$, and $b_{v_i - 1} \geqq b_u \geqq 1$.

(c) For $T^j \in F(u)$ such that *TMAX* is a subtree of T^j, $h_j \leqq hmax = hdpa + 1$, and $b_{v_{j-1}} \geqq b_x \geqq b_u + b(F^v) \geqq 2$.

Thus, we can apply (9) and get $h(S') \leqq hdpa - 1$, contradicting Theorem 4.4.

Case 2. $hdpa = 4$. By (10), to achieve $B > 1.091$ we need $a \leqq 31$. Since $bop(TMAX)$, $bop_x \geqq 15$, and by Theorem 5.5, $uop \geqq 4$, we get $a \geqq 34$, a contradiction.

Case 3. $hdpa = 5$. By (10), to achieve $B > 1.091$ we need $a \leqq 42$. Since $bop(TMAX)$, $bop_x \geqq 31$, we get $a \geqq 62$, a contradiction. □

Since by Lemma 4.7, *TMAX* is a full binary tree, by Lemma 4.6, $\mu(TMAX) = hmax$. Therefore, if more than one maximal subtree of T would exist then, by Lemma 2.1, $\mu \geqq hmax + 1 = \mu(TMAX) + 1$ contradicting Lemma 6.5. Therefore, we can make the following additional assumptions:

(A4) T has a unique maximal tree, denoted *TMAX*.

(A5) $\mu = \mu(TMAX)$.

LEMMA 6.6. *Let v be the leftmost bouncing operation of S. Let T_u, T_w, and F^v be induced by v and S. Assume that TMAX and T_u are disjoint. There exists a subtree T_x of T with the following properties*:

(i) *T_x is computed in S before v.*

(ii) *T_x and TMAX are disjoint.*

(iii) *T_x and T_u are disjoint.*

(iv) $b_x \geqq 1$.

Let S^x be the projection of S on the vertices of T_x:

(v) $\rho(S^x) \leqq \rho(S) - 1$.

(vi) $h(S^x) \leqq hdpa - 1$.

Proof. Let S^1, S^2, and S^3 be the projection of S on the vertices of T_u, T_w, and F^v, respectively. Let $PQ^1 = P^*_\infty(S^1)$, $PQ^2 = P^*_\infty(S^2)$, and $PQ^3 = P^*_\infty(S^3)$. By Lemma 5.4(i), $b_1, b_3 \geqq 1$ and $h_1 < h_3 < h_2$. Since $h_2 \leqq hmax$, (A2) implies $h_1 \leqq hdpa - 1$.

Case 1. *TMAX* is computed in S after v.

By (A4), T_w cannot be maximal, and therefore, $h_2 < hmax$. By (A2), $h_3 \leqq h_2 - 1 \leqq hmax - 2 = hdpa - 1$. F^v is a forest with one or more trees. Since $b_3 = b(F^v) \geqq 1$, at least one of these trees has positive balance. Let $x \in F^v$ be the leftmost vertex of S such that $b_x \geqq 1$. Part (iv) holds by the construction. Also, since $x \in F^v$, (i), (ii), and (iii) hold. Since, by Lemma 5.4(iii), $\rho(S^x) \leqq \rho(S) - 1$, (v) holds. To prove (vi) let $\bar{S}$ be the (possibly empty) prefix of S^3 that is computed before T_x. By the choice of T_x, $b(\bar{S}) \leqq 0$. By (5), we get

$$hdpa - 1 \geqq h_3 \geqq \max(h(\bar{S}), h(S^x) - b(\bar{S})) \geqq h(S^x).$$

Therefore, (vi) holds.

Case 2. *TMAX* is computed in S before v.

By Lemmas 5.4(i) and 6.3, $b_1 = 1$. By Lemma 6.4(iii), there exist two disjoint subtrees T_x and T_y of T satisfying (i), (ii), (iv), and (vi). They cannot both be subtrees of T_u since $b_1 = 1$. Therefore, at least one of them satisfies (iii). Let this subtree be T_x. Since T_x is computed in S before v, it may be computed in S before F^v, or $x \in F^v$, or $x \in T_w$. By Lemma 5.4(ii) and (iii), in either case $\rho(S^x) \leqq \rho(S) - 1$. Therefore, (v) holds, and we are done. □

THEOREM 6.7. $B = cdpa/copt \leqq 1.091$.

Proof. Assume, by contradiction, that $B > 1.091$. Let v be the leftmost bouncing operation of S. Let T_u, T_w, and F^v be induced by v and S. Let S^u and S^w be the projection of S on the vertices of T_u and T_w, respectively. Let $PQ^u = P^*_\infty(S^u)$ and $PQ^w = P^*_\infty(S^w)$. By Lemma 5.4(i), $b_u \geqq 1$ and $h_w - 1 > h_u \geqq 1$. Therefore, by (A2), $h_u \leqq hdpa - 1$. By (A4), T has a unique maximal subtree *TMAX*.

CLAIM 1. *TMAX* and T_u are disjoint.

Proof of Claim 1. Otherwise, *TMAX* is a subtree of T_u. By Lemma 5.7, there exists a normal evaluation $S' \in \Omega(S^u)$ such that $\rho(S') \leqq \rho(S)$. Summarizing the information needed to apply (9), we get the following:

(a) $h_u \leqq hdpa - 1$.

(b) By (A4), no $T^i \in F(u)$ is maximal. Therefore, by (A2), $h_i \leqq hmax - 1 = hdpa$. Also, $b_{v_{i-1}} \geqq b_u \geqq 1$. Therefore, we can apply (9) and get $h(S') \leqq hdpa - 1$, contradicting Theorem 4.4. □

Let $T_y \in F(u)$ be a subtree of T such that *TMAX* is a subtree of T_y. Let $z = parent(y)$, and let s be the second child of z. There exists a subtree T_x of T that satisfies conditions (i)-(vi) of Lemma 6.6. Let the projection of S on the vertices of T_x be S^x.

Case 1. $x \in T_z$.

Subcase 1.1. $x \in T_s$. Since T_x is disjoint of T_u, $b_u \geqq 1$, $b_x \geqq 1$, $b_s \geqq 2$. By Lemma 5.7, there exists a normal evaluation $S' \in \Omega(S^1)$ such that $\rho(S') \leqq \rho(S)$. In S', T_s is completed before *TMAX* starts. Again, summarizing the information needed to apply (9), we get the following:

(a) $h_u \leqq hdpa - 1$.

(b) For every $T^i \in F(u)$, $T^i \neq T_y$, by (A2) and (A4), $h_i \leqq hmax - 1 = hdpa$, and $b_{v_{i-1}} \geqq b_u \geqq 1$.

(c) Let $T_y = T^i$. Then $h_i \leqq hmax = hdpa + 1$, and $b_{v_{i-1}} = b_s \geqq b_u + b_x \geqq 2$.

Therefore, we can apply (9) and get $h(S') \leqq hdpa - 1$, contradicting Theorem 4.4.

Subcase 1.2. $x \in T_y$. By (A5), $\mu_y = \mu(TMAX)$. Let us apply Lemma 5.8 to T_y. It follows that there exists a normal evaluation S^y of T_y that computes T_x first by S^x such that $\rho(S^y) \leqq \max(\rho(S^x), 1 + \mu_y)$. If *TMAX* is computed in S before v then, by Lemma 6.4(ii), $\rho(S) \geqq 3 + \mu(TMAX)$. Otherwise, $\rho(S) \geqq 2 + \mu(TMAX)$, since by Lemma 6.6(i) both T_u and T_x are computed in S before v and therefore, before *TMAX*. Therefore, $1 + \mu_y \leqq \rho(S) - 1$. By Lemma 6.6(v), $\rho(S^x) \leqq \rho(S) - 1$. Together, we get $\rho(S^y) \leqq \rho(S) - 1$. By (8), there exists a normal evaluation $S' \in \Omega(S^1)$ such that

$$\rho(S') = \max(\rho(S^u), 1 + \max_{X \in F(u)} \mu(X)).$$

However, we would like to make a stronger assumption on the evaluation of T_y in S', namely, that it is identical to S^y. Then we get

$$\rho(S') = \max(\rho(S^u), 1 + \rho(S^y), 1 + \max_{X \in F(u), X \neq T_y} \mu(X)).$$

By Lemma 5.7, $\rho(S^u)$ and $1 + \max_{X \in F(u), X \neq T_y} \mu(X)$ are both bounded by $\rho(S)$. Since $\rho(S^y) \leqq \rho(S) - 1$, we get $\rho(S') \leqq \rho(S)$.

CLAIM 2. $h(S^y) \leqq hdpa$.

Proof of Claim 2. Note that S^y computes first T_x by S^x and then proceeds with the rest of T_y. Let $\hat{S} \in \Omega(S^x)$ be a sequential evaluation of T whose projection on T_y is S^y. Let S^i be the projection of $\hat{S}$ on the vertices of $T^i \in F(x)$. By Lemma 6.6(vi), $h(S^x) \leqq hdpa - 1$. Also, by Lemma 6.6(iv), $b_x \geqq 1$. Since for every S^i, $h(S^i) \leqq hmax = hdpa + 1$, using (9), we get $h(\hat{S}) \leqq hdpa$. However, since S^y is a prefix of $\hat{S}$, $h(S^y) \leqq h(\hat{S}) \leqq hdpa$. □

Now we complete the proof of Subcase 1.2. Summarizing the information needed to apply (9), we get the following:

(a) $h_u \leqq hdpa - 1$.

(b) Since *TMAX* is a unique maximal subtree of T and by Claim 2, for every $T^i \in F(u)$, $h_i \leqq hmax - 1 = hdpa$.

(c) $b_{v_{i-1}} \geqq b_u \geqq 1$.

Therefore, we can apply (9) and get $h(S') \leqq hdpa - 1$, contradicting Theorem 4.4.

Case 2. $x \notin T_z$. x cannot be an ancestor of z since by the choice of x, T_x, and *TMAX* are disjoint and *TMAX* is a subtree of T_z. Let $T_t \in F(x)$ be the subtree that contains z. By (A5), $\mu_t = \mu(TMAX)$. Let us apply Lemma 5.8 to T_t. It follows that there exists a normal evaluation S^t of T_t that computes T_u first by S^u and $\rho(S^t) \leqq \max(\rho(S^u), 1 + \mu_t)$. If *TMAX* is computed in S before v then, by Lemma 6.4(ii), $\rho(S) \geqq 3 + \mu(TMAX)$. Otherwise, $\rho(S) \geqq 2 + \mu(TMAX)$ since by Lemma 6.6(i), both T_u and T_x are computed in S before v and therefore, before *TMAX*. Therefore, $1 + \mu_t \leqq \rho(S) - 1$. By Lemma 5.4(ii), $\rho(S^u) \leqq \rho(S) - 1$. Together, we get $\rho(S^t) \leqq \rho(S) - 1$. By (8), there exists a normal evaluation $S' \in \Omega(S^x)$ such that

$$\rho(S') = \max(\rho(S^x), 1 + \max_{X \in F(x)} \mu(X)).$$

However, as in Subcase 1.2, we would like to make a stronger assumption on the evaluation of T_t in S', namely, that it is identical to S^t. Then we get

$$\rho(S') = \max(\rho(S^x), 1 + \rho(S^t), 1 + \max_{X \in F(x), X \neq T_t} \mu(X)).$$

By Lemma 5.7, $\rho(S^x)$ and $1+\max_{X\in F(x),X\neq T_t}\mu(X)$ are both bounded by $\rho(S)$. Since $\rho(S')\leqq\rho(S)-1$, we get $\rho(S')\leqq\rho(S)$. Also, note that, by Lemma 6.6(vi), $h_x\leqq hdpa-1$.

CLAIM 3. $h(S')\leqq hdpa$.

Proof of Claim 3. Note that S' first computes T_u by S^u and then proceeds with the rest of T_t. Let $\hat{S}\in\Omega(S^u)$ be a sequential evaluation of T whose projection on T_t is S'. Let S^i be the projection of $\hat{S}$ on the vertices of $T^i\in F(u)$. By Lemma 5.4(i), $h(S^u)\leqq hdpa-1$, and $b_u\geqq 1$. Since for every S^i, $h(S^i)\leqq hmax=hdpa+1$, using (9), we get $h(\hat{S})\leqq hdpa$. However, since S' is a prefix of $\hat{S}$, $h(S')\leqq h(\hat{S})\leqq hdpa$. □

Now we complete the proof of Case 2. Summarizing the information needed to apply (9), we get the following:

(a) $h_x\leqq hdpa-1$.

(b) Since *TMAX* is a unique maximal subtree of T and by Claim 3, for every $T^i\in F(x)$, $h_i\leqq hmax-1=hdpa$.

(c) $b_{v_{i-1}}\geqq b_x\geqq 1$.

Therefore, we can apply (9) and get $h(S')\leqq hdpa-1$, contradicting Theorem 4.4. □

6.2. A worse-case example. In § 6.1, we proved that $B\leqq 1.091$. Actually, for the tree of Fig. 1, $B=1.091$. For five registers or more, the optimal evaluation of this tree is PQ^6 of Fig. 2 with $c=11$, while $c(PQ_{root}[r])=12$ for $r\geqq 4$.

6.3. An ε-approximation scheme. Since $B\to 1$ as $n\to\infty$, we can develop an *ε-approximation algorithm*—given ε, design an algorithm that finds a schedule that is worse than the optimal by at most $B\leqq 1+\varepsilon$. The idea is straightforward: apply some exhaustive algorithm to find optimal schedule for trees with at most n_t vertices, and use DPA for larger trees. Formally, given ε, let n_ε be the solution of $n_\varepsilon=2^{n_\varepsilon\varepsilon/2}$. Since

$$B\leqq 1+2\log\frac{n_\varepsilon}{n_\varepsilon}$$

we get $B\leqq 1+\varepsilon$. For example, for $\varepsilon=0.1$, $n_\varepsilon=144$, and for $\varepsilon=0.01$, $n_\varepsilon=224$.

7. Extensions and open problems. Our results can be extended in a number of ways, but they also leave a few problems open for further research.

7.1. Larger arity. Every operation of our machine model has at most two operands, and therefore, only binary trees have been considered. Obviously, the machine model may be extended to allow larger arity of the expression trees. The DP scheme can be easily extended to nonbinary trees as suggested in [AJ76]. In the extended DP scheme at each internal vertex v all possible orders of computations of the subtrees rooted at the children of v should be considered and a minimal cost evaluation chosen. The complexity of this algorithm is still linear in n, but the amount of work performed at each vertex is exponential in the maximum allowed arity. The question is, how to extend the approximation techniques developed in this paper to nonbinary trees. It turns out that the generalization of our algorithm may yield schedules that are worse than the optimal by d time units even for trees with only $O(d)$ vertices.

7.2. Arbitrary execution times. We can relax our restriction that the execution time of memory access is equal to that of the arithmetic operations. Let τ_1 be the number of machine cycles required for memory access, and τ_2 be the elapsed time of every arithmetic operation. The P_r transformation of § 2.4 can be upgraded to accommodate unequal execution times of machine operations. Arbitrary execution times of operations affect the approximation results derived in § 6. If $\tau_2>\tau_1$ then all the results are valid, and it is even possible to decrease the worse-case ratio of the completion

times of the schedule generated by DPA and the shortest schedule. Moreover, for binary trees if $\tau_2 \geqq 2\tau_1$ then the algorithm becomes optimal since in this case optimal evaluations can always be found among normal form evaluations. By contrast, if $\tau_1 > \tau_2$ then it has an effect similar to that of larger arity (§ 7.1).

7.3. Store operations. Throughout the previous sections we have assumed that it is always possible to compute the given expression tree using R machine registers. Clearly, this assumption does not always holds.

DPA can be extended to a minimal store algorithm (MS) by producing store operations only when needed. Such a treatment of store operations is not general enough as in the dynamic programming algorithm of [AJ76]. It is quite similar to the approach proposed in [SU70] that leads to an optimal algorithm for a class of sequential machines.

In our case, the MS algorithm works fine when the cost of a store operation is significantly larger than the cost of the other operations. However, if a store operation takes about the same time as the other operations, then a new phenomenon arises. Sometimes it is preferable to perform a store operation even if it is not absolutely necessary so as to increase the parallelism between arithmetic and memory operations.

For example, consider the tree of Fig. 4. It can be computed using four registers. Since the minimal number of registers needed to compute T_2 without a store is four, T_2 must be computed first. Then, the cost is 16. But if T_1 is computed first and a store operation is used after the completion of T_1, the cost is reduced to 15. As a consequence of the last observation, more work concerning store operations is needed.

7.4. Different register allocation assumptions. Different classes of machines impose different restrictions on the use of registers. In our machine model, we only assumed that when an arithmetic operation is performed in parallel with a load, results of these

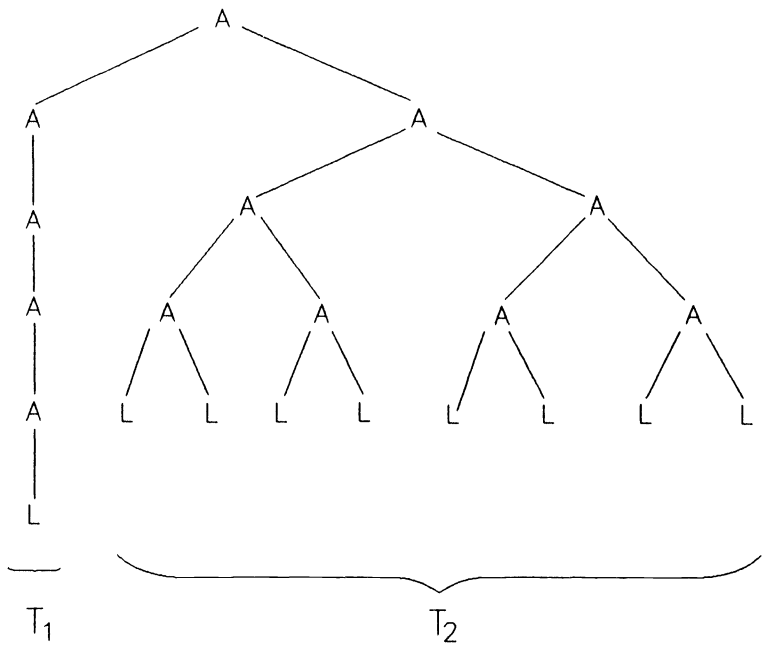


FIG. 4. *The tree demonstrating the store phenomenon.*

two parallel instructions are stored in different registers. Additional constraints on the structure of the machine registers may invalidate our results.

A simple refinement of our machine model is to assume that the register released by a binary arithmetic operation at time slot i cannot be used by a load operation performed in the same time slot. The earliest time that this register can be used again is time slot $i+1$. The main effect of this additional requirement is that an evaluation PQ of width ρ may use $\rho+1$ registers on such machine, i.e., one register more than previously. Still when we allow a number registers $r \geqq \rho(PQ)+1$, the completion time of PQ is the same on both machine models. It can be shown that when the optimal evaluation is bouncing, the number of registers needed by an optimal evaluation is greater than the minimal width of the best normal form evaluation. Therefore, the approximation results of § 6 remain valid.

7.5. ε-approximation algorithms. In § 6 we have shown how an ε-approximation algorithm can be obtained from DPA using the enumeration on the trees with $n \leqq n_\varepsilon$. Clearly, this approach is not practical since even to achieve $\varepsilon = 0.01$ the enumeration should be performed on the trees with $n = 2224$. Another ε-approximation scheme may be developed by keeping several candidates for each $PQ_v[r]$. Analyzing such a scheme is probably a nontrivial job.

7.6. The complexity of the optimal algorithm. This is the major question that remains open.

Appendix. Example for the DPA algorithm. The following example illustrates how DPA processes the vertices of the expression tree of Fig. 1. Let $R = 5$. We organize the results computed by DPA in Tables 1 and 2. The entry in the row v_i and in the column $PQ[j]$ of Table 1 is the $\langle c, h, t\rangle$ triple corresponding to $PQ_{vi}[j]$ where instead of $\langle\infty, \infty, \infty\rangle$ we use a single ∞. The entry in the row v_i and in the column $S[j]$ of Table 2 is the evaluation $S_{v_i}[j]$ that is compatible with $PQ_{vi}[j]$.

TABLE 1
The c, h, and t parameters computed by DPA *for the tree of Fig.* 1.

Vertex	$PQ[1]$	$PQ[2]$	$PQ[3]$	$PQ[4]$	$PQ[5]$
v_1	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_2	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_3	∞	⟨3, 2, 1⟩	⟨3, 2, 1⟩	⟨3, 2, 1⟩	⟨3, 2, 1⟩
v_4	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_5	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_6	∞	⟨3, 2, 1⟩	⟨3, 2, 1⟩	⟨3, 2, 1⟩	⟨3, 2, 1⟩
v_7	∞	∞	⟨6, 3, 2⟩	⟨6, 3, 2⟩	⟨6, 3, 2⟩
v_8	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_9	⟨2, 1, 1⟩	⟨2, 1, 1⟩	⟨2, 1, 1⟩	⟨2, 1, 1⟩	⟨2, 1, 1⟩
v_{10}	⟨3, 1, 2⟩	⟨3, 1, 2⟩	⟨3, 1, 2⟩	⟨3, 1, 2⟩	⟨3, 1, 2⟩
v_{11}	∞	∞	⟨9, 3, 4⟩	⟨8, 2, 3⟩	⟨8, 2, 3⟩
v_{12}	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_{13}	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩	⟨1, 1, 0⟩
v_{14}	∞	⟨3, 2, 1⟩	⟨3, 2, 1⟩	⟨3, 2, 1⟩	⟨3, 2, 1⟩
v_{15}	∞	⟨4, 2, 2⟩	⟨4, 2, 2⟩	⟨4, 2, 2⟩	⟨4, 2, 2⟩
v_{16}	∞	⟨5, 2, 3⟩	⟨5, 2, 3⟩	⟨5, 2, 3⟩	⟨5, 2, 3⟩
v_{17}	∞	∞	⟨13, 3, 6⟩	⟨12, 2, 5⟩	⟨12, 2, 5⟩

TABLE 2
The sequential version of the evaluations computed by DPA *for the tree of Fig.* 1.

Vertex	$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$
v_1	(1)	(1)	(1)	(1)	(1)
v_2	(2)	(2)	(2)	(2)	(2)
v_3	Λ	(1-3)	(1-3)	(1-3)	(1-3)
v_4	(4)	(4)	(4)	(4)	(4)
v_5	(5)	(5)	(5)	(5)	(5)
v_6	Λ	(4-6)	(4-6)	(4-6)	(4-6)
v_7	Λ	Λ	(1-7)	(1-7)	(1-7)
v_8	(8)	(8)	(8)	(8)	(8)
v_9	(8, 9)	(8, 9)	(8, 9)	(8, 9)	(8, 9)
v_{10}	(8-10)	(8-10)	(8-10)	(8-10)	(8-10)
v_{11}	Λ	Λ	(1-11)	(8-10, 1-7, 11)	(8-10, 1-7, 11)
v_{12}	(12)	(12)	(12)	(12)	(12)
v_{13}	(13)	(13)	(13)	(13)	(13)
v_{14}	Λ	(12-14)	(12-14)	(12-14)	(12-14)
v_{15}	Λ	(12-15)	(12-15)	(12-15)	(12-15)
v_{16}	Λ	(12-16)	(12-16)	(12-16)	(12-16)
v_{17}	Λ	Λ	(1-17)	(8-10, 1-7, 11-17)	(8-10, 1-7, 11-17)

REFERENCES

[AJ76] A. V. AHO AND S. C. JOHNSON, *Optimal code generation for expression trees*, J. Assoc. Comput. Mach., 23 (1976), pp. 488-501.

[AJUa77] A. V. AHO, S. C. JOHNSON, AND J. D. ULLMAN, *Code generation for expression with common subexpressions*, J. Assoc. Comput. Mach., 24 (1977), pp. 146-160.

[AJUb77] ———, *Code generation for machines with multiregister operations*, Proc. 4th ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1977, pp. 21-28.

[BJR87] D. BERNSTEIN, J. M. JAFFE, AND M. RODEH, *Scheduling arithmetic and load operations in parallel with no spilling*, Tech. Report 88.225, IBM-Israel Scientific Center, July 1987.

[BJPR85] D. BERNSTEIN, J. M. JAFFE, R. Y. PINTER, AND M. RODEH, *Optimal scheduling of arithmetic operations in parallel with memory access*, Tech. Report 88.136, IBM-Israel Scientific Center May 1985. A preliminary version has appeared in the Proc. 12th ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1985, pp. 325-333.

[BS76] J. L. BRUNO AND R. SETHI, *Code generation for a one-register machine*, J. Assoc. Comput. Mach., (1976), pp. 502-510.

[H84] J. L. HENNESY, VLSI *processor architecture*, IEEE Trans. Comput., (1984), pp. 1221-1246.

[HB84] K. HWANG AND F. A. BRIGGS, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.

[N67] I. NAKATA, *On compiling algorithms for arithmetic expression*, Comm. ACM, 18 (1967), pp. 492-494.

[Ra83] G. RADIN, *The* 801 *minicomputer*, IBM J. on R&D, 27 (1983), pp. 237-246.

[Re69] R. R. REDZIEJOWSKI, *On arithmetic expressions and trees*, Comm. ACM, 12 (1969), pp. 81-84.

[Ru78] R. M. RUSSELL, *The Cray-*1 *computer system*, Comm. ACM, 21 (1978), pp. 63-72.

[SU70] R. SETHI AND J. D. ULLMAN, *The generation of optimal code for arithmetic expressions*, J. Assoc. Comput. Mach., 17 (1970), pp. 715-728.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1128-1139, December 1989

004

COMPUTING SIMPLE CIRCUITS FROM A SET OF LINE SEGMENTS IS NP-COMPLETE*

DAVID RAPPAPORT†

Abstract. Given a collection of line segments in the plane, the segments are connected by their endpoints to construct a simple circuit. (A simple circuit is the boundary of a simple polygon.) However, there are collections of line segments where this cannot be done. In this note it is proved that deciding whether a set of line segments admits a simple circuit is NP-complete. The NP-completeness proof relies on the fact that line segments may intersect at their endpoints. Deciding whether a set of horizontal line segments can be connected with horizontal and vertical line segments to construct an orthogonal simple circuit is also shown to be NP-complete.

Key words. NP-complete problem, Hamiltonian path, Euclidean travelling salesman problem, computational geometry, line segments

AMS(MOS) subject classifications. 68Q15, 68U05

1. Introduction. A natural generalization of the problem of finding simple circuits from a set of points is the problem of finding a simple circuit from a set of line segments. In general a set of line segments does not necessarily admit a simple circuit. An example of a set of line segments that does not admit a simple circuit is given in Fig. 1. An interesting question is when do a set of line segments admit a simple circuit?

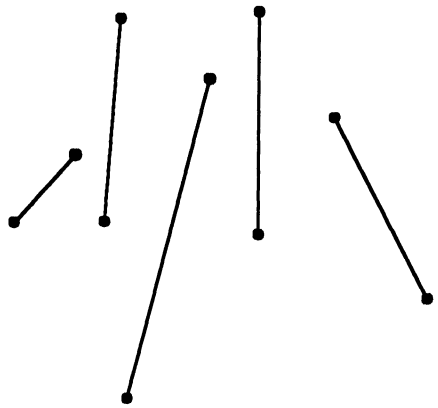


FIG. 1

The task of obtaining a simple circuit from a set of points is a recurring theme that appears in a variety of applications. In network routing problems the tour of shortest Euclidean distance that begins and ends at a common site and visits all other sites exactly once is a simple circuit. This is the Euclidean travelling salesman problem and is known to be NP-hard [3]-[5]. Simple circuits have also been used in the area

* Received by the editors February 2, 1988; accepted for publication (in revised form) February 20, 1989. A preliminary version of this paper appeared at the 3rd ACM Symposium on Computational Geometry, June 1987, pp. 322-330. The research of this author was partially supported by Natural Sciences and Engineering Research Council of Canada grant A9204.

† Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N6.

of pattern recognition for extracting perceptual information from sparse data [2], [7], [9], [16], [17]. Surprisingly, the problem of computing simple circuits from a set of line segments has not received much attention.

Given a collection of sites on a plane and the requirement that certain sites are visited in a prescribed order, the task of finding a shortest tour reduces to finding shortest tours from a set of line segments, or chains of line segments. If, in addition, edges in the tour that cross greatly increases the cost of the tour (because bridges or tunnels are required) then we are interested in finding a shortest tour with few crossings. In the extreme this reduces to deciding if a set of line segments, or chains of line segments, admits a simple circuit. In the context of pattern recognition, meaningful perceptual information may be obtained from a collection of line segments, or chains of line segments, by computing a minimal set of disjoint simple circuits. Again, in the extreme this problem reduces to deciding if a set of line segments admits one simple circuit.

In [13] it has been shown that, if the set of line segments are constrained so that every segment has at least one of its endpoints on the convex hull of the segments, an $O(n \log n)$ algorithm can be used to determine whether the set admits a simple circuit. Furthermore, we can deliver simple circuits, and optimize over the area and the perimeter of the polygons constructed, in the same time bound. Other special cases of the problem of obtaining a simple circuit from a set of line segments are discussed in [1], [12].

In this note it is shown that to determine whether a set of segments admits a simple circuit is NP-complete. After preliminary definitions a reduction is given to a variant of this problem. In this variant we are given a set of orthogonal line segments and are asked whether we can construct a simple circuit whose edges are orthogonal to the coordinate axes. After proving that the orthogonal simple circuit problem is NP-complete we show that the simple circuit problem is NP-complete. A detail in the proof requires that some of the segments intersect at their endpoints.

2. Preliminaries. A *simple circuit* is a sequence of edges, $e_0, e_1, \cdots, e_{k-1}$, such that for all $0 \leqq i < k$, e_i and $e_{(i+1) \bmod k}$ intersect at their endpoints, and no other intersections between edges occur. A simple circuit is the boundary of a simple polygon. Let S be a set of line segments in the plane. We require that the segments be properly disjoint, that is, no segments intersect in their interiors; however, we permit segments of S to intersect at their endpoints. If a set A can be found such that $S \cup A$ is a simple circuit of $|S|+|A|$ edges, then we say that S *admits* a simple circuit and A is a set of *augmenting segments.* To obtain a set of augmenting segments, we begin by considering a set of segments as *candidates.* We say that two points *see* each other, if the line segment between them does not intersect any segment. Since we are looking for crossing free circuits it is natural to choose as candidates connections of endpoints of segments that see each other.

An *orthogonal simple circuit* is a simple circuit whose edges are orthogonal to the coordinate axes. One can say two points *orthogonally see* each other, if they agree in one of their coordinates and they see each other.

It will be shown that the following problem is NP-complete.

SIMPLE CIRCUIT (SC).

Instance. A set of line segments S.

Question. Does S admit a simple circuit?

To simplify the presentation first it will be shown that the following more structured problem is NP-complete.

ORTHOGONAL SIMPLE CIRCUIT (OSC).

Instance. A set of line segments S orthogonal with respect to the coordinate axes.

Question. Does S admit an orthogonal simple circuit?

3. Orthogonal simple circuits is NP-complete. In this section it will be shown that the orthogonal simple circuit problem (OSC) is NP-complete. The following problem is known to be NP-complete [3].

HAMILTONIAN PATH IN PLANAR CUBIC GRAPHS (HPPCG).

Instance. A planar cubic (all vertices are of degree three) graph $G=(V, E)$.

Question. Is there a Hamiltonian path in G?

The idea behind the transformation of HPPCG to OSC is to build *modules* out of collections of line segments. Given a planar cubic graph G, the collection of modules $M(G)$ is constructed. Each module m_a will uniquely represent a vertex a of G. The edges of the graph will be simulated by a subset of the candidates of the collection of modules. The remainder of this section leads to the conclusion that a Hamiltonian path exists in a planar cubic graph, if and only if the set of corresponding modules admits an orthogonal simple circuit.

To construct modules corresponding to vertices we first compute a *rectilinear planar layout* of the graph. This layout maps vertices to horizontal line segments and maps edges to vertical line segments, such that all the endpoints of segments are at positive integer coordinates. Two horizontal segments are intersected by the endpoints of a vertical segment, if and only if the corresponding vertices are adjacent in the graph. Figure 2 shows a straight line drawing of a planar cubic graph with its rectilinear planar layout.

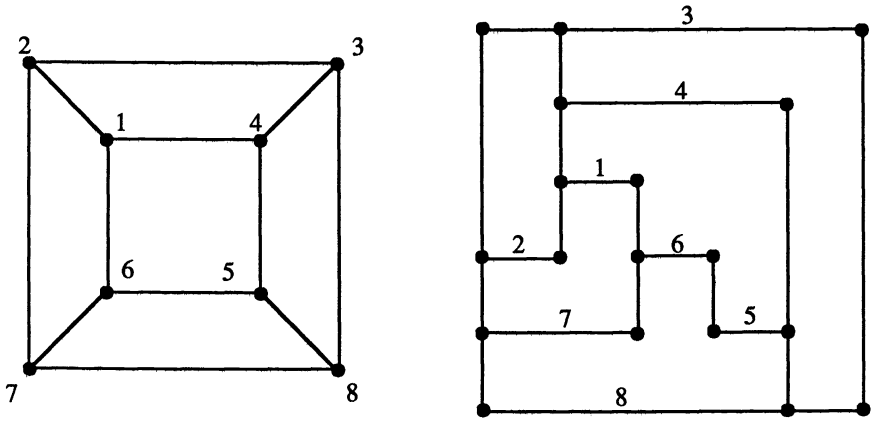


FIG. 2

In [14] it has been shown that a rectilinear planar layout can be computed for planar graphs with n vertices in $O(n)$ time. The height of the layout of this algorithm is guaranteed to be at most n, and the width at most F, where F is the number of faces in the graph. In cubic graphs $F=n/2+2$, by Euler's relation. For details regarding the algorithm used to obtain rectilinear planar layouts, refer to [14].

Given a planar cubic graph $G=(V, E)$, a configuration of modules can be constructed in polynomial time. In Fig. 3, we show a configuration of modules corresponding to the graph in Fig. 2. For completeness we give a procedure to construct such a configuration of modules.

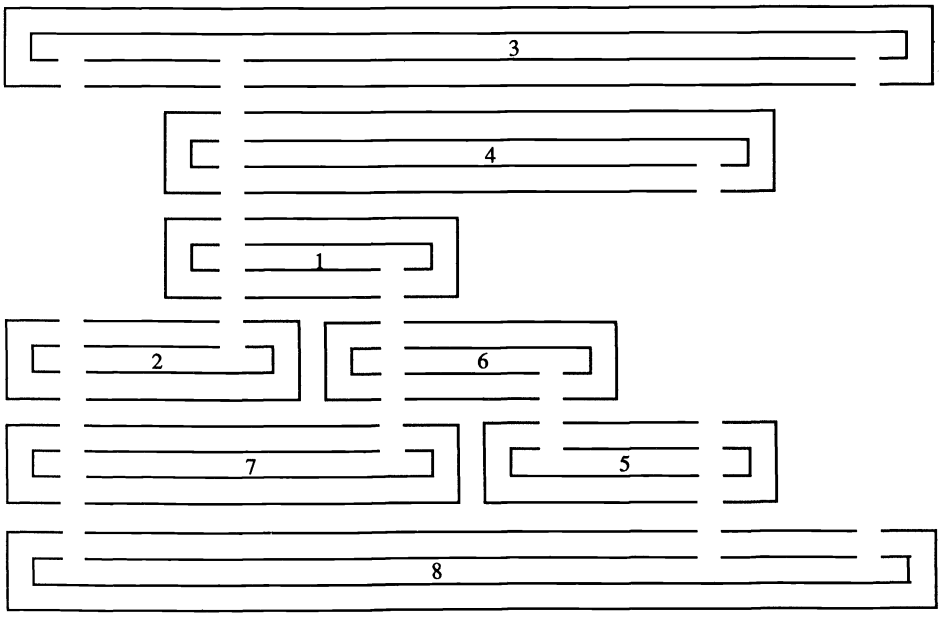


FIG. 3

PROCEDURE CONSTRUCT MODULES.

(1) Obtain a rectilinear planar layout of G.

(2) **for** each vertex v in G **do** steps 2.0, 2.1, and 2.2

(2.0) Let (x_l, y), (x_r, y) denote the coordinates of the horizontal line segment h that corresponds to v. Every vertex in G is of degree three. Therefore, there are three vertical segments intersecting h. Denote their intersection points as (x_1, y), (x_2, y), and (x_3, y).

if x_l is the top endpoint of its vertical line segment
then $top_i \leftarrow$ true **else** $top_i \leftarrow$ false.

(2.1) Construct an axis parallel outer rectangle with corners, $(6x_l, 4y)$, $(6(x_r+1)-1, 4y+3)$ and an axis parallel inner rectangle with corners, $(6x_l+1, 4y+1)$, $(6(x_r+1)-2, 4y+2)$.

(2.2) **for** $i \leftarrow 1$ **to** 3 **do**
if top_i is true **then**
Place a gap of width one at $(6x_i+2, 4y)$ of the outer rectangle and at $(6x_i+2, 4y+1)$ of the inner rectangle.
else
Place a gap of width one at $(6x_i+2, 4y+3)$ of the outer rectangle and at $(6x_i+2, 4y+2)$ of the inner rectangle. □

Denote the gaps found in the outer and inner frames of the modules as outer and inner *doors*, respectively. The endpoints at the extremes of these gaps will be referred to as left and right *doorjambs*. An inner and outer door pair will be referred to collectively as a DOOR. If the edge (a, b) is in G, then a door of module m_a faces a door of module m_b. We say that these doors are *neighbours*. By a logical extension, modules m_a and m_b are also termed as neighbours, if their doors are neighbours. Denote an alternating sequence of vertices and edges beginning and ending at a vertex, such that each edge

intersects its neighbouring edge at a vertex and no two edges intersect except at their endpoints, as a *polygonal chain.* Observe that the topology of every module is the same. Every module consists of three polygonal chains beginning and ending at outer doorjambs, and three polygonal chains beginning and ending at inner doorjambs.

Suppose there is a Hamiltonian path in the graph G. For notational simplicity assume that the sequence of vertices $1, 2, \cdots, n$ corresponds to the Hamiltonian path. For every edge, $(i, i+1)$, $i = 1, \cdots, n-1$, in the Hamiltonian path, we *link* the module m_i to module m_{i+1}. Since the vertices i and $i+1$ are neighbours in G, then m_i and m_{i+1} have neighbouring doors. We will use the following terminology to describe how modules are linked. We say a door is *augmented* if there are augmenting segments using both of its doorjambs. Denote a *gate* as an augmenting segment with endpoints on the left and right doorjambs of the same door. Denote an *egress* as a pair of augmenting segments linking the left and right outer doorjambs of neighbouring doors. Denote an *ingress* as a pair of augmenting segments linking the left and right doorjambs of an outer door to an inner door. See Fig. 4.

PROCEDURE LINK.

(1) Augment a DOOR of m_1 not a neighbour of m_2 with an ingress, and augment another DOOR of m_1 not a neighbour of m_2 with two gates. These choices can be made arbitrarily.

(2) **for** $i \leftarrow 2$ **to** n **do** steps 2.0, 2.1, and 2.2

(2.0) use an egress to augment the neighbouring outer doors of m_i and m_{i-1}.

(2.1) augment the corresponding inner doors of m_i and m_{i-1} with gates.

(2.2) **if** $i < n$ **then**

augment the DOOR of m_i not a neighbour of m_{i+1} or m_{i-1} with an ingress

else

augment a DOOR of m_n not a neighbour of m_{n-1} with an ingress.

(3) Augment the remaining unaugmented DOOR of m_n with two gates.

It should be clear that the procedure LINK has a linear worst-case running time. We now show that LINK produces an orthogonal simple circuit.

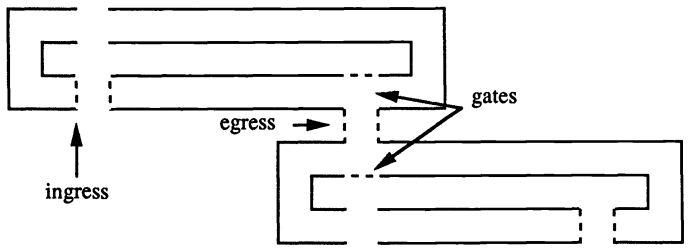


FIG. 4

LEMMA 1. *Given an ordering of the modules $M(G)$ corresponding to a Hamiltonian path in the graph G, procedure* LINK *produces an orthogonal simple circuit.*

Proof. Procedure LINK maintains the loop invariant that before and after every iteration of step (2) the modules m_1 to m_i and the added augmenting segments form

two disjoint polygonal chains. One chain begins at the inner right doorjamb and ends at the corresponding right outer doorjamb of a DOOR of module m_i. The other polygonal chain begins and ends at the left opposing inner and outer doorjambs of the same DOOR. See Fig. 5. After step (2) terminates (and $i = n$) we have two polygonal chains using all of the segments in $M(G)$. These chains are joined to complete the simple circuit in step (3). □

It remains to show that every orthogonal simple circuit in $M(G)$ can be used to obtain a Hamiltonian path in G in polynomial time. We denote a sequence of neighbouring modules, $m_1, m_2, \cdots, m_n$, augmented using procedure LINK as a *path of modules.* We will show that every orthogonal simple circuit that can be obtained from $M(G)$ is a path of modules.

Let us examine the ways in which a module can be linked to its neighbouring module with augmenting segments. A first step is to establish for each module a list of candidates, that is, line segments joining doorjambs that orthogonally see each other. In Fig. 6, a module is shown with its entire set of candidates drawn in dashed lines. The following lemma exhibits a crucial property of two neighbouring modules.

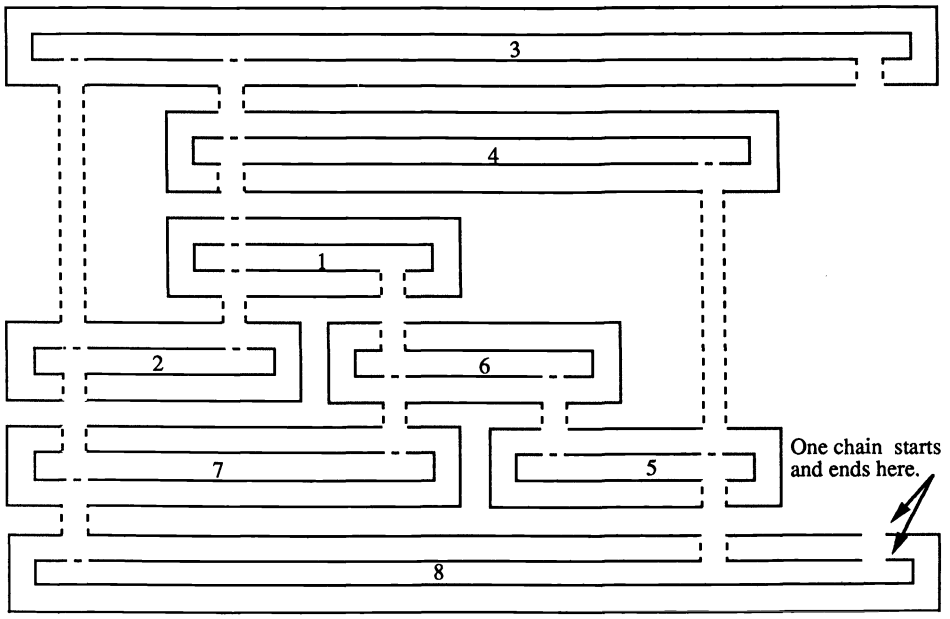


FIG. 5

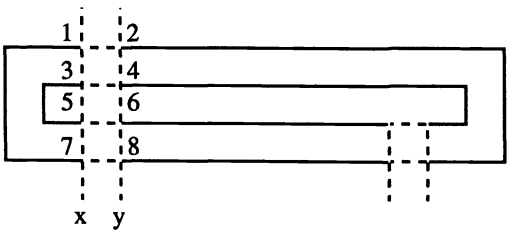


FIG. 6

LEMMA 2. *If two neighbouring modules are joined by a single augmenting segment, then an orthogonal simple circuit cannot be obtained in the collection of modules.*

Proof. First observe that all DOORS of modules have a similar structure. There are cases where there are two DOORS that are in the same vertical column. This is the case that will be argued. In the simpler case (for example, the unlabeled door in Fig. 6) a similar but simpler argument can be used. Referring to Fig. 6, assume we have the augmenting segment $(7, x)$. The edge $(8, 6)$ is forced, because we are assuming there is only one augmenting segment between modules. Now 5 can only be connected to 3, causing a disjoint circuit. This rules out the possibility of obtaining a single orthogonal simple circuit. Assume instead that we have augmenting segment $(8, y)$. This forces $(7, 5)$ and $(6, 4)$, which forces $(3, 1)$. But this causes a disjoint circuit. Therefore, neighbouring modules are joined with two augmenting segments, in every orthogonal simple circuit. □

We have established that at each door, modules are joined with two augmenting segments. Therefore, saying two modules are *connected*, refers to linking the modules with an egress. It should be clear that the corresponding inner doorjambs must also be joined with gates, because these doorjambs do not orthogonally see any alternate points. Another required fact is Lemma 3.

LEMMA 3. *A module can be connected to at most two of its neighbours.*

Proof. Assume that a module is connected to three of its neighbours. This leaves the inner frame disconnected from the rest of the segments of the module and isolated from all other modules. □

It should now be clear that the internal structure in each module forces each door to have two augmenting segments connecting each neighbour, and it forces each module to be connected to at most two of its neighbours. It is obvious that every module must be connected to at least one other module in every orthogonal simple circuit. Let the *circuit degree* of a module denote the number of neighbours a module is connected to in an orthogonal simple circuit. Clearly the circuit degree of a module can only be one or two.

LEMMA 4. *Every orthogonal simple circuit from a collection of modules is a path of modules.*

Proof. We cannot have an odd number of modules with circuit degree one. To see this, first obtain the total sum σ of all circuit degrees. If there is an odd number of modules of circuit degree one, then σ is odd. But σ must always be even, since σ counts each module connection twice.

Suppose there are four or more modules with circuit degree one. Since each module can have circuit degree at most two, σ is at most $2(n-4)+4$. As in graphs, if the sum of the degrees is less than $2n$ (less than n edges), then the graph must have disconnected components. Similarly, with σ less than $2n$ there must be some disconnected modules.

Suppose there are no modules of circuit degree one. Therefore, all modules are of circuit degree two. It has been shown that a path of modules is a simple circuit. If all modules are of circuit degree two, then we have the equivalent of a path of modules that is connected at its endpoints (module m_1 is connected to m_k) (or two or more such circuits of modules). This is topologically equivalent to taking a simple circuit, breaking it into two disjoint paths, and connecting each path to itself, thus creating two disjoint circuits.

We have exhausted the possibilities, therefore, every orthogonal simple circuit must be a path of modules. □

Finally, the preceding lemmas lead to the following conclusion.

THEOREM. OSC *is* NP-*complete.*

Proof. It is routine to show that OSC is in NP. If we are given a set of orthogonal segments with a set of augmenting segments, then the existence of a simple orthogonal circuit can be checked in linear time. We have shown that given a planar cubic graph G, we can construct a collection of modules $M(G)$, such that there is a Hamiltonian path in G, if and only if there is an orthogonal simple circuit in $M(G)$. Therefore OSC is NP-complete. □

In the reduction that has just been given there are segments that intersect at their endpoints. It is not hard to convert the collection of modules into a set of disjoint horizontal segments. Simply remove the vertical edges of each module and place each module on a row of its own. Modules with these changes are shown in Fig. 7. By examination we can see that all the vertical segments that were removed are now forced in the new layout of horizontal segments.

It is worth noting at this point that a remarkably similar problem has a polynomial time solution. Suppose we are given a set of orthogonal line segments and we wish to determine if the segments admit an *alternating orthogonal simple circuit.* This restricts the resulting orthogonal simple circuit to having edges that alternate between horizontal and vertical. An algorithm due to O'Rourke [8] used to decide whether there is an alternating orthogonal simple circuit in a set of points that can be applied in a straightforward manner. This algorithm returns an orthogonal simple circuit, if it exists, in $O(n \log n)$ time. Furthermore, it is shown that if an alternating orthogonal simple

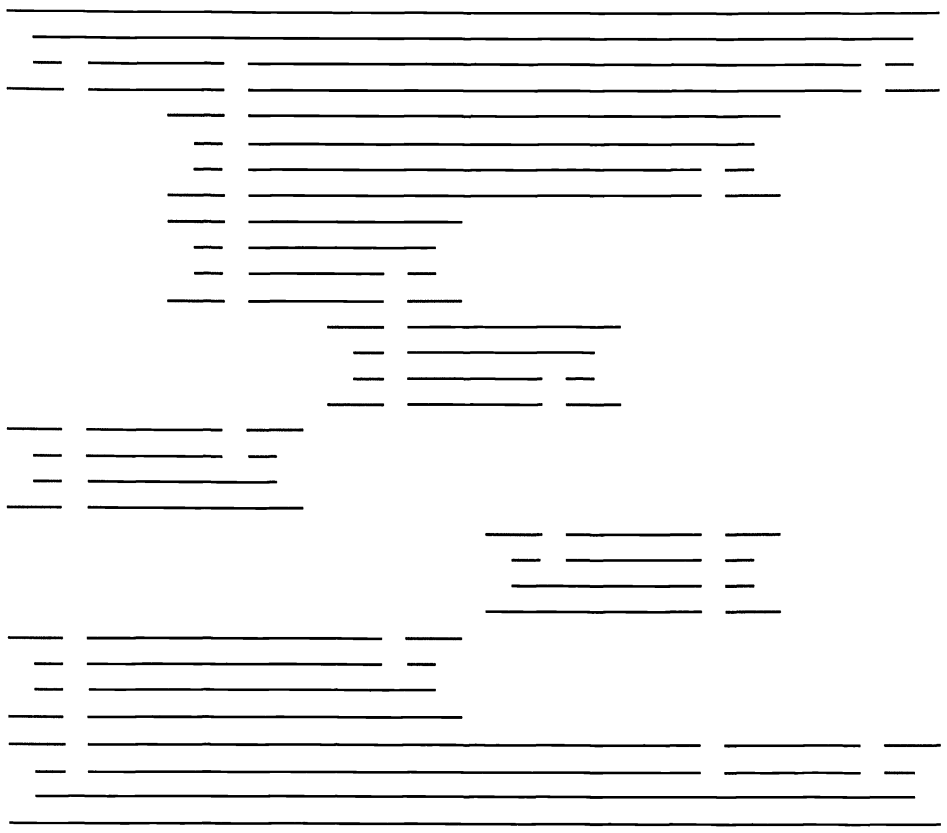


FIG. 7

circuit exists, then it is unique. Concerning (not necessarily alternating) orthogonal simple circuits of points, this problem has been shown to be NP-complete [11], [12].

4. Simple circuit is NP-complete. In this section it will be shown that the OSC problem, for a set of modules obtained from an instance of HPPCG, polynomially transforms to SC. Following the strategy taken in the previous section, we will build a collection of modules out of line segments. As a distinguishing feature we will denote the modules described in this section as SC modules, and those of the previous section as OSC modules.

SC modules are constructed in much the same way as OSC modules. Each module has an inner and outer rectangular frame with doors on the inner and outer frames. Shown in Fig. 8 is a collection of SC modules corresponding to OSC modules in Fig. 3, and ultimately to the graph in Fig. 2. In SC modules greater care must be taken to limit the visibility of the doorjambs. For this reason the doors are recessed. A possible solution is to let each door in an SC module be one unit wide, in a recessed three-unit enclosure that is two units deep. See Fig. 8. This limits the field of view of any doorjamb to a 45° angle. Let D be the distance between the top of the lowest module and the bottom of the highest module. If h is the total number of rows in the rectilinear planar layout (h is bounded by n, the number of vertices in G), and each module is w units wide (SC modules are constructed with a width of 10 units), with a one-unit space between rows of modules, then $D=(h-2)(w+1)+1$. DOORS are spaced so that the distance between them is at least D. The limited field of view ensures that a doorjamb can only see doorjambs of its neighbour. Another feature found in SC modules, and not in OSC modules, is the obstacle that runs the length of every module. These obstacles ensure that the visibility between inner and outer doorjambs remains local. The details concerning the construction of SC modules are omitted, as they are quite tedious, but given the above informal description it is a routine matter to construct the SC modules in $O(n)$ time.

A correspondence between the candidates of an OSC module and an SC module will be established. Since there are some candidates (endpoints that see each other) in SC modules that do not exist in OSC modules, it will be necessary to introduce the notion of a *useful candidate.* A candidate is useful if it can eventually appear as an augmenting segment. As will be shown, there are some candidates in SC modules that are not useful. However, all useful candidates in SC modules correspond exactly to the useful candidates of an OSC module.

Referring to Fig. 8, there are candidates in OSC modules that cross. For example the candidates $(5, 8)$ and $(6, 7)$. It will be shown that all these crossing candidates are not useful. Since the candidates found at every door of every module are equivalent, it is sufficient to examine a single door. Referring to Fig. 8, the inclusion of $(5, 8)$ forces $(6, 7)$, a crossing. Similarly $(6, 7)$ forces $(5, 8)$, a crossing. Therefore, $(6, 7)$ and $(5, 8)$ can never be augmenting segments. Including $(7, y)$ forces either $(8, x)$, $(8, 6)$, or $(8, 5)$. With $(8, x)$ we get a crossing, with $(8, 6)$ we isolate 5, and $(8, 5) = (5, 8)$ has been shown to be forbidden. As is the case for $(7, y)$, $(8, x)$ can never appear in a simple circuit. Therefore, $(5, 8)$, $(6, 7)$, $(7, y)$, and $(8, x)$ cannot be augmenting segments.

We can now conclude that the list of useful candidates in OSC modules is identical to those in SC modules. It is not hard to see that OSC modules are topologically equivalent to SC modules. Therefore we have the following theorem.

THEOREM. *There is an orthogonal simple circuit in* OSC *modules if and only if there is a simple circuit in the corresponding SC modules.*

The main result of the paper can now be proved.

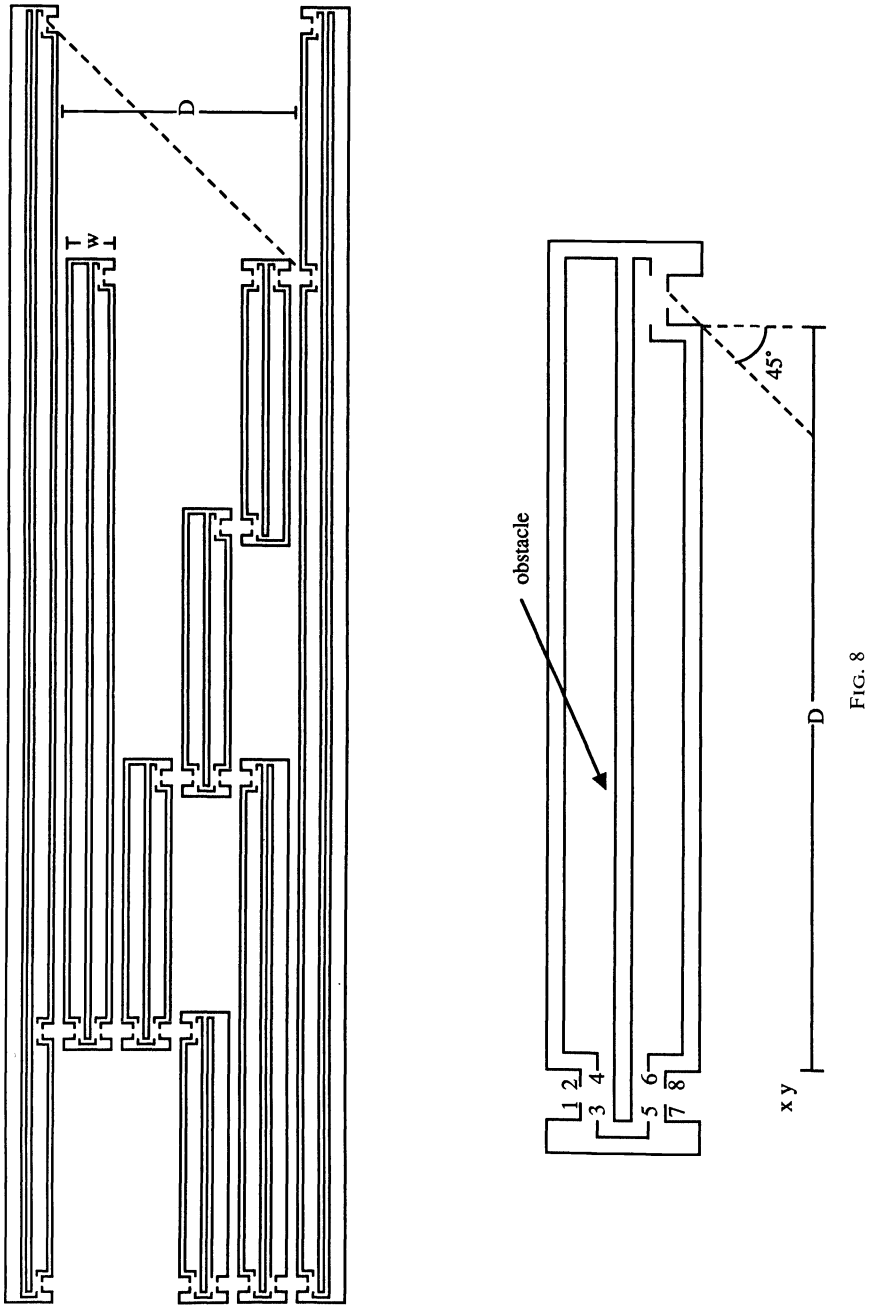


FIG. 8

THEOREM. SC *is* NP-*complete.*

Proof. It is routine to verify that SC is in NP. Given an instance of HPPCG, the graph G, we can construct a set of SC modules $M(G)$, so that there is a simple circuit from $M(G)$ if and only if there is a Hamiltonian path in G. Therefore SC is NP-complete. □

In the previous section it has been shown that OSC modules can be built from individual disjoint line segments. This does not appear to be the case for the SC modules described here. It remains an open problem to determine if SC is NP-complete even if we insist that the closed intervals of the segments are disjoint.

Observe that all the segments used in this reduction for SC are orthogonal. Therefore, SC is NP-complete for orthogonal segments.

5. Discussion. It has been shown that deciding whether a set of line segments admits a simple circuit is NP-complete. It is somewhat annoying that the result requires that some of the line segments must intersect at their boundaries. It would be more satisfying if this restriction could be removed. The reduction used in this paper is to a problem that is not far removed from a geometric setting. Using the rectilinear planar layout of planar cubic graphs is the major conceptual step of the result. Perhaps what is needed to prove a stronger result is a reduction to a more "primitive" NP-complete problem. A common approach used in other geometric NP-complete and NP-hard results has been to reduce the problem to the Planar 3-SAT problem [6], [10], [15]. It may be that this strategy would bear fruit in realizing an NP-completeness proof for disjoint segments. On the other hand, it may be (this appears to be the more exciting outcome) that the problem is not NP-complete at all and a polynomial solution exists!

Acknowledgments. I thank David Avis and Rafe Wenger for reading and commenting on earlier drafts of this paper.

REFERENCES

[1] D. AVIS AND D. RAPPAPORT, *Computing monotone simple circuits in the plane*, in Computational Morphology, G. T. Toussaint, ed., Elsevier Science Publishers B.V., North-Holland, 1988 pp. 13–23.

[2] J. D. BOISSONNAT, *Geometric structures for three-dimensional shape representation*, ACM Trans. Graphics, 3 (1984), pp. 266–286.

[3] M. R. GAREY, D. S. JOHNSON, AND R. E. TARJAN, *The planar Hamiltonian circuit problem is* NP-*complete*, SIAM J. Comput., 5 (1976), pp. 704–714.

[4] A. ITAI, C. H. PAPADIMITRIOU, AND J. L. SZWARCFITER, *Hamiltonian paths in grid graphs*, SIAM J. Comput., 11 (1982), pp. 676–686.

[5] E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOOY KAN, AND D. B. SHMOYS, *The Traveling Salesman Problem*, John Wiley, New York, 1985.

[6] D. T. LEE AND A. K. LIN, *Computational complexity of art gallery problems*, IEEE Trans. Inform. Theory, 32 (1986), pp. 276–282.

[7] D. MARR, *Vision: A Computational Investigation into Human Representation and Processing of Visual Information*, W. H. Freeman, San Francisco, CA, 1982.

[8] J. O'ROURKE, *Uniqueness of orthogonal connect-the-dots*, in Computational Morphology, G. T. Toussaint, ed., Elsevier Science Publishers B.V., North-Holland, 1988, pp. 97–104.

[9] J. O'ROURKE, H. BOOTH, AND R. WASHINGTON, *Connect-the-dots: a new heuristic*, Comput. Vision Graphics Image Process., 39 (1987), pp. 258–266.

[10] J. O'ROURKE AND K. SUPOWIT, *Some* NP-*hard polygon decomposition problems*, IEEE Trans. Inform. Theory, 29 (1983), pp. 181–190.

[11] D. RAPPAPORT, *On the complexity of computing orthogonal polygons from a set of points*, Tech. Report TR-SOCS-86.9, McGill University, Montreal, Quebec, Canada, April 1986.

[12] ———, *The complexity of computing simple circuits in the plane*, Ph.D. dissertation, McGill University, Montreal, Quebec, Canada, December 1986.

[13] D. RAPPAPORT, H. IMAI, AND G. T. TOUSSAINT, *Computing simple circuits from a set of segments*, Discrete Comput. Geom., to appear.

[14] P. ROSENSTIEHL AND R. E. TARJAN, *Rectilinear planar layouts of planar graphs and bipolar orientations*, Discrete Comput. Geom., 1 (1986), pp. 343–353.

[15] K. J. SUPOWIT, *Topics in computational geometry*, Ph.D. dissertation, University of Illinois, Urbana-Champaign, IL, 1981.

[16] G. TOUSSAINT, *Pattern recognition and geometrical complexity*, in Proc. 5th International Conference on Pattern Recognition, Miami Beach, FL, December 1980, pp. 1324–1347.

[17] ———, *Computational geometry and morphology*, in Proc. First International Symposium for Science on Form, Tsukuba, Japan, November 1985, pp. 217–248.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1140-1148, December 1989

005

THE TWO-PROCESSOR SCHEDULING PROBLEM IS IN RANDOM NC*

UMESH V. VAZIRANI† AND VIJAY V. VAZIRANI‡

Abstract. An efficient parallel algorithm (RNC^2) for the two-processor scheduling problem is presented. An interesting feature of this algorithm is that it finds a highest-level-first schedule: such a schedule defines a lexicographically first solution to this problem in a natural way. A key ingredient of the algorithm is a generalization of a theorem of Tutte which establishes a one-to-one correspondence between the bases of the Tutte matrix of a graph and the sets of matched nodes in maximum matchings in the graph.

Key words. scheduling, matching, parallel algorithms, randomized algorithms, basis of a matrix

AMS(MOS) subject classifications. 15A03, 15A15, 68Q25, 68R05, 68R10

1. Introduction. We present an efficient parallel RNC^2 algorithm for the following well-studied problem. Schedule n unit length jobs subject to given precedence constraints on two identical processors so as to minimize the finish time. Our algorithm uses $O(n^2M(n))$ processors, where $M(n)$ is the number of arithmetic operations required to multiply two $n \times n$ matrices.

There are close connections between two-processor scheduling and matching. The first polynomial time sequential algorithm for two processor scheduling relied on this connection [FKN], and in fact, techniques for finding matchings in parallel play an important role in the algorithm presented in this paper. The recent parallel matching algorithms [KUW], [MVV] are based on an algebraic connection established by a theorem of Tutte [Tu]. We present a generalization of this theorem, establishing a one-to-one correspondence between the bases of the Tutte matrix of a graph and the sets of matched nodes in maximum matchings in the graph. This yields an RNC^2 algorithm for computing the lexicographically largest such node-set. This subroutine is central to the parallel scheduling algorithm. The generalization is also of independent interest, and has been used in [MVV] to obtain an RNC^2 algorithm for maximum vertex-weighted matching.

The parallel scheduling algorithm finds a special type of schedule called a highest-level-first (HLF) schedule; a basic theorem of Gabow [Ga1], [Ga2] establishes the optimality of such a schedule. In § 5 we show that HLF schedules are "well behaved" by proving two new properties of such schedules. Using these properties, the task of computing an HLF schedule is shown to be reducible to parallel invocations of the lexicographically first node-set subroutine.

It is pointed out in § 4 that an HLF schedule is the lexicographically first schedule under a certain natural ordering on the jobs. In general, the parallel complexity of a problem is very sensitive to whether an arbitrary solution is acceptable or whether the lexicographically first solution is sought. Two important cases illustrate this point:

* Received by the editors January 23, 1987; accepted for publication (in revised form) March 24, 1989. This research was partially done while the authors were visiting the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, during the summer of 1984.

† Department of Computer Science, University of California at Berkeley, Berkeley, California 94720. This author is currently supported by a Presidential Young Investigator Award. This research was partially supported by National Science Foundation grant 82-04506 and an IBM Doctoral Fellowship.

‡ Department of Computer Science, 4130 Upson Hall, Cornell University, Ithaca, New York 14853-7501. This author is currently being supported by a Presidential Young Investigator Award, with matching funds from AT&T Bell Laboratories and Sun Microsystems Inc. This research was also partially supported by an IBM Faculty Development Award.

maximal independent set and depth-first search. The problems of finding the lexicographically first maximal independent set or the lexicographically first depth-first search tree are P-complete. Nevertheless, if an arbitrary solution is acceptable, in each case fast parallel algorithms exist. In the case of the maximum matching problem, fast parallel algorithms are known when an arbitrary solution suffices, but the parallel complexity of finding the lexicographically first maximum matching is still unresolved. Using the connection between two-processor scheduling and matching, the parallel algorithm presented in this paper can be interpreted as an algorithm for computing lexicographically first matchings for incomparability graphs under a certain natural ordering on the edges.

The algorithm presented here is randomized in the Las Vegas sense: it always produces the correct answer, but the guarantee on the running time is probabilistic. A preliminary report on this algorithm has appeared in [VV]. Subsequently, Helmboldt and Mayr [HM] have devised an algorithm that avoids the use of randomization. Their algorithm runs in $O(\log^2 n)$ time and uses $O(n^7 L^2)$ processors, where L is the length of the optimal schedule. Their algorithm shares some common features with ours; the proof of correctness of their algorithm relies on the "well-behavedness" properties of HLF schedules that are implicitly assumed in their paper, and explicitly proved here.

There is extensive literature on the two-processor scheduling problem (see [LR] for a general reference). We will use results from [FKN], [Ga1], and [Ga2].

2. Connection with matchings. We denote the set of jobs by V, and the precedence relations among the jobs by a directed acyclic graph $G(V, E)$.

DEFINITION. A *schedule* is a sequence of sets $S_1, \cdots, S_t$ that partition V, with $|S_i| \leqq 2$ and such that if x precedes y, $x \in S_i$, and $y \in S_j$ then $i < j$. The length of the schedule is t.

It is useful to associate a compatibility graph $H(V, E')$ with the given precedence graph $G(V, E)$. $H(V, E')$ is an undirected graph with an edge between a pair of vertices x and y if they are compatible, i.e., there is no directed path from x to y or y to x in G. The following theorem gives the fundamental relationship between matchings and schedules.

THEOREM 1 (Fujii, Kasami, and Ninomiya [FKN]). *For G and H as defined above, the paired jobs in any optimal schedule for G form a maximum matching in H. Moreover, any maximum matching in H can be transformed into an optimal schedule for G, leaving the set of unpaired jobs unchanged.*

Note that not all maximum matchings in H correspond to valid schedules for G; for a maximum matching to correspond to a valid schedule, it should be possible to linearly order the pairs so that all precedence constraints are satisfied. The content of the above theorem of [FKN] is a sequential procedure to transform any maximum matching into a valid schedule without changing the set of unpaired jobs.

The connection between matchings and schedules goes further. In § 5 we show that the lexicographically first maximum matching (under a natural ordering on the edges of H) always corresponds to an optimal schedule. The parallel algorithm presented in this paper finds an optimal schedule by constructing precisely such a lexicographically first matching.

3. The algebraic connection. The results of this section are of independent interest; in this section, we will let $G(V, E)$ denote an undirected graph, with $|V| = n$.

DEFINITION. The determinant of matrix A will be denoted by $\det(A)$, and its (i, j)th entry will be denoted by a_{ij}. The adjacency matrix D of graph $G(V, E)$ is given

by

$$d_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

The Tutte matrix A of graph $G(V, E)$ is given by

$$a_{ij} = \begin{cases} x_{ij} & \text{if } (v_i, v_j) \in E \text{ and } i > j, \\ -x_{ij} & \text{if } (v_i, v_j) \in E \text{ and } i < j, \\ 0 & \text{otherwise.} \end{cases}$$

THEOREM 2 (Tutte [Tu]). *Let A be the Tutte matrix of a graph $G(V, E)$. Then, $\det(A) \not\equiv 0$ if and only if there exists a perfect matching in G.*

The following is a natural generalization of Tutte's theorem to maximum matchings.

THEOREM 3 (Lovász [LP]). *Let A be the Tutte matrix of a graph $G(V, E)$, and let m be the number of edges in a maximum matching in G. Then,* $\operatorname{rank}(A) = 2m$.

Rabin and Vazirani [RV] have given a simple proof of Theorem 3 using a theorem of Fröbenius [Ko, p. 144] stated below. We use their approach to give a further generalization below.

DEFINITION. The *node-set* of a maximum matching M is an n-dimensional 0/1 vector whose ith component is 1 if and only if vertex v_i is matched in M. Let n_G denote the lexicographically first such node-set.

Notation. We represent a basis of an $n \times n$ matrix A as an n-dimensional 0/1 vector, whose ith component is one if and only if the ith row of A is in the basis. Given n-dimensional vectors u, v, A_{uv} is the submatrix of A obtained by choosing rows corresponding to indices having 1's in u, and columns corresponding to indices having 1's in v. Similarly, G_u is the subgraph of G induced on vertices corresponding to indices having 1's in u. We will denote the number of 1's in u by $\#u$.

THEOREM 4 (Fröbenius [Ko, p. 144]). *Let B be an n by n skew-symmetric matrix. Let u, v be n-dimensional vectors such that $\#u = \#v = \operatorname{rank}(B)$. Then*

$$\det(B_{uu}) \cdot \det(B_{vv}) = (-1)^{\#u} \det(B_{uv})^2.$$

THEOREM 5. *Let A be the Tutte matrix of a graph $G(V, E)$. A vector $w \in \{0, 1\}^n$ is a basis of A if and only if w is the node-set of a maximum matching in G.*

Proof. Let w be the node set of a maximum matching in G. Since G_w has a perfect matching, by Tutte's theorem $\det(A_{ww}) \not\equiv 0$. Thus the rows of A indexed by w are linearly independent. Moreover, by Theorem 3 $\operatorname{rank}(A) = \#w$. Thus w is a basis of A.

Conversely, let w be a basis of A. Then, there is a vector $v \in \{0, 1\}^n$ such that $\#v = \#w$, and $\det(A_{wv}) \not\equiv 0$. Since $\operatorname{rank}(A) = \#w = \#v$, by Fröbenius' theorem $\det(A_{ww}) \not\equiv 0$. Now, by Tutte's theorem G_w has a perfect matching, and by Theorem 3 this matching is a maximum matching in G. Hence w is the node-set of a maximum matching in G. □

COROLLARY. *The lexicographically first basis of A is n_G.*

THEOREM 6. *There is an* RNC^2 *algorithm for finding n_G. It requires $(M(n) \log^2 n)$ processors.*

Proof. Choose a random substitution S: set of variables in $A \to [1, \cdots, 2n]$ to obtain the matrix A^S. Certainly, any basis of A^S is a basis of A. Let $w \in \{0, 1\}^n$ be the lexicographically first basis of A. Then, $\det(A_{ww}) \not\equiv 0$. This is a nonvanishing polynomial of degree at most n. Hence, by Schwartz's theorem [Sc], the probability that this

polynomial vanishes under the random substitution S is:

$$\Pr[\det(A_{ww}^S) = 0] \leqq \frac{\deg(\det(A_{ww}))}{2n} \leqq \frac{1}{2}.$$

Therefore, with probability at least $\frac{1}{2}$, A^S has the same lexicographically first basis as A. We now use the RNC^2 algorithm of Borodin, van zur Gathen, and Hopcroft [BGH] for finding the lexicographically first basis of A. For better processor efficiency, the computations can be done over a finite field (see [RV] for details). This requires $O(M(n)\log^2 n)$ processors, where $M(n)$ is the number of arithmetic operations required to multiply two $n \times n$ matrices.

4. Highest-level-first schedules. As before, let V be the set of jobs to be scheduled, and let $G(V, E)$ be a directed acyclic graph that represents the precedence constraints among the jobs.

DEFINITION. The *level* of a job is one plus the length of the longest directed path in G starting at the job.

Note that all constraints go from higher (numbered) levels to lower levels, and there are no constraints among jobs that are at the same level. Assume that the levels are numbered $L, L-1, \cdots, 1$.

DEFINITION. A schedule is a *level schedule* if for every level i, there is at most one job at level i paired with a job at a lower level, say j. Such a pair (if it exists) is called a *jump*, and the jump of level i is said to be j. If all jobs at level i are paired with jobs at level i or larger, then its jump is i, and if a job remains unpaired at level i, then its jump is 0.

THEOREM 7 (Gabow [Ga1]). *Any optimal schedule on a dag $G(V, E)$ can be transformed into a level schedule, without changing the set of unpaired jobs.*

DEFINITION. The *jump-sequence* of a level schedule is an L-tuple $(p_L, p_{L-1}, \cdots, p_1)$ where p_i is the jump of level i.

The lexicographic ordering on jump-sequences induces a partial ordering on level schedules.

DEFINITION. A level schedule is *highest-level-first* (HLF) if its jump sequence is lexicographically largest among all valid schedules for G.

Intuitively, such a schedule always jumps to the highest level possible. Moreover, within this level, it jumps to a job that allows subsequent jumps to be highest possible. The importance of such a schedule comes from the following theorem.

THEOREM 8 (Gabow [Ga1]). *Any highest-level-first schedule on a dag $G(V, E)$ is optimal.*

Remark. Let us assign to each edge in the compatibility graph the ordered pair (h, l) where h is the level of the higher endpoint of the edge and l is the level of the lower endpoint. Assign priorities to the edges by comparing the associated ordered pair; note that edges that run between the same levels have the same priority. Now matchings can be compared lexicographically in the obvious manner. It is easy to check that the paired jobs of any HLF schedule form a lexicographically highest matching in this sense.

5. Further connections between matchings and HLF schedules. In this section we prove two new properties of HLF schedules; these properties show that HLF schedules are "well behaved," and are essential to proving the correctness of the parallel algorithm. The first property (proved in Theorem 9) states that HLF schedules leave unpaired jobs at the lowest possible levels. The second (proved in Theorem 10) states that restricting an HLF schedule to jobs at levels i or greater yields an HLF schedule for the resulting dag.

DEFINITION. The coarse-jump-sequence of a level schedule S is an L-dimensional 0/1 vector $(s_L \cdots s_1)$ such that s_i is 0 if and only if S leaves a job unpaired on level i. Note that if S is a level schedule with jump sequence $(p_L, \cdots, p_1)$, its coarse-jump-sequence is given by $s_i = 0$ if and only if $p_i = 0$.

DEFINITION. The *level-set* of a maximum matching M in H is an L-dimensional 0/1 vector whose ith component is 0 if and only if there is an unmatched vertex at level i. Let l_H denote the lexicographically largest level-set. Recall that n_H is the lexicographically first node-set of a maximum matching in H.

THEOREM 9. *The coarse-jump-sequence of an* HLF *schedule is lexicographically greater than or equal to the coarse-jump-sequence of any level schedule on G.*

Proof. The proof is by contradiction. Suppose S is an HLF schedule on G with coarse-jump-sequence s, and T is a level schedule whose coarse-jump-sequence t is lexicographically larger than s. Let i be the highest level at which t and s differ; then S has an unpaired job at level i and T does not. Let k be the number of unpaired jobs in T at levels lower than i, and l the number of unpaired jobs in T at levels higher than i.

Obtain a new dag G' from G as follows. Add k new jobs at the first level; put in precedence constraints from each job at level i or higher to each new job. Let S' be an HLF schedule on G'. Note that any level schedule for G when restricted to levels i or higher (including jumps from these levels to lower levels) is valid for G', and vice versa. Since S and S' are HLF in G and G', respectively, their jump sequences must agree on level $\geqq i$. Therefore S' has at least $l+1$ unpaired jobs. On the other hand, T can be modified into a schedule for G' with only l unpaired jobs. This contradicts the optimality of S'.

LEMMA 1. *The* HLF *coarse-jump-sequence in G is the same as* l_H.

Proof. Let M be a maximum matching in H whose level-set is l_H. By Theorems 1 and 7 there is a level schedule for G having the same unpaired vertices as M, and therefore its coarse-jump-sequence is also l_H. The lemma now follows from Theorem 9.

Computing the HLF unpaired jobs in G has now been reduced to finding l_H. In Lemma 2 we will reduce this to finding n_G, for which an RNC^2 algorithm has been given in § 3. Note that if a maximum matching M achieves node-set n_H, then M leaves at most one vertex unmatched at each level. It is therefore straightforward to obtain the level-set of M.

LEMMA 2. *Pick any ordering* $>$ *on the vertices of H such that if* level (x) *is higher than* level (y) *then* $x > y$. *Under this ordering, the level-set of any maximum matching M achieving node-set* n_H *is* l_H.

Proof. Let M be a maximum matching achieving node-set n_H for this ordering, and let N be a maximum matching with level-set l_H. Since both matchings are maximum, the symmetric difference of M and N contains even cycles and even length paths. Each even length path must have its endpoints at the same level, otherwise we could improve either M or N. Hence M and N have their unmatched nodes at the same levels. □

DEFINITION. Let $G(V, E)$ be a dag, partitioned into L levels. For $1 \leqq i \leqq L$, define G_i to be the dag induced on vertices belonging to levels $L, L-1, \cdots, i$. For convenience, we number the levels of G_i as $L, L-1, \cdots, i$.

THEOREM 10. *Let the* HLF *jump-sequence for G be* $(p_L, p_{L-1}, \cdots, p_1)$. *Then, for* $1 \leqq i \leqq L$, *the* HLF *jump-sequence for* G_i *is* $(q_L, q_{L-1}, \cdots, q_i)$, *where*

$$q_j = \begin{cases} p_j & \text{if } p_j \geqq i, \\ 0 & \text{otherwise.} \end{cases}$$

We follow the method of Gabow for modifying schedules. First we introduce the relevant definitions and lemmas from [Ga1].

DEFINITION. A *k-schedule* is a level schedule for levels $L, \cdots, k$ and nodes in lower levels jumped from these levels. (A k-schedule is just a prefix of a complete schedule.)

DEFINITION. Let S be a k-schedule, with level $(x) >$ level (y). Say that y is *x-ready* in S if x does not precede y, and all predecessors of y below level (x) are jumped from above level (x).

Let M be an HLF k-schedule, and let O be a k-schedule that is HLF up to level $k+1$.

LEMMA 3 (Gabow [Ga1]). *If (x, y) is a jump in O with* level $(x) > k$, *then y is x-ready in M.*

DEFINITION. A *chain* consists of nodes x_i, $1 \leqq i \leqq c+1$, and y_i, $0 \leqq i \leqq c$ for $c \geqq 1$, where

(1) (x_i, y_i) is a jump in O, for $1 \leqq i \leqq c$.

(2) (y_i, x_{i+2}) is a jump in M, for $0 \leqq i \leqq c-1$.

(3) level (x_i) = level (y_{i-1}), for $1 \leqq i \leqq c+1$.

Adapting Theorem 2.1 of [Ga1] to our setting, we obtain Lemma 4.

LEMMA 4 (Gabow [Ga1]). *Let M and O be as above; consider a chain with* level (x_1) *as high as possible, and with c maximal with* level $(x_{c+1}) \geqq k$. *Let x_1 be paired with u in M and y_0 be paired with x_0 in O; clearly,* level (u) = level (y_0) = level (x_0) = level (x_1).

(1) *If there is no jump from x_{c+1} in O, then O can be modified into a valid k-schedule by replacing (x_i, y_i) by (y_{i-1}, x_{i+1}) for $i = 1$ to c, and by replacing (x_0, y_0) and (x_{c+1}, y_{c+1}) with (x_0, x_1) and (y_{c+1}, y_c).*

(2) *If there is no jump from y_c in M, then M can be modified into a valid k-schedule by replacing (y_{i-1}, x_{i+1}) with (x_i, y_i) for $i = 1$ to c, and by replacing (u, x_1) and (v, y_c) with (u, y_0) and (v, x_{c+1}).*

(3) level (x_{c+1}) = level $(y_c) = k$, *and (x_{c+1}, y_{c+1}) is a jump in O and (y_c, x_{c+2}) is a jump in M. Then O can be modified into a valid k-schedule, by replacing (x_i, y_i) with (y_{i-1}, x_{i+1}) for $i = 1$ to $c+1$, and by replacing (x_0, y_0) with (x_0, x_1).*

Proof of Theorem 10. Let S' be an HLF schedule for G_i, and let $(r_L, r_{L-1}, \cdots, r_i)$ be its jump-sequence. Assume for contradiction that $(r_L, r_{L-1}, \cdots, r_i)$ is larger than $(q_L, q_{L-1}, \cdots, q_i)$. Let k be the highest level where the two jump-sequences differ.

Choose k-schedules M and O for $G(V, E)$:

(1) M has jump sequence $(p_L, p_{L-1}, \cdots, p_k)$.

(2) O has jump sequence $(p_L, \cdots, p_{j+1}, r_j, \cdots, r_k)$.

(3) j is as small as possible.

(4) Among pairs of k-schedules satisfying (1), (2), and (3), M and O have the largest number of common jumps.

If $j = k$ then O has a larger jump-sequence than M, thus contradicting the fact that M is HLF. Otherwise, pick jumps (x_1, y_1) in O and (y_0, x_2) in M, with level (x_1) = level (y_0) as high as possible, and $(x_1, y_1) \neq (y_0, x_2)$. Let these jumps be the start of a chain as defined above with c maximal. We first prove that level $(x_c) \geqq j$. Assume for contradiction that level $(x_c) < j$. Let level $(x_d) > j >$ level $(x_{d+1}) \geqq k \geqq i$. Let x be the job that jumps from level j in M. By Lemma 3, x_{d+1} is y_{d+1}-ready in M, and therefore x-ready in M. Thus M can be modified by jumping x to y_{d-1}, thus contradicting the fact that M is HLF.

Now let M' and O' denote j-schedules corresponding to M and O, respectively. Then M' and O' satisfy the conditions of Lemma 4. Therefore either M' or O' can be

modified (according to which of the three cases applies to the chain defined above) to increase the number of common jumps or to make O' jump to level p_j from level j. The new pair of j-schedules can be extended to k-schedules, thus contradicting condition (3) or condition (4) above.

COROLLARY. $\max\{i : jth\ component\ of\ the\ \text{HLF}\ jump\text{-}sequence\ for\ G_i\ is\ nonzero\} = p_j$.

Say that a jump-sequence is *realizable* if there is a level schedule for G with this jump-sequence. Given a jump-sequence, we show how to construct a graph $K(V', E')$ which has a perfect matching if and only if the jump-sequence is realizable, and if so each perfect matching reveals the set of jumps of one such schedule.

$V' = V \cup V_1 \cup \cdots V_L$, where L is the number of levels in G, and the V_i's are disjoint from each other and from V.

Let $l(i)$ denote the number of vertices at level i;

Let $k(i)$ = number of levels that jump to level i;

If $0 < \text{jump}(i) < i$, then $|V_i| = l(i) - k(i) - 1$, and otherwise $|V_i| = l(i) - k(i)$.

$(x, y) \in E'$ if either

(1) $x, y \in V$, with level $(x) = i$ and level $(y) = j$, jump $(i) = j$, $0 < j < i$, and x is not a predecessor of y (these will be called the type 1 edges), or

(2) $x \in V$, with level $(x) = i$, and $y \in V_i$ (these will be called the type 2 edges).

LEMMA 5. *H has a perfect matching if and only if the given jump sequence is realizable. Moreover type* 1 *edges in the perfect matching are a valid realization of the jump sequence (with the exception of jumps to level* 0).

Proof. If the jump sequence is realizable, then H has the following perfect matching. Match a type 1 edge only if it is a jump. This leaves exactly the correct number of vertices at level i to be matched with vertices in V_i.

We prove that the type 1 edges in the perfect matching realize the jump sequence by induction on L. This is true for the Lth level by the choice of $|V_L|$. Removing V_L and the Lth level of V, we obtain a graph with $L-1$ partitions, and the proof follows by induction. □

6. The parallel algorithm. Procedure COARSE-JUMP-SEQUENCE below computes the coarse-jump-consequence of an HLF schedule for the given directed graph $G(V, E)$. Procedure JUMP-SEQUENCE computes the HLF jump-sequence for $G(V, E)$. Procedure JUMPS computes the jumps for some HLF schedule for $G(V, E)$. Once the jumps are known, it is an easy matter to pair up the remaining jobs to get an HLF schedule.

PROCEDURE COARSE-JUMP-SEQUENCE.

1. In parallel topologically sort G to assign levels.
2. Assign distinct numbers to the vertices of G, with $\#v > \#w$ if level $(v) >$ level (w).
3. Obtain the compatibility graph H.
4. In parallel compute n_H (under the above numbering).
5. Compute l_H from n_H and output it.

end.

PROCEDURE JUMP-SEQUENCE.

For $j = 1$ to L do in parallel:

Find the HLF coarse-jump-sequence for G_j.

end;

For $j = 1$ to L do in parallel:

jump $(j) = \max \{i : i \leqq j \wedge j$th component of the HLF coarse-jump-sequence for G_i is one$\}$.
output jump (j);
end;
end procedure;

PROCEDURE JUMPS $[G]$.
Compute the HLF jump-sequence for G;
Construct an undirected graph K for this jump sequence as in Lemma 6.
Find a perfect matching in H;
Output the type 1 edges in the perfect matching.
end procedure;

THEOREM 11. There is an RNC^2 algorithm that finds an optimal two-processor schedule (in particular, an HLF schedule) in a dag $G(V, E)$.

Proof. The correctness of procedure COARSE-JUMP-SEQUENCE follows from Theorem 9 and Lemmas 1 and 2; the correctness of procedure JUMP-SEQUENCE follows from the corollary to Theorem 10. The correctness of procedure JUMPS follows from Lemma 5. Procedure JUMPS determines the set of jumps of an HLF schedule (except the jumps to level 0). Now, if a level has an even number of jobs remaining, they can be paired off, and if it has an odd number of jobs remaining, one job will remain unpaired. The jobs can also be assigned appropriate time slots.

Each of the procedures can be implemented in RNC^2, yielding an RNC^2 algorithm overall. Two computations require the most number of processors: computing the ranks of n matrices in parallel, and finding a perfect matching. The second task dominates the first, since in the first task, the computations can be done modulo a prime, as elaborated in [RV] (see Theorem 6). The graph H in which a perfect matching is found has $O(n)$ vertices and $O(n^2)$ edges. This requires $O(n^2 M(N))$ processors using the matching algorithm of [MVV] that uses the matrix inversion algorithm of [Pa]. Here $M(n)$ denotes the number of arithmetic operations required to multiply two $n \times n$ matrices. The current best bound for $M(n)$ is $O(n^{2.376})$ [CW]. □

Using [FKN] and the RNC^2 algorithm of Karloff [Ka] for upper-bounding the size of maximum matching in a graph, we get an RNC^2 algorithm for upper-bounding the length of the optimal schedule. By running this algorithm and the scheduling algorithm in parallel, until they agree on the size of the optimal sequence, we get the following corollary.

COROLLARY. *There is a Las Vegas parallel algorithm that in expected time* $O(\log^2 n)$ *finds an optimal schedule in a dag* $G(V, E)$. *It runs on* $O(n^2 M(n))$ *processors.*

7. Discussion. An important issue left open in this paper is whether two-processor scheduling is in (deterministic) NC. Helmboldt and Mayr [HM] have resolved this question affirmatively. This result, together with the NC algorithm for transitively directing a comparability graph, yields a NC algorithm for constructing a maximum matching in an incomparability graph [KVV]. Both the parallel algorithm presented here as well as that due to Helmboldt and Mayr suffer from a large processor complexity—the parallel algorithm of this paper uses $O(n^{4.5})$ processors, and Helmboldt and Mayr's algorithm uses $O(n^7 L^2)$ processors. On the other hand, the problem can be solved sequentially in linear time [Ga2], [GT]. Improving the processor efficiency of the parallel algorithm remains an open problem.

A long-standing open problem in scheduling theory is to resolve the complexity of the k-processor scheduling problem, for fixed k, $k \geqq 3$. We conjecture that in contrast to the $k = 2$ case, for $k \geqq 3$, this problem is log-space hard for P.

Finally, the problem of finding an HLF schedule may be stated as the problem of finding the lexicographically largest maximum matching in an incomparability graph, with a partial ordering on the edges given by the level numbers of their endpoints (see the remark in § 4). The general problem of finding the lexicographically first maximum matching is open. Can some of the techniques of this paper be generalized to handle this larger problem?

Acknowledgments. The algorithm presented here benefited greatly from work done with Dexter Kozen on computing the interval representation of interval graphs in parallel. We wish to thank Dexter for his generous help throughout the evolution of this paper. We are also thankful to Ashok Chandra and Michael Rabin for several inspiring discussions.

REFERENCES

[BGH] A. BORODIN, J. VON ZUR GATHEN, AND J. E. HOPCROFT, *Fast parallel matrix and* GCD *computations*, Inform. Control, 52 (1982), pp. 241–256.

[CW] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual ACM Symposium on Theory of Computing, IEEE Computing Society, Washington, DC, 1987, pp. 1–6.

[FKN] M. FUJII, T. KASAMI, AND K. NINOMIYA, *Optimal sequencing of two equivalent processors*, SIAM J. Appl. Math., 17 (1969), pp. 784–789.

[Ga1] H. N. GABOW, *An almost-linear algorithm for two-processor scheduling*, Tech. Report CU-CS-169-80, University of Colorado, Boulder, CO, January, 1980.

[Ga2] ———, *An almost-linear algorithm for two-processor scheduling*, J. Assoc. Comput. Mach., 29 (1982), pp. 766–780.

[GT] H. N. GABOW AND R. E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Assoc. Comput. Mach., 30 (1985), pp. 209–221.

[HM] D. HELMBOLDT AND E. MAYR, *Two processor scheduling is in* NC, SIAM J. Comput., 16 (1987), pp. 747–759.

[Ka] H. KARLOFF, *A randomized parallel algorithm for the odd set cover problem*, Combinatorica, 6 (1986), pp. 387–391.

[Ko] G. KOWALEWSKI, *Einfuhrung in die Determinanten Theorie*, Leipzig Verlag von Veit und Comp. 1909, p. 144.

[KUW] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a maximum matching is in random* NC, Combinatorica, 6 (1986), pp. 35–48.

[KVV] D. KOZEN, U. V. VAZIRANI, AND V. V. VAZIRANI, NC *algorithms for comparability, interval graphs, and testing for unique perfect matchings*, in Proc. 5th Annual Foundations of Software Technology and Theoretical Computer Science Conferences, New Delhi, India, 1985.

[LP] L. LOVASZ AND M. D. PLUMMER, *Matching Theory*, Academic Press, Budapest, Hungary.

[LR] J. K. LENSTRA AND A. H. G. RINNOY KAN, *Complexity of scheduling under precedence constraints*, Oper. Res., 26 (1978), pp. 22–35.

[MVV] K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, *Matching is as easy as matrix multiplication*, Combinatorica, 7 (1987), pp. 105–113.

[Pa] V. PAN, *Fast and efficient algorithms for the exact inversion of integer matrices*, in Proc. 5th Annual Foundations of Software Technology and Theoretical Computer Science Conferences, New Delhi, India, 1985.

[RV] M. O. RABIN AND V. V. VAZIRANI, *Maximum matchings in general graphs through randomization*, J. Algorithms, 10 (1989), to appear.

[Sc] J. T. SCHWARTZ, *Fast polynomial algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[Tu] W. T. TUTTE, *The factorization of linear graphs*, J. London Math. Soc., 22 (1974), pp. 107–111.

[VV] U. V. VAZIRANI AND V. V. VAZIRANI, *The two-processor scheduling problem is in* R-NC, in Proc. 17th Annual Symposium on Theory of Computing, Providence, RI, 1985.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1149–1178, December 1989

006

APPROXIMATING THE PERMANENT*

MARK JERRUM† AND ALISTAIR SINCLAIR†

Abstract. A randomised approximation scheme for the permanent of a 0–1 matrix is presented. The task of estimating a permanent is reduced to that of almost uniformly generating perfect matchings in a graph; the latter is accomplished by simulating a Markov chain whose states are the matchings in the graph. For a wide class of 0–1 matrices the approximation scheme is fully-polynomial, i.e., runs in time polynomial in the size of the matrix and a parameter that controls the accuracy of the output. This class includes all dense matrices (those that contain sufficiently many 1's) and almost all sparse matrices in some reasonable probabilistic model for 0–1 matrices of given density.

For the approach sketched above to be computationally efficient, the Markov chain must be rapidly mixing: informally, it must converge in a short time to its stationary distribution. A major portion of the paper is devoted to demonstrating that the matchings chain is rapidly mixing, apparently the first such result for a Markov chain with genuinely complex structure. The techniques used seem to have general applicability, and are applied again in the paper to validate a fully-polynomial randomised approximation scheme for the partition function of an arbitrary monomer-dimer system.

Key words. permanent, perfect matchings, counting problems, random generation, Markov chains, rapid mixing, monomer-dimer systems, statistical physics, simulated annealing

AMS(MOS) subject classifications. 05C70, 05C80, 60J20, 68Q20

1. Summary.

The *permanent* of an $n \times n$ matrix A with 0–1 entries a_{ij} is defined by

$$\operatorname{per}(A) = \sum_{\sigma} \prod_{i=0}^{n-1} a_{i\sigma(i)},$$

where the sum is over all permutations σ of $[n] = \{0, \cdots, n-1\}$. Evaluating $\operatorname{per}(A)$ is equivalent to counting perfect matchings (1-factors) in the bipartite graph $G = (U, V, E)$, where $U = V = [n]$ and $(i, j) \in E$ if and only if $a_{ij} = 1$. The permanent function arises naturally in a number of fields, including algebra, combinatorial enumeration, and the physical sciences, and has been an object of study by mathematicians since first appearing in 1812 in the work of Cauchy and Binet (see [26] for background). Despite considerable effort, and in contrast with the syntactically very similar determinant, no efficient procedure for computing this function is known.

Convincing evidence for the inherent intractability of the permanent was provided in the late 1970s by Valiant [32], who demonstrated that it is complete for the class #P of enumeration problems, and thus as hard as counting *any* NP structures. Interest has therefore recently turned to finding computationally feasible approximation algorithms for this and other hard enumeration problems [18], [30]. To date, the best approximation algorithm known for the permanent is due to Karmarkar et al. [17] and has a runtime that grows exponentially with the input size.

The notion of approximation we will use in this paper is as follows. Let f be a function from input strings to natural numbers. A *fully-polynomial randomised approximation scheme* (fpras) for f is a probabilistic algorithm that, when presented with a string x and a real number $\varepsilon > 0$, runs in time polynomial in $|x|$ and $1/\varepsilon$ and outputs

* Received by the editors October 6, 1988; accepted for publication January 10, 1989. A preliminary version of this paper appeared in the Proceedings of the 20th ACM Symposium on Theory of Computing, Chicago, May 1988, under the title "Conductance and the Rapid Mixing Property for Markov Chains: The Approximation of the Permanent Resolved."

† Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, United Kingdom.

a number that with high probability approximates $f(x)$ within ratio $1+\varepsilon$.[1] For definiteness we take the phrase "with high probability" to mean with probability at least $\frac{3}{4}$. This may be boosted to $1-\delta$ for any desired $\delta > 0$ by running the algorithm $O(\lg \delta^{-1})$ times and taking the median of the results [16, Lemma 6.1]. An fpras therefore embodies a strong notion of effective approximation.

A promising approach to finding an fpras for the permanent was recently proposed by Broder [8], who reduces the problem of approximately counting perfect matchings in a graph to that of generating them randomly from an almost uniform distribution. The latter problem is then amenable to the following dynamic stochastic technique. Given a graph G, construct a Markov chain whose states correspond to perfect and "near-perfect" matchings in G and which converges to a stationary distribution that is uniform over the states. Transitions in the chain correspond to simple local perturbations of the structures. Then, provided convergence is fast enough, we can generate matchings almost uniformly by simulating the chain for a small number of steps and outputting the structure corresponding to the final state.

When applying this technique, we are faced with the thorny problem of proving that a given Markov chain is *rapidly mixing*, i.e., that after a short period of evolution the distribution of the final state is essentially independent of the initial state. "Short" here means bounded by a polynomial in the input size. Since the state space itself may be exponentially large, rapid mixing is a strong property: the chain must typically be close to stationarity after visiting only a very small fraction of the space.

Recent work on the rate of convergence of Markov chains has focused on stochastic concepts such as coupling [1] and stopping times [3]. While these methods are intuitively appealing and yield tight bounds for simple chains, the analysis involved becomes extremely complicated for more interesting processes that lack a high degree of symmetry. Using a complex coupling argument, Broder [8] claimed that the perfect matchings chain above is rapidly mixing provided the bipartite graph is *dense*, i.e., has minimum vertex degree at least $n/2$. This immediately implies the existence of an fpras for the dense permanent. However, the coupling proof is hard to penetrate; more seriously, as was first observed by Mihail [25], it contains a fundamental error that is apparently not correctable. As a result Broder has withdrawn his proof (see the erratum to [8]).

In this paper we propose a completely different approach to analysing the rate of convergence of Markov chains that is based on a structural property of the underlying weighted graph. Under fairly general conditions, a finite ergodic Markov chain is rapidly mixing if and only if the *conductance* of its underlying graph is not too small. This characterisation, discussed in detail in a companion paper [29], is related to recent work by Alon [4] and Alon and Milman [5] on eigenvalues and expander graphs.

While similar characterisations of rapid mixing have been noted by other authors (see, e.g., [2], [22]), they have been of little practical value because independent estimates of the conductance have proved elusive for nontrivial chains. Using a novel method of analysis, we are able to derive a lower bound on the conductance of Broder's perfect matchings chain under the same density assumption, thus verifying that it is indeed rapidly mixing. This is the first rapid mixing result we know of for a Markov chain with nontrivial structure. The existence of an fpras for the dense permanent is therefore established.

Reductions from approximate counting to almost uniform generation similar to that mentioned above for perfect matchings also hold for the large class of combinatorial

[1] For nonnegative real numbers a, $\tilde{a}$, ε, we say that $\tilde{a}$ *approximates* a *within ratio* $1+\varepsilon$ if $a(1+\varepsilon)^{-1} \leqq \tilde{a} \leqq a(1+\varepsilon)$.

structures that are *self-reducible* [16], [29]. Consequently, the Markov chain approach is potentially a powerful general method for obtaining approximation algorithms for hard combinatorial enumeration problems.

In fact, Markov chain simulation is a rather useful algorithmic tool with a variety of computational applications. Perhaps the most familiar of these are to be found in the field of statistical physics, where a physical system is typically modelled by a set of combinatorial structures, or *configurations*, each of which has an associated weight depending on its energy. Most interesting properties of the model can be computed from the *partition function*, which is just the sum of the weights of the configurations. An fpras for this function may usually be obtained with the aid of a generator that selects configurations with probabilities roughly proportional to their weights. This suggests looking for a Markov chain on configurations with the appropriate (nonuniform) stationary distribution. Such chains are in fact the basis of the ubiquitous Monte Carlo method of Metropolis et al. [6] that is extensively used among other things to estimate the expectation of certain operators on configurations under the weighted distribution.

In such applications efficiency again depends crucially on the rate of convergence of the Markov chain. Significantly, our proof technique for rapid mixing seems to generalise easily to other interesting chains. We substantiate this claim here by considering a Metropolis-style process for *monomer-dimer systems* [14], which are a model of physical systems involving diatomic molecules. In this case configurations correspond to matchings (independent sets of edges of any size) in a given weighted graph, and the weight of a configuration is the product of its edge weights. Using our earlier method of analysis, we are able to show that this Markov chain is rapidly mixing under very general conditions. As a result we deduce the existence of an fpras for the monomer-dimer partition function. This includes as a special case an fpras for the #P-complete problem of counting all matchings in an arbitrary graph.

The monomer-dimer chain also provides valuable new insight into our original problem of approximating the permanent. Most notably, by appending suitably chosen weights to the edges of the input graph we can use the chain to obtain a more elegant approximation scheme for counting perfect matchings. The scheme is immediately seen to be fully-polynomial if and only if the number of "near-perfect" matchings in the graph does not exceed the number of perfect matchings by more than a polynomial factor. This turns out to be rather a weak condition: it is satisfied not only by all dense graphs but also, in a sense that we will make precise later, by almost every bipartite graph that contains a perfect matching. Moreover, we present an efficient randomised algorithm for testing the condition for an arbitrary graph, allowing pathological examples to be recognised reliably.

A further byproduct of our work on the monomer-dimer process is the following. Consider the problem of finding a maximum cardinality matching in a given graph. The Markov chain above may be viewed as an application of the search heuristic known as *simulated annealing* [21] to this optimisation problem. Suppose the maximum cardinality of a matching is m and let $\varepsilon > 0$ be fixed. Then our results readily imply that, with a suitable choice of edge weights (or equivalently, "temperature"), the search finds a matching of size at least $(1-\varepsilon)m$ in polynomial time with high probability. This represents a considerable simplification of a recent result of Sasaki and Hajek [27].

The remainder of the paper is structured as follows. In § 2 we state our characterisation of rapid mixing in terms of conductance for a broad class of Markov chains, and illustrate by means of a simple example our technique for obtaining lower bounds on the conductance. Section 3 is devoted to a proof that Broder's method does indeed yield an fpras for the dense permanent. In § 4 we discuss the Markov chain for

monomer-dimer systems and derive an fpras for the partition function. Other applications of this chain, including the improved algorithm for the permanent, appear in § 5. Section 6 deals with the approximation of the permanent of a randomly selected 0-1 matrix of given density. Finally, in § 7 we list a few open problems.

2. Markov chains and rapid mixing. In this section we establish some general machinery for reasoning about the nonasymptotic behavior of Markov chains which will play a central role throughout the paper. We assume a nodding acquaintance with the elementary theory of finite Markov chains in discrete time. (For more information the reader is referred to [20].)

Let $(X_t)_{t=0}^{\infty}$ be a time-homogeneous Markov chain with finite state space $\mathcal{N}$ and transition matrix $P=(p_{ij})_{i,j\in\mathcal{N}}$. (All chains in this paper are assumed to be of this form.) If the chain is ergodic we let $\boldsymbol{\pi}=(\pi_i)_{i\in\mathcal{N}}$ denote its stationary distribution, the unique vector satisfying $\boldsymbol{\pi}P=\boldsymbol{\pi}$ and $\sum_i \pi_i = 1$. In this case, if the chain is allowed to evolve for t steps from any initial state the distribution of its final state approaches $\boldsymbol{\pi}$ as $t\to\infty$. Necessary and sufficient conditions for ergodicity are that the chain is (a) *irreducible*, i.e., any state can be reached from any other in some number of steps; and (b) *aperiodic*, i.e., $\gcd\{s : i \text{ is reachable from } j \text{ in } s \text{ steps}\} = 1$ for all $i, j \in \mathcal{N}$.

As explained in the previous section, our intention is to use simulation of an ergodic chain as a means of sampling elements of the state space $\mathcal{N}$ from a distribution close to $\boldsymbol{\pi}$. We shall always assume that individual transitions can be simulated at low cost. From the point of view of efficiency, our major concern is therefore the rate at which the chain approaches stationarity. As a time-dependent measure of deviation from the limit, we define the *relative pointwise distance* (r.p.d.) after t steps by

$$\Delta(t) = \max_{i,j\in\mathcal{N}} \frac{|p_{ij}^{(t)} - \pi_j|}{\pi_j},$$

where $p_{ij}^{(t)}$ is the t-step transition probability from state i to state j. Thus $\Delta(t)$ gives the largest relative difference between the distribution of the state at time t and $\boldsymbol{\pi}$, maximised over initial states i. Our aim is to establish conditions under which the chain is *rapidly mixing*, in the sense that $\Delta(t)$ tends to zero fast as a function of t.

An ergodic Markov chain is said to be *time-reversible* if it satisfies the detailed balance condition

$$p_{ij}\pi_i = p_{ji}\pi_j \quad \forall i, j \in \mathcal{N}. \tag{1}$$

Analysis of time-reversible Markov chains is simplified by the following observation.

LEMMA 2.1. *Suppose (X_t) is an ergodic Markov chain with finite state space $\mathcal{N}$ and transition matrix P. Let $\boldsymbol{\pi}=(\pi_i)_{i\in\mathcal{N}}$ be any vector satisfying the detailed balance condition* (1) *and the normalisation condition $\sum_i \pi_i = 1$. Then the Markov chain (X_t) is time-reversible and $\boldsymbol{\pi}$ is its (unique) stationary distribution.* □

We may naturally identify a time-reversible chain with an underlying weighted graph as follows. The vertices of the graph are the states of the chain, and for each (not necessarily distinct) pair $i, j \in \mathcal{N}$ with $p_{ij} > 0$ there is an edge (i, j) of weight $w_{ij} = p_{ij}\pi_i = p_{ji}\pi_j$. For convenience we set $w_{ij} = 0$ for all pairs of states i, j between which no transition is possible. Note that this graph uniquely specifies the Markov chain.

As in [29], we define the *conductance* of a time-reversible chain with underlying graph H by

$$\Phi(H) = \min \frac{\sum_{i\in S, j\notin S} w_{ij}}{\sum_{i\in S} \pi_i} \tag{2}$$

where the minimisation is over all subsets S of states with $0<\sum_{i\in S}\pi_i\leqq\frac{1}{2}$. Note that the quotient in (2) is just the conditional probability that the stationary process escapes from S in a single step, given that it is initially in S. The conductance in some sense measures the rate at which the process can flow around the state space, so we might expect it to be connected with the rate of convergence of the chain. In fact, by relating $\Phi(H)$ to the second eigenvalue of the transition matrix that governs the transient behaviour of the chain, it is possible to obtain the following result. A proof can be found in [29].

THEOREM 2.2. *Let H be the underlying graph of a time-reversible ergodic Markov chain in which* $\min_i p_{ii}\geqq\frac{1}{2}$, *and let* $\pi_{\min}=\min_i \pi_i$ *be the minimum stationary state probability. Then the* r.p.d. *of the chain is bounded by*

$$\Delta(t)\leqq\frac{(1-\Phi(H)^2/2)^t}{\pi_{\min}}.$$ □

The following immediate corollary is useful. Define the function $\tau:\mathbb{R}^+\to\mathbb{N}$ by

$$\tau(\varepsilon)=\min\{t\in\mathbb{N}:\Delta(t')\leqq\varepsilon \text{ for all } t'\geqq t\}.$$

COROLLARY 2.3. *With the notation of Theorem* 2.2, *we have*

$$\tau(\varepsilon)\leqq\frac{2}{\Phi(H)^2}(\ln\pi_{\min}^{-1}+\ln\varepsilon^{-1}).$$ □

Theorem 2.2 allows us to investigate the rate of convergence of a time-reversible chain by examining the structure of its underlying graph: convergence will usually be rapid if the conductance is not too small. In our applications we will always be dealing with *families* of Markov chains $\mathcal{MC}(x)$ indexed by problem instances x. (Thus x might be a graph and $\mathcal{MC}(x)$ some Markov chain on the set of matchings in the graph.) Let $\tau^{(x)}$ denote the function τ above for the chain $\mathcal{MC}(x)$. Then the rapid mixing property referred to informally earlier requires that $\tau^{(x)}(\varepsilon)$ should be bounded above by a polynomial in the input size $|x|$ and $\lg\varepsilon^{-1}$. This means that the number of steps required to achieve some specified sampling accuracy increases only polynomially with the problem size. As is clear from Corollary 2.3, rapid mixing will usually follow from a lower bound of the form $1/\text{poly}(|x|)$ on the conductance Φ. In this and subsequent sections, we show how to derive such bounds for several interesting chains. Other examples are given in [29].

Remarks. (a) The condition $\min_i p_{ii}\geqq\frac{1}{2}$ is a technical device that simplifies the statement of the theorem by damping oscillatory or "near-periodic" behaviour [29]. Note that an arbitrary Markov chain can be modified to make the condition hold simply by replacing P by $\frac{1}{2}(I+P)$, where I is the $|\mathcal{N}|\times|\mathcal{N}|$ identity matrix. This operation leaves the stationary distribution unchanged and merely reduces the conductance by a factor of $\frac{1}{2}$. (In fact, we have just added a self-loop probability of $\frac{1}{2}$ to each state; for the purposes of practical implementation the waiting time at each state may be simulated more efficiently by a separate random process.)

(b) Theorem 2.2 has a converse stating that, under the same assumptions, $\Delta(t)\geqq(1-2\Phi(H))^t$ [28]. Hence we effectively have a *characterisation* of rapid mixing for a large class of time-reversible chains in terms of the graph-theoretic quantity Φ.

(c) Similar relationships between subdominant eigenvalues of graphs and their structural properties have appeared in the work of Alon [4] and Alon and Milman [5]. The significance of Alon's result as a sufficient condition for rapid mixing for certain Markov chains has been noted by several authors; in particular, Aldous [2] states a restricted version of Theorem 2.2 for random walks on regular graphs. Our

conductance Φ is a weighted edge analogue of the *magnification* studied in [4], [2] and gives a cleaner and more natural formulation of this connection. Very recently, Lawler and Sokal [22] have independently discovered a characterisation similar to ours but in a rather more general context. □

As it stands, the rapid mixing criterion of Theorem 2.2 is essentially only of theoretical interest. To turn it into a useful practical tool, we need to develop some technology for estimating the conductance of the underlying graphs of natural Markov chains. This we now do with the aid of a simple example.

For a positive integer n, let $B(n)=\{0,1\}^n$ denote the set of bit vectors of length n. Consider the family of Markov chains $\mathscr{MC}(n)$ with state space $B(n)$ and transitions as follows. In any state $v=(v_0,\cdots,v_{n-1})$, select $i\in\{0,\cdots,n-1\}$ uniformly at random and flip the value of the bit v_i. To eliminate periodicity, remain at v with probability $\frac{1}{2}$.

$\mathscr{MC}(n)$ is obviously irreducible and aperiodic, and hence ergodic. Using Lemma 2.1, it is easily verified that $\mathscr{MC}(n)$ is time-reversible and that its stationary distribution is uniform over $B(n)$. The conditions of Theorem 2.2 are satisfied, so the rate of convergence of $\mathscr{MC}(n)$ depends on the conductance of its underlying graph $H(n)$. In $H(n)$, two states are adjacent if and only if they differ in at most one bit. Thus $H(n)$ is just the n-dimensional hypercube, each nonloop edge having weight $(2nN)^{-1}$, where $N=|B(n)|$ is the number of states.

PROPOSITION 2.4. *The conductance of* $H(n)$ *satisfies* $\Phi(H(n))\geqq 1/2n$.

Before presenting the proof of Proposition 2.4, which is the main point of this example, let us note that it implies rapid mixing for the family of chains $\mathscr{MC}(n)$. By Corollary 2.3 the number of simulation steps required to achieve an r.p.d. of ε is $O(n^2(n+\ln\varepsilon^{-1}))$, which is polynomially bounded in n and $\lg\varepsilon^{-1}$. Thus an algorithm that simulates $\mathscr{MC}(n)$ from some arbitrary initial state constitutes an efficient almost uniform sampling procedure for bit strings of length n. (Of course, there are more direct ways of doing this!)

Proof of Proposition 2.4. From the definition (2) we may write

$$\Phi(H(n))=\frac{1}{2n}\min_{0<|S|\leqq N/2}\frac{|\mathrm{cut}(S)|}{|S|} \tag{3}$$

where for each $S\subseteq B(n)$, $\mathrm{cut}(S)$ denotes the set of cut edges in $H(n)$ defined by S. Our argument hinges on the following observation. Suppose it is possible to specify a canonical simple path in $H(n)$ between each ordered pair of distinct states in such a way that no oriented edge of $H(n)$ is contained in more than bN of the paths. If S is any subset of states with $0<|S|\leqq N/2$, then the number of paths which cross the cut from S to its complement is clearly

$$|S|(N-|S|)\geqq|S|N/2.$$

Thus for any such S the number of cut edges must be at least $|S|N/2bN=|S|/2b$, and so from (3) we have

$$\Phi(H(n))\geqq\frac{1}{4nb}. \tag{4}$$

To get a lower bound on $\Phi(H(n))$ it therefore suffices to define a collection of canonical paths in $H(n)$ that are "sufficiently edge disjoint," as measured by the parameter b.

We now proceed to define a suitable set of paths. Let $u=(u_i)_{i=0}^{n-1}$ and $v=(v_i)_{i=0}^{n-1}$ be distinct elements of $B(n)$, and $i_1<\cdots<i_l$ be the positions in which u and v differ.

Then for $1 \leqq j \leqq l$, the jth edge of the canonical path from u to v corresponds to a transition in which the i_jth bit is flipped from u_{i_j} to v_{i_j}.

Consider now an arbitrary transition t of $\mathcal{MC}(n)$ (or, equivalently, an oriented edge of $H(n)$); our aim is to bound the number of paths that contain t. Suppose that t takes state $w = (w_i)$ to state $w' = (w'_i)$ by flipping the value of w_k, and let $P(t)$ denote the set of paths containing t, viewed as ordered pairs of states. Rather than counting elements of $P(t)$ directly, we will set up an *injective* mapping from $P(t)$ into the state space $B(n)$; this will yield an upper bound on the ratio b appearing in (4).

The mapping $\sigma_t: P(t) \to B(n)$ is defined as follows. Given an ordered pair $\langle u, v \rangle \in P(t)$, set $\sigma_t(u, v) = (s_i)$, where

$$s_i = \begin{cases} u_i, & 0 \leqq i \leqq k, \\ v_i, & k < i < n. \end{cases}$$

Thus $\sigma_t(u, v)$ agrees with u on the first $k+1$ bits and with v on the remainder. Note that we can express this definition more succinctly as $\sigma_t(u, v) = u \oplus v \oplus w'$, where $\oplus$ denotes bitwise exclusive-or.

We claim that $\sigma_t(u, v)$ is an unambiguous *encoding* of the endpoints u and v, so that σ_t is indeed injective. To see this, simply note that

$$u_i = \begin{cases} s_i, & 0 \leqq i \leqq k, \\ w_i, & k < i < n, \end{cases} \qquad v_i = \begin{cases} w'_i, & 0 \leqq i \leqq k, \\ s_i, & k < i < n. \end{cases}$$

Hence u and v may be recovered from knowledge of t and $\sigma_t(u, v)$, so σ_t is injective. It follows immediately that $|P(t)| \leqq N$; in fact, since all vectors (s_i) in the range of σ_t satisfy $s_k = w_k$, we have the slightly stronger result that $|P(t)| \leqq N/2$. Since t was chosen arbitrarily, the number of paths traversing *any* oriented edge cannot exceed $N/2$. Thus we may set $b = \frac{1}{2}$ in inequality (4) and deduce the desired bound on the conductance $\Phi(H(n))$. □

Remark. The bound of Proposition 2.4 is tight. To see this, let S be the subset of $B(n)$ consisting of all vectors with first bit 0 and note that $|\text{cut}(S)|/|S| = 1$. Hence $\Phi(H(n)) = 1/2n$. □

Some observations on the above proof are in order here. The idea of path counting is quite general and has been used before in the literature to investigate the connectivity properties of various graphs in other contexts (see e.g., [31], in which the hypercube is also studied). The novelty of our proof lies in the use of the injective mapping technique to bound the number of paths which traverse an edge. This is not actually necessary in this simple example as the paths could have been counted explicitly. The point is that in more complex cases the states of the chain will be less trivial structures, such as matchings in a graph, and we will have no useful information about their number—indeed, this is what we will ultimately be trying to compute. It is then crucial to be able to bound the maximum number of paths through any edge in terms of the number of states *without* explicit knowledge of these quantities. This is precisely what the injective mapping technique achieves. As we will see presently, it turns out to be rather generally applicable.

Other simple Markov chains may be analysed in a similar fashion [28]. Examples of rapidly mixing families include random walks on n-dimensional cubes of side d and a host of "card-shuffling" processes whose state space is the set of permutations of n objects and whose transitions correspond to some natural shuffling scheme. These and similar processes have been extensively studied using other methods such as

coupling, stopping times, and group representation theory (see [1], [3], [10] for a variety of examples). The time bounds obtained by these methods are generally rather tighter than ours and can often be shown to be optimal. However, the full power of our approach will become apparent in the sequel where it will permit the analysis of highly irregular chains with only a little additional effort. Most significantly, such chains have seemingly not proved amenable to analysis by any of the other established methods.

3. Approximating the permanent. In this section we consider Broder's method for approximating the permanent of a square 0-1 matrix, as sketched in § 1. Our main result is that the method yields an fpras for a large class of matrices, including all those that are sufficiently dense. Thus Broder's principal claim in [8] turns out to be true despite the fallacious coupling argument given there.

We will work with the perfect matching formulation of the permanent as described in § 1. Let $G=(U, V, E)$ be a bipartite graph with $U=V=\{0, \cdots, n-1\}$, and for $k \in \mathbb{N}$ let $M_k(G)$ denote the set of matchings of size k in G. Thus $M_n(G)$ is the set of perfect matchings in G and its cardinality is equal to the permanent of the $n \times n$ 0-1 matrix associated with G. We assume throughout this section that $M_n(G)$ is nonempty: it is well known that this property can be tested in polynomial time [11].

The method is based on the observation that an fpras for $|M_n(G)|$ can be constructed easily given an efficient procedure for sampling perfect and "near-perfect" matchings in G almost uniformly at random. First we must say more precisely what we mean by such a procedure. Let $\mathcal{N}$ be the set $M_n(G) \cup M_{n-1}(G)$. An *almost uniform generator* for $\mathcal{N}$ is a probabilistic algorithm that, when presented with G and a positive real *bias* ε, outputs an element of $\mathcal{N}$ such that the probability of each element appearing approximates $|\mathcal{N}|^{-1}$ within ratio $1+\varepsilon$. The generator is *fully polynomial* (f.p.) if it runs in time bounded by a polynomial in n and $\lg \varepsilon^{-1}$. (Generators for combinatorial structures are discussed in a more general framework in [16], [28], [29].) Actually, since all the generators we construct in this paper are based on rapidly mixing Markov chains, it should be clear that they embody effective procedures for sampling structures from a given distribution under *any* reasonable definition.

We call the bipartite graph G *dense* if its minimum vertex degree is at least $n/2$. It is shown in [8] that the problem of counting perfect matchings in dense graphs is no easier than counting them in general graphs, and hence is #P-complete. The following result is also proved in [8], and formalises the reduction from approximate counting to almost uniform generation in the case of dense graphs.

THEOREM 3.1 (Broder). *Suppose that, for all dense bipartite graphs G, there exists an f.p. almost uniform generator for $M_n(G) \cup M_{n-1}(G)$. Then there exists an fpras for $|M_n(G)|$ for all such graphs G.* □

Broder investigates the generation problem using a family of ergodic Markov chains $\mathcal{MC}_{\mathrm{pm}}(G)$ with state space $\mathcal{N}=M_n(G) \cup M_{n-1}(G)$ and uniform stationary distribution, in which transitions are made by adding and/or deleting edges locally. We now give a slightly modified definition of $\mathcal{MC}_{\mathrm{pm}}(G)$. View E as a subset of $U \times V$ and matchings in G as subsets of E. If $A, B \subseteq E$ and $e \in E$ then $A \oplus B$ denotes the symmetric difference of A and B, while $A+e$ and $A-e$ denote the sets $A \cup \{e\}$, $A \backslash \{e\}$, respectively. Transitions in $\mathcal{MC}_{\mathrm{pm}}(G)$ are specified as follows. In any state $M \in \mathcal{N}$, choose an edge $e=(u, v) \in E$ uniformly at random and then

(i) If $M \in M_n(G)$ and $e \in M$, move to state $M'=M-e$ (*Type* 1 *transition*);

(ii) If $M \in M_{n-1}(G)$ and u, v are unmatched in M, move to $M'=M+e$ (*Type* 2 *transition*);

(iii) If $M \in M_{n-1}(G)$, u is matched to w in M and v is unmatched in M, move to $M' = (M+e)-(u, w)$; symmetrically, if v is matched to w and u is unmatched, move to $M' = (M+e)-(w, v)$ (*Type 0 transition*);

(iv) In all other cases, do nothing.

We again eliminate periodicity by introducing an additional self-loop probability of $\frac{1}{2}$ for each state, i.e., with probability $\frac{1}{2}$ the process does not select a random edge as above but simply remains at M.

Using Lemma 2.1, it is a simple matter to check that $\mathcal{MC}_{\text{pm}}(G)$ is ergodic and time-reversible with uniform stationary distribution. What is not at all obvious is that the family of chains is rapidly mixing. This surprising fact is a consequence of the following theorem.

THEOREM 3.2. *Let G be dense and H be the underlying graph of the Markov chain $\mathcal{MC}_{\text{pm}}(G)$. Then $\Phi(H) \geqq 1/12n^6$.*

We shall prove the theorem in a moment after examining its implications.

COROLLARY 3.3. *There exists an f.p. almost uniform generator for $M_n(G) \cup M_{n-1}(G)$ in all dense bipartite graphs G.*

Proof. On input $\langle G, \varepsilon \rangle$, the generator deterministically constructs an initial state of $\mathcal{MC}_{\text{pm}}(G)$ and then simulates the chain for some number $T \geqq \tau(\varepsilon/2)$ of steps, outputting the final state. (Assuming as we may that $\varepsilon \leqq 1$, an r.p.d. of $\varepsilon/2$ guarantees a bias of at most ε.) To see that the generator is f.p., note that for any G individual steps of $\mathcal{MC}_{\text{pm}}(G)$ can be simulated, and a perfect matching to serve as initial state found, in polynomial time. Moreover, since $\pi_{\min}^{-1} = |\mathcal{N}|$ is certainly bounded above by 2^{n^2}, Theorem 3.2 and Corollary 2.3 imply that $\tau(\varepsilon/2) \leqq \text{poly}(n, \lg \varepsilon^{-1})$, so T need only be this large. Hence the overall runtime is bounded as required. □

Combining this with Theorem 3.1 immediately yields the following Corollary.

COROLLARY 3.4. *There exists an fpras for $|M_n(G)|$ in all dense bipartite graphs G, and hence for the permanent of all dense square 0-1 matrices.* □

We return now to the proof of the key result above.

Proof of Theorem 3.2. As in (3) the conductance is given by

$$\Phi(H) = \frac{1}{2|E|} \min_{0<|S| \leqq |\mathcal{N}|/2} \frac{|\text{cut}(S)|}{|S|} \tag{5}$$

where again cut (S) denotes the set of cut edges in H defined by S. We will proceed as in the proof of Proposition 2.4 by defining a set of canonical paths in H. If no transition occurs in more than $b|\mathcal{N}|$ of these, by analogy with (4) we will have the bound

$$\Phi(H) \geqq \frac{1}{4b|E|} \geqq \frac{1}{4bn^2}. \tag{6}$$

We begin by specifying, for each $M \in \mathcal{N}$, canonical paths to and from a unique "closest" perfect matching $\bar{M} \in M_n(G)$ as follows, where $u \in U$ and $v \in V$ denote the unmatched vertices (if any) of M:

(i) If $M \in M_n(G)$ then $\bar{M} = M$ and the path is empty;

(ii) If $M \in M_{n-1}(G)$ and $(u, v) \in E$, then $\bar{M} = M + e$ and the path consists of a single Type 2 transition;

(iii) If $M \in M_{n-1}(G)$ and $(u, v) \notin E$, fix some $(u', v') \in M$ such that $(u, v'), (u', v) \in E$: note that at least one such edge must exist by the density assumption on G. Then $\bar{M} = (M-(u', v'))+(u, v')+(u', v)$, and we specify one of the two possible paths of length two from M to $\bar{M}$, involving a Type 0 transition followed by a Type 2 transition.

The canonical path from $\bar{M}$ to M consists of the same edges of H traversed in the opposite direction.

For future reference, we observe that no perfect matching is involved in too many canonical paths of the above form: for $M \in M_n(G)$ define the set

$$\mathcal{K}(M) = \{M' \in \mathcal{N} : \bar{M}' = M\}.$$

Then, since each matching in $\mathcal{K}(M)$ has at least $n-2$ edges in common with M, it is easy to see that $|\mathcal{K}(M)| \leq n^2$. Note that the sets $\mathcal{K}(M)$ partition $\mathcal{N}$, implying that $|\mathcal{N}| \leq n^2 |M_n(G)|$. It is also worth noting that this is the only point in the proof at which the bipartite structure of G is used: we will have more to say about this later (see remark (d) below).

Next we define a canonical path in H between an ordered pair I, F of *perfect* matchings (refer to Fig. 1(a)). To do this, we first assume a fixed ordering of all even cycles of G, and distinguish in each cycle a *start vertex* in U. Now consider the symmetric difference $I \oplus F$; we may write this as a sequence $C_1, \cdots, C_r$ of disjoint even cycles, each of length at least four, where the indices respect the above ordering. The path from I to F involves *unwinding* each of the cycles $C_1, \cdots, C_r$ in turn in the following way. Suppose the cycle C_i has start vertex u_0 and consists of the sequence of distinct vertices $(u_0, v_0, u_1, v_1, \cdots, u_l, v_l)$, where $(u_j, v_j) \in I$ for $0 \leq j \leq l$ and the remaining edges are in F. Then the first step in the unwinding of C_i is a Type 1 transition that removes the edge (u_0, v_0). This is followed by a sequence of l Type 0 transitions, the jth of which replaces the edge (u_j, v_j) by (u_j, v_{j-1}). The unwinding is completed by a Type 2 transition that adds the edge (u_0, v_l).

Finally, the canonical path between any pair of matchings $I, F \in \mathcal{N}$ is defined as the concatenation of three segments as follows:

initial segment: follow the canonical path from I to $\bar{I}$,
main segment: follow the canonical path from $\bar{I}$ to $\bar{F}$,
final segment: follow the canonical path from $\bar{F}$ to F.

Now consider an arbitrary oriented edge of H, corresponding to a transition t in the Markov chain. We aim to establish an upper bound of the form $b|\mathcal{N}|$ on the number of canonical paths that contain this transition. Suppose first that t occurs in the *initial* segment of a path from I to F, where $I, F \in \mathcal{N}$. Then it is clear from the definition of initial segment that the perfect matching $\bar{I}$ is uniquely determined by t. But we have already seen that $|\mathcal{K}(\bar{I})| \leq n^2$. Since $I \in \mathcal{K}(\bar{I})$, the number of paths that contain t in their initial segment is thus at most $n^2|\mathcal{N}|$. A symmetrical argument shows that the number of paths containing t in their final segment is similarly bounded.

To handle the main segments of the paths, we make use of the injective mapping technique seen in the proof of Proposition 2.4. This will obviate the need for any explicit counting of structures in $\mathcal{N}$, which is crucial here (cf. the discussion following the proof of Proposition 2.4). Let t be a transition from M to M', where $M, M' \in \mathcal{N}$ are distinct, and denote by $P(t)$ the set of ordered pairs $\langle I, F \rangle$ of *perfect* matchings such that t is contained in the canonical path from I to F. We proceed to define, for each pair $\langle I, F \rangle \in P(t)$, an encoding $\sigma_t(I, F) \in \mathcal{N}$ from which I and F can be uniquely reconstructed. The intention is that, if $C_1, \cdots, C_r$ is the ordered sequence of cycles in $I \oplus F$, and t is traversed during the unwinding of C_i, then the encoding should agree with I on $C_1, \cdots, C_{i-1}$ and on that portion of C_i that has already been unwound, and with F elsewhere.

With this in mind, consider the set $S = I \oplus F \oplus (M \cup M')$. Since $I \cap F \subseteq M \cup M' \subseteq I \cup F$ and $|I| = |F| = |M \cup M'| = n$, elementary set theory tells us that $|S| = n$. Furthermore, suppose that some vertex u is adjacent to two edges in S. Then both these edges necessarily lie in $I \oplus F$, which in turn implies that neither edge lies in $M \cup M'$. Hence

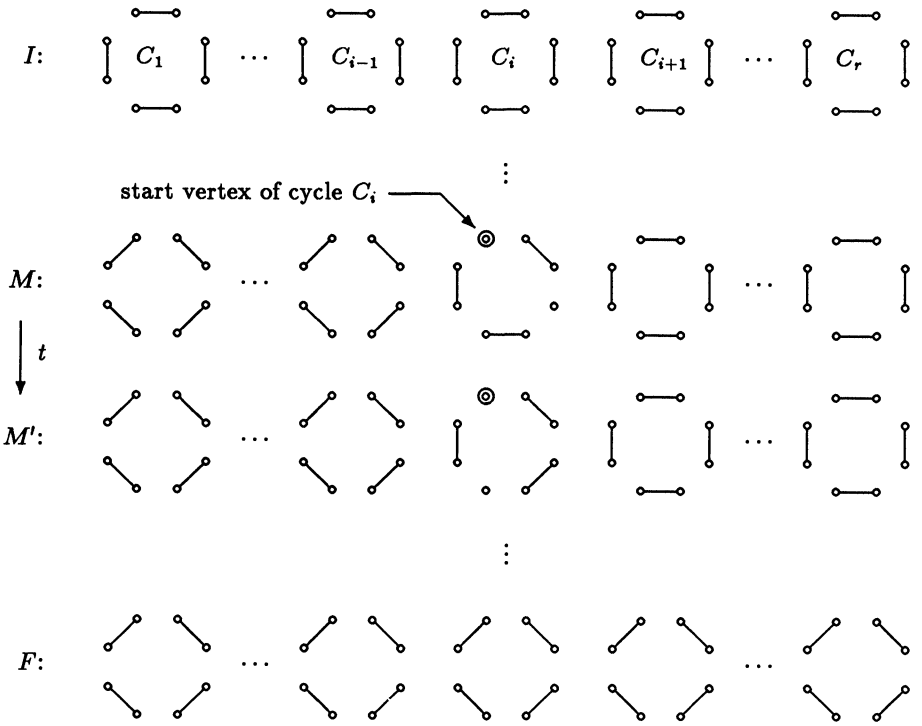


FIG. 1(a). *A transition t on the canonical path from I to F.*

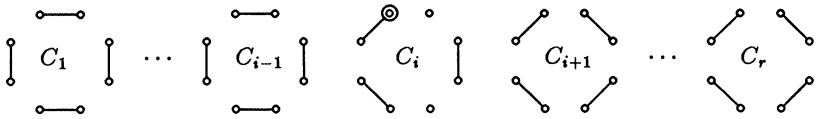


FIG. 1(b). *The corresponding encoding* $\sigma_t(I, F)$.

the vertex u must be unmatched in $M \cup M'$. From the form of the transitions, however, it is clear that $M \cup M'$ contains at most one such vertex $u = u_t$; moreover, this is the case if and only if t is a Type 0 transition, and u_t must then be the start vertex of the cycle currently being unwound. In this case, we denote by $e_{I,t}$ the edge of I incident with u_t.

We are now in a position to define the encoding:

$$\sigma_t(I, F) = \begin{cases} (I \oplus F \oplus (M \cup M')) - e_{I,t} & \text{if } t \text{ is Type 0,} \\ I \oplus F \oplus (M \cup M') & \text{otherwise.} \end{cases}$$

Figure 1(b) illustrates this definition for a Type 0 transition. In view of the above discussion, $\sigma_t(I, F)$ is always a matching of cardinality at least $n-1$, and hence an element of $\mathcal{N}$. It remains for us to show that I and F can be recovered from it.

First observe that $I \oplus F$ can be recovered immediately using the relation

$$I \oplus F = \begin{cases} (\sigma_t(I, F) \oplus (M \cup M')) + e_{I,t} & \text{if } t \text{ is Type 0,} \\ \sigma_t(I, F) \oplus (M \cup M') & \text{otherwise.} \end{cases}$$

(Note that $e_{I,t}$ is the unique edge that must be added to $\sigma_t(I, F) \oplus (M \cup M')$ to ensure that $I \oplus F$ is a union of disjoint cycles.) Thus we may infer the ordered sequence

$C_1, \cdots, C_r$ of cycles to be unwound on the path from I to F. The cycle C_i that is currently being unwound, together with its parity with respect to I and F, is then determined by the transition t. The parity of all remaining cycles may be deduced from M and the cycle ordering. Finally, the remaining portions of I and F may be recovered using the fact that $I \cap F = M \setminus (I \oplus F)$. Hence $\sigma_t(I, F)$ uniquely determines the pair $\langle I, F \rangle$, so σ_t is an injective mapping from $P(t)$ to $\mathcal{N}$.

The existence of σ_t ensures that $|P(t)| \leqq |\mathcal{N}|$ for any transition t. Since also $|\mathcal{K}(M)| \leqq n^2$ for any perfect matching M, we see that t is contained in the main segment of at most $n^4|\mathcal{N}|$ paths. Combining this with the results for initial and final segments derived earlier, we deduce that the maximum total number of paths that contain t is bounded by

$$(n^2 + n^2 + n^4)|\mathcal{N}| \leqq 3n^4|\mathcal{N}|.$$

To complete the proof we set $b = 3n^4$ in (6). □

Remarks. (a) In his original paper [8], Broder claimed that the above rapid mixing property holds under the same density assumption. However, as indicated in § 1, his proof based on coupling ideas is both complex and fundamentally flawed. The problem is that the "coupling" defined in [8] is not, in fact, a coupling because one of the two processes involved is not a faithful copy of $\mathcal{MC}_{\text{pm}}(G)$: this is explained in detail by Mihail [25]. As a result Broder has withdrawn his proof (see the erratum to [8]). We feel that this is compelling evidence of the unsuitability of coupling and related methods for the analysis of Markov chains that lack a high degree of symmetry.

(b) A chain with larger conductance is obtained by modifying $\mathcal{MC}_{\text{pm}}(G)$ slightly so that transitions are effected by selecting a random vertex in V rather than a random edge. This increases the transition probability in (5) from $1/2|E|$ to $1/2n$, saving a factor of n in the conductance bound.

(c) The f.p. almost uniform generator for $M_n(G) \cup M_{n-1}(G)$ may be adapted to one for *perfect* matchings in dense graphs G, by repeatedly generating elements of $M_n(G) \cup M_{n-1}(G)$ and outputting the first perfect matching which occurs. Since $|M_n(G)| \geqq n^{-2}|\mathcal{N}|$ we should not have to wait too long, but if so some arbitrary perfect matching may be output without affecting the bias too much. Among other things, with appropriate choice of G this provides a way of generating certain natural restricted classes of permutations that satisfy the density condition, such as displacements or ménage arrangements.

(d) So far we have concentrated exclusively on bipartite graphs because of their connection with the permanent. The Markov chain $\mathcal{MC}_{\text{pm}}(G)$ can be applied without essential modification to arbitrary graphs G. In fact, the only point at which we have relied on the bipartite structure of G is in the definition of the sets $\mathcal{K}(M)$ in the proof of Theorem 3.2 and the bound on their size. Let $G = (V, E)$ be an arbitrary graph with $|V| = 2n$. As before, we assume that G contains a perfect matching. Call G *dense* if its minimum vertex degree is at least n. This ensures that $\mathcal{K}(M)$ for $M \in M_n(G)$ is still well defined, and that $|\mathcal{K}(M)| \leqq 2n^2$. The rest of the proof carries through as before, yielding $b = 8n^4$ and consequently $\Phi(H) \geqq 1/64n^6$. (This can again be improved if transitions are implemented by random vertex selection.) Since a construction analogous to that of Theorem 3.1 holds for general dense graphs, we have:

COROLLARY 3.5. *There exists an fpras for $|M_n(G)|$ in arbitrary dense graphs.* □

We conclude this section by examining the role played in our results by the density assumption. In the proof of Theorem 3.2 it was used to show that each element of $M_{n-1}(G)$ is "close to" a perfect matching, so that near-perfect matchings could effectively be ignored in the conductance argument. In fact, it is enough to know that

the total *number* of near-perfect matchings is not too large. More specifically, the above results hold under the considerably weaker assumption that

$$|M_{n-1}(G)|/|M_n(G)| \leqq q(n) \tag{7}$$

for some fixed polynomial q. (Recall that we are assuming $|M_n(G)| > 0$.) This condition will arise more naturally in the context of the improved algorithm of § 5. Accordingly, we only sketch the proof modifications necessary to extend the method of this section. A fuller account can be found in [28].

First, it is not hard to see that the reduction of Theorem 3.1 still holds for graphs G satisfying the weaker condition (7). (This fact relies on Theorem 5.1 of § 5.) It is therefore enough to generate elements of $\mathcal{N} = M_n(G) \cup M_{n-1}(G)$ using the Markov chain $\mathcal{MC}_{\text{pm}}(G)$: the only thing we have to check is that its conductance remains bounded below by an inverse polynomial function of n. This is a consequence of the following generalised version of Theorem 3.2.

THEOREM 3.6. *For any graph $G = (V, E)$ with $|V| = 2n$ and $|M_n(G)| > 0$, the conductance of the underlying graph of $\mathcal{MC}_{\text{pm}}(G)$ is bounded below by*

$$\frac{1}{16|E|}\left(\frac{|M_n(G)|}{|M_{n-1}(G)|}\right)^2.$$

Sketch of proof. Let H be the underlying graph of $\mathcal{MC}_{\text{pm}}(G)$. The proof differs from that of Theorem 3.2 in one or two details. First, we use a variant of the canonical path counting argument in which only paths from perfect to near-perfect matchings are considered. If these can be defined in such a way that no edge of H carries more than $b|\mathcal{N}|$ of them, then a little algebra yields (cf. (6))

$$\Phi(H) \geqq \frac{1}{16b|E|}\left(\frac{|M_n(G)|}{|M_{n-1}(G)|}\right). \tag{8}$$

The paths themselves are similar to those between perfect matchings in the proof of Theorem 3.2: if $I \in M_n(G)$ and $F \in M_{n-1}(G)$, the symmetric difference $I \oplus F$ consists of a sequence $C_1, \cdots, C_r$ of disjoint cycles as before, together with a single open path O that is unwound in the obvious way *after* all the C_i.

Let t be an arbitrary transition. The injective mapping technique can again be used to bound the cardinality of the set $P(t)$ of paths of the above kind that involve t. For $\langle I, F\rangle \in P(t)$ the encoding $\sigma_t(I, F)$ is defined exactly as before, and σ_t is again injective. Now, however, we find that $\sigma_t(I, F) \in M_{n-1}(G) \cup M_{n-2}(G)$, as we get an additional pair of unmatched vertices arising from the open path O. (Note that this time the encoding takes us outside the state space.) Since σ_t is injective we have

$$|P(t)| \leqq |M_{n-1}(G)| + |M_{n-2}(G)| \leqq \left(\frac{|M_{n-1}(G)|}{|M_n(G)|}\right)|\mathcal{N}|,$$

where the second inequality again appeals to Theorem 5.1. Thus we may take $b = |M_{n-1}(G)|/|M_n(G)|$ in (8), completing the proof of the theorem. □

COROLLARY 3.7. *Let q be any fixed polynomial. There exists an fpras for $|M_n(G)|$ in all $2n$-vertex graphs G that satisfy $|M_{n-1}(G)|/|M_n(G)| \leqq q(n)$.* □

Our earlier results for dense graphs can be derived as a special case of Corollary 3.7 with $q(n) = O(n^2)$. Note that the density bound quoted is tight in the sense that it is possible to construct, for any fixed $\delta > 0$, a sequence of (bipartite) graphs (G_n) with $2n$ vertices and minimum vertex degree at least $n/(2+\delta)$ such that the ratio $|M_{n-1}(G_n)|/|M_n(G_n)|$ is exponentially large.

Remark. Dagum et al. [9] have shown that the reduction from counting to generation of Theorem 3.1 may be replaced by the following mechanism, with a small increase in efficiency. In analogous fashion to $\mathcal{MC}_{\text{pm}}(G)$, we may define for $1 \leqq k \leqq n$ a Markov chain $\mathcal{MC}_k(G)$ whose states are k- and $(k-1)$-matchings in G. (Thus $\mathcal{MC}_n(G)$ is just $\mathcal{MC}_{\text{pm}}(G)$.) By a simple extension of the proof of Theorem 3.6, whereby multiple rather than unique canonical paths between states are counted, it can be shown that each of the chains $\mathcal{MC}_k(G)$ is rapidly mixing under the same condition (7) on G. This allows the ratios $|M_k(G)|/|M_{k-1}(G)|$ to be estimated directly for each k in turn. We do not dwell on this point here as we will present a more natural algorithm in § 5. □

This concludes our discussion of perfect matchings for now. An alternative view of all these results will emerge as a by-product of our work on a different problem in the next section.

4. Monomer–dimer systems. This section is concerned with counting and generating at random all matchings (independent sets of edges) in a graph. Apart from their inherent interest, these problems arise in the theory of statistical physics, a rich source of combinatorial counting and generation problems.

A *monomer–dimer system* consists of a graph $G=(V, E)$, which is usually some form of regular lattice, together with a positive weight on each edge. The vertices of G represent physical sites, adjacent pairs of which may be occupied by diatomic molecules or *dimers*. Configurations of the system correspond to arrangements of dimers on the lattice in which no two dimers overlap, i.e., to matchings in G. In a configuration consisting of fewer than $|V|/2$ dimers, unoccupied sites are referred to as *monomers*. Monomer–dimer systems have been extensively studied as models of physical systems involving diatomic molecules. In the two-dimensional case they model the adsorption of dimers on the surface of a crystal. Three-dimensional systems occur in the theory of mixtures of molecules of different sizes and in the cell-cluster theory of the liquid state. For further information, see [14] and the references given there.

For each edge e of G, the weight $c(e)$ represents the relative probability of occupation by a dimer. This will depend on the contribution of such a dimer to the global energy of the system. Most thermodynamic properties of the system can be deduced from knowledge of the *partition function*

$$Z(G) = \sum_{M \in M_*(G)} W(G, M) \tag{9}$$

where $M_*(G)$ is the set of configurations (matchings in G) and $W(G, M) = \prod_{e \in M} c(e)$ is the *weight* of the configuration M. Counting matchings, i.e., computing (9) in the special case where all edge weights are unity, is a #P-complete problem even when restricted to planar graphs [15], [33]. The main result of this section is that the more general sum (9) can in fact be *approximated* efficiently for any weighted graph G.

We will proceed as in the previous section via a related random generation problem for configurations. Since the sum in (9) is weighted, however, configurations should be generated not uniformly but with probabilities proportional to their weights. In fact, this problem is of interest in its own right as a means of estimating the expectation of various physical operators on configurations by the so-called Monte Carlo method [6].

The notion of an almost uniform generator can be generalised to the weighted case in the obvious way. We will call a probabilistic algorithm an *almost W-generator* for matchings if, given a graph G with positive edge weights and a positive real bias $\varepsilon > 0$, it outputs a matching M in G with probability that approximates

$W(G, M)/Z(G)$ within ratio $1+\varepsilon$. As usual, the generator is f.p. if its runtime is bounded by a polynomial in the size of the input G and in $\lg \varepsilon^{-1}$.

The problem of approximating the partition function (9) is reduced to the weighted generation problem as follows. Let $e=(u, v)$ be any edge of the weighted graph $G=(V, E)$, G^- the graph obtained by removing e from G, and G^+ the graph obtained by removing the vertices u and v together with all their incident edges. By partitioning matchings in G into two sets according to whether they do or do not contain e, it is readily seen that

(i) There is a (1-1)-correspondence between $M_*(G)$ and the disjoint union of $M_*(G^+)$ and $M_*(G^-)$;

(ii) $Z(G)=c(e)Z(G^+)+Z(G^-)$.

This suggests the following recursive procedure for estimating $Z(G)$:

(1) Using an almost W-generator, construct an independent sample of elements of $M_*(G)$ as detailed below.

(2) For some edge e of G, let z^+, z^- denote the proportions of elements in the sample that do and do not contain e, respectively. Note that these quantities estimate $c(e)Z(G^+)/Z(G)$ and $Z(G^-)/Z(G)$, respectively.

(3) If $z^+ \geqq z^-$, recursively estimate $Z(G^+)$ and multiply the result by $c(e)/z^+$; otherwise, recursively estimate $Z(G^-)$ and multiply by $1/z^-$.

The procedure terminates when the input graph contains no edges. Note that the choice of the larger ratio in step (3) maximises the accuracy of the method.

Some elementary but tedious statistics (see [16, Thm. 6.4]) confirms that, if the bias in the generator is set to $\varepsilon/\alpha|E|$ for some constant α, the sample size required in step (1) to ensure that the final answer approximates $Z(G)$ within ratio $1+\varepsilon$ with probability at least $\frac{3}{4}$ is only $O(|E|^3\varepsilon^{-2})$. We therefore have the following theorem.

THEOREM 4.1. *Suppose there exists an f.p. almost W-generator for matchings. Then there exists an fpras for the partition function of monomer-dimer systems.* □

Remark. The foregoing is an example of a more general reduction from approximate counting to random generation, justified in detail in [16], [28], that applies to all structures that are *self-reducible*. Informally, this means that the set of structures corresponding to any problem instance is in (1-1)-correspondence with the disjoint union of sets corresponding to a few smaller problem instances (subproblems). In the case of matchings this property is expressed in condition (i) above. Since we are dealing here with *weighted* combinatorial sums, we need to supplement the definition of self-reducibility by demanding that any sum can be computed easily given the sums for its subproblems: condition (ii) states this for the monomer-dimer partition function. This implies that the Markov chain approach to random generation studied in this paper is potentially a powerful *general* method for approximating hard combinatorial counting problems. (Note that in the previous section it was necessary to resort to the specialised reduction provided by Theorem 3.1. The reason is that, while perfect matchings are easily seen to be self-reducible in general, this property is apparently destroyed when restrictions are placed on the input graph as in § 3.) □

Theorem 4.1 says that we will get an fpras for the monomer-dimer partition function provided we can efficiently generate matchings with probabilities roughly proportional to their weights. This we achieve by simulating a Markov chain in the style of Metropolis et al. [6]. Given a graph $G=(V, E)$ with positive edge weights $\{c(e): e \in E\}$, we consider the chain $\mathcal{MC}_{\mathrm{md}}(G)$ with state space $\mathcal{N}=M_*(G)$ and transitions as follows. In any state $M \in \mathcal{N}$, choose an edge $e=(u, v) \in E$ uniformly at random and then

(i) If $e \in M$, move to $M-e$ with probability $1/(1+c(e))$ (*Type* 1 *transition*);

(ii) If u and v are both unmatched in M, move to $M+e$ with probability $c(e)/(1+c(e))$ (*Type 2 transition*);

(iii) If $e'=(u, w)\in M$ for some w, and v is unmatched in M, move to $(M+e)-e'$ with probability $c(e)/(c(e)+c(e'))$ (*Type 0 transition*);

(iv) In all other cases, do nothing.

As always, we simplify matters by adding a self-loop probability of $\frac{1}{2}$ to each state. It is then readily checked that $\mathscr{MC}_{\mathrm{md}}(G)$ is irreducible and aperiodic, and hence ergodic. The stationary probability π_M of $M\in M_*(G)$ is easily seen, by Lemma 2.1, to be proportional to its weight $W(G, M)=\prod_{e\in M} c(e)$, so simulation of the chain will yield an f.p. almost W-generator for matchings provided the family $\mathscr{MC}_{\mathrm{md}}(G)$ is rapidly mixing. Now $\mathscr{MC}_{\mathrm{md}}(G)$ is clearly time-reversible by virtue of the detailed balance condition (1), so we may again apply Theorem 2.2. The crucial fact is the following.

THEOREM 4.2. *For a graph $G=(V, E)$ with positive edge weights $\{c(e): e\in E\}$, the conductance of the underlying graph of the Markov chain $\mathscr{MC}_{\mathrm{md}}(G)$ is bounded below by $1/(8|E|c_{\max}^2)$, where $c_{\max}=\max\{1, \max_{e\in E} c(e)\}$.*

Proof. Let H be the underlying graph of $\mathscr{MC}_{\mathrm{md}}(G)$. The first step is to establish a weighted version of the path counting argument that led to the bound (4). Suppose that between each ordered pair $\langle I, F\rangle$ of distinct states we have a canonical path in H, and let us associate with the path a weight $\pi_I\pi_F$. Also, for any subset S of states define

$$C_S=\sum_{M\in S}\pi_M \quad \text{the } \textit{capacity} \text{ of } S,$$

$$F_S=\sum_{M\in S, M'\notin S}\pi_M p_{MM'} \quad \text{the } \textit{ergodic flow} \text{ out of } S.$$

($p_{MM'}$ is the transition probability from M to M'.) Note that the conductance $\Phi(H)$ is just the minimum value of the ratio F_S/C_S over subsets S with $0<C_S\leqq\frac{1}{2}$. For any such S, the aggregated *weight* of all paths crossing the cut from S to its complement $\bar{S}$ in $\mathscr{N}$ is

$$\sum_{I\in S, F\in\bar{S}}\pi_I\pi_F=C_S C_{\bar{S}}\geqq\frac{C_S}{2}. \tag{10}$$

Now let t be a transition from a state M to a state $M'\neq M$, and denote by $P(t)$ the set of all ordered pairs $\langle I, F\rangle$ whose canonical path contains t. Suppose it is known that, for any such transition t, the aggregated weight of paths containing t satisfies

$$\sum_{\langle I,F\rangle\in P(t)}\pi_I\pi_F\leqq b w_t \tag{11}$$

where $w_t=\pi_M p_{MM'}=\pi_{M'}p_{M'M}$ is the weight of the edge in H corresponding to t. Taking (10) and (11) together, we have the following bound on the ergodic flow out of S, where $\mathrm{cut}(S)$ denotes the set of transitions crossing the cut from S to $\bar{S}$:

$$\begin{aligned} F_S=\sum_{t\in\mathrm{cut}(S)} w_t &\geqq b^{-1}\sum_{t\in\mathrm{cut}(S)}\sum_{\langle I,F\rangle\in P(t)}\pi_I\pi_F \\ &\geqq b^{-1}\sum_{I\in S, F\in\bar{S}}\pi_I\pi_F \\ &\geqq\frac{C_S}{2b}. \end{aligned}$$

By definition, the conductance of H therefore satisfies

$$\Phi(H)\geqq\frac{1}{2b}. \tag{12}$$

Our aim is thus to define a set of paths obeying a suitable bound b in (11).

To do this we generalise the proof of Theorem 3.2. Suppose there is an underlying order on all simple paths in G and designate in each of them a start vertex, which must

be an endpoint if the path is not a cycle but is arbitrary otherwise. For distinct $I, F \in \mathcal{N}$, we can write the symmetric difference $I \oplus F$ as a sequence $Q_1, \cdots, Q_r$ of disjoint paths that respects the ordering. The canonical path from I to F involves unwinding each of the Q_i in turn as follows. There are two cases to consider:

Case i. *Q_i is not a cycle.* Let Q_i consist of the sequence $(v_0, v_1, \cdots, v_l)$ of vertices, with v_0 the start vertex. If $(v_0, v_1) \in F$, perform a sequence of Type 0 transitions replacing (v_{2j+1}, v_{2j+2}) by (v_{2j}, v_{2j+1}) for $j = 0, 1, \cdots$, and finish with a single Type 2 transition if l is odd. If on the other hand $(v_0, v_1) \in I$, begin with a Type 1 transition removing (v_0, v_1) and proceed as before for the reduced path $(v_1, \cdots, v_l)$.

Case ii. *Q_i is a cycle.* Let Q_i consist of the sequence $(v_0, v_1, \cdots, v_{2l+1})$ of vertices, where v_0 is the start vertex, $l \geqq 1$ and $(v_{2j}, v_{2j+1}) \in I$ for $0 \leqq j \leqq l$, the remaining edges belonging to F. Then the unwinding begins with a Type 1 transition to remove (v_0, v_1). We are left with an open path O with endpoints v_0, v_1, one of which must be the start vertex of O. Suppose v_k, $k \in \{0, 1\}$, is *not* the start vertex. Then we unwind O as in Case (i) above but treating v_k as the start vertex. This trick serves to distinguish cycles from open paths, as will prove convenient shortly.

Now let t be a transition from M to $M' \neq M$. The next step is to define our injective mapping $\sigma_t : P(t) \to \mathcal{N}$. As in the proof of Theorem 3.2, we set $\sigma_t(I, F)$ equal to $I \oplus F \oplus (M \cup M')$, and remove the edge $e_{I,t}$ of I adjacent to the start vertex of the path currently being unwound if necessary: this is so if and only if the path is a cycle and t is Type 0. It is now easily seen that $\sigma_t(I, F)$ consists of independent edges, and so is an element of $\mathcal{N}$. The difference $I \oplus F$ can be recovered from $\sigma_t(I, F)$ using the relation

$$I \oplus F = \begin{cases} (\sigma_t(I, F) \oplus (M \cup M')) + e_{I,t} & \text{if } t \text{ is Type 0 and the current path is a cycle;} \\ \sigma_t(I, F) \oplus (M \cup M') & \text{otherwise.} \end{cases}$$

Note that we can tell whether the current path is a cycle from the sense of unwinding. Recovery of I and F themselves now follows as before from the path ordering. Hence σ_t is injective.

Moreover, it should be clear that $\sigma_t(I, F)$ is very nearly the complement of M in the union of I and F viewed as a multiset, so that the product $\pi_I \pi_F$ is approximately equal to $\pi_M \pi_{\sigma_t(I,F)}$, giving us a handle on b in (11). We now make this precise.

CLAIM. *For any $\langle I, F \rangle \in P(t)$, we have*

$$\pi_I \pi_F \leqq 4|E| c_{\max}^2 w_t \pi_{\sigma_t(I,F)}. \tag{13}$$

The claim will be proved in a moment. First note that it immediately yields the desired bound b in (11), since for any transition t we have

$$\sum_{\langle I,F \rangle \in P(t)} \pi_I \pi_F \leqq 4|E| c_{\max}^2 w_t \sum_{\langle I,F \rangle \in P(t)} \pi_{\sigma_t(I,F)} \leqq 4|E| c_{\max}^2 w_t$$

where the second inequality follows from the fact that σ_t is injective. We may therefore take $b = 4|E| c_{\max}^2$, which in light of (12) gives the conductance bound stated in the theorem.

It remains only for us to prove the claim. We distinguish three cases:

Case i. *t is a Type 1 transition.* Suppose $M' = M - e$. Then $\sigma_t(I, F) = I \oplus F \oplus M$, so, viewed as multisets, $M \cup \sigma_t(I, F)$ and $I \cup F$ are equal. Hence we have

$$\begin{aligned} \pi_I \pi_F &= \pi_M \pi_{\sigma_t(I,F)} \\ &= (w_t / p_{MM'}) \pi_{\sigma_t(I,F)} \\ &= 2|E|(1 + c(e)) w_t \pi_{\sigma_t(I,F)}, \end{aligned}$$

from which (13) follows.

Case ii. *t is a Type* 2 *transition.* This is handled by a symmetrical argument to Case (i) above, with M replaced by M'.

Case iii. *t is a Type* 0 *transition.* Suppose $M' = (M + e) - e'$, and consider the multiset $\sigma_t(I, F) \cup M$. This is equal to the multiset $I \cup F$ except that the edge e, and possibly also the edge $e_{I,t}$ are absent from it. Assuming $e_{I,t}$ is absent, which happens precisely when the current path is a cycle, we have

$$\begin{aligned}\pi_I \pi_F &= c(e_{I,t})c(e)\pi_M \pi_{\sigma_t(I,F)} \\ &= c(e_{I,t})c(e)(w_t/p_{MM'})\pi_{\sigma_t(I,F)} \\ &= 2|E|c(e_{I,t})(c(e)+c(e'))w_t \pi_{\sigma_t(I,F)},\end{aligned}$$

again satisfying (13). If $e_{I,t}$ is not absent, the argument is identical with the factor $c(e_{I,t})$ omitted.

This concludes the proof of the claim and the theorem. □

COROLLARY 4.3. *There exists an f.p. almost W-generator for matchings in arbitrary weighted graphs provided the edge weights are positive and presented in unary.*

Proof. Define $c_{\min} = \min\{1, \min_{e \in E} c(e)\}$. Then the minimum stationary state probability in $\mathcal{MC}_{\mathrm{md}}(G)$ is at least $c_{\min}^n 2^{-|E|} c_{\max}^{-n}$, where $n = |V|$. The logarithm of this quantity is at least $-p(|G|)$, where $|G|$ is the size of the description of G and p is a polynomial. Hence by Theorems 2.2 and 4.2, the Markov chains are rapidly mixing. Simulation of $\mathcal{MC}_{\mathrm{md}}(G)$ is a simple matter, starting from the empty matching. □

In view of Theorem 4.1, we may now state the main result of this section.

COROLLARY 4.4. *There exists an fpras for the monomer-dimer partition function of arbitrary weighted graphs with edge weights presented in unary.* □

COROLLARY 4.5. *There exists an fpras for the number of matchings in arbitrary graphs.* □

5. Some applications: The permanent revisited. As we have already mentioned, our analysis of the monomer-dimer Markov chain $\mathcal{MC}_{\mathrm{md}}(G)$ sheds new light on the results of § 3. In this section we will demonstrate that it yields a more natural approximation algorithm for counting perfect matchings in $2n$-vertex graphs G for which the ratio $|M_{n-1}(G)|/|M_n(G)|$ is polynomially bounded, and in addition allows this condition to be probabilistically tested for an arbitrary graph in polynomial time. We will also discuss an application of the chain to finding a maximum matching in a graph by simulated annealing. The key to all these algorithms is the introduction of carefully chosen edge weights.

The results of this section (and indeed Corollary 3.7 of § 3) depend crucially on the following property of matchings. As usual, let $M_k(G)$ denote the set of k-matchings in a graph G.

THEOREM 5.1. *For any graph G, the sequence* $\{|M_k(G)|: k \in \mathbb{N}\}$ *is log-concave, i.e.,*

$$|M_{k+1}(G)|\,|M_{k-1}(G)| \leqq |M_k(G)|^2 \quad \forall k \in \mathbb{N}^+.$$

Proof. A proof that relies on machinery from complex analysis can be found in Theorem 7.1 of [14] (see also [23, Exercise 8.5.10]). We present an elementary combinatorial proof that uses ideas seen elsewhere in this paper. Since log-concavity results in combinatorics tend to be rather hard to come by, we believe the simpler proof to be of independent interest.

Let $k \in \mathbb{N}^+$. We may assume that $|M_{k+1}(G)| > 0$, since the inequality is trivially true otherwise. Define the sets $A = M_{k+1}(G) \times M_{k-1}(G)$ and $B = M_k(G) \times M_k(G)$. Our aim is to show that $|A| \leqq |B|$.

As in the proof of Theorem 4.2, the symmetric difference $M \oplus M'$ of any two matchings M, M' in G consists of a set of disjoint simple paths (possibly closed) in G. Let us call such a path an *M-path* if it contains one more edge of M than of M'; an M'-path is defined similarly. Clearly, all other paths in $M \oplus M'$ contain equal numbers of edges from M and M'. Now for any pair $\langle M, M' \rangle \in A$, the number of M-paths in $M \oplus M'$ must exceed the number of M'-paths by precisely two. We may therefore partition A into disjoint classes $\{A_r : 0 < r \leqq k\}$, where

$$A_r = \{\langle M, M' \rangle \in A : M \oplus M' \text{ contains } r+1 \ M\text{-paths and } r-1 \ M'\text{-paths}\}.$$

Similarly, the sets $\{B_r : 0 \leqq r \leqq k\}$ with

$$B_r = \{\langle M, M' \rangle \in B : M \oplus M' \text{ contains } r \ M\text{-paths and } r \ M'\text{-paths}\}$$

partition B. The lemma will follow from the fact that $|A_r| \leqq |B_r|$ for each $r > 0$.

Let us call a pair $\langle L, L' \rangle \in B_r$ *reachable* from $\langle M, M' \rangle \in A_r$ if and only if $L \oplus L' = M \oplus M'$ and L is obtained from M by taking some M-path of $M \oplus M'$ and flipping the parity of all its edges with respect to M and M'. (This is analogous to *unwinding* the path in the proof of Theorem 4.2.) Clearly, the number of elements of B_r reachable from a given $\langle M, M' \rangle \in A_r$ is just the number of M-paths in $M \oplus M'$, namely $r+1$. Conversely, any given element of B_r is reachable from precisely r elements of A_r. Hence if $|A_r| > 0$ we have

$$\frac{|B_r|}{|A_r|} = \frac{r+1}{r} > 1,$$

completing the proof of the lemma. □

Remark. In [14] the tight inequality

$$|M_k(G)|^2 \geqq \frac{(k+1)(m-k+1)}{k(m-k)} |M_{k+1}(G)| \, |M_{k-1}(G)|$$

is proved, where $m = \lceil n/2 \rceil$ and n is the number of vertices in G. The bound in our proof can also be improved, but we will not labour this point here as simple log-concavity is quite adequate for our purposes. □

Note that Theorem 5.1 immediately implies the following:

COROLLARY 5.2. *For a $2n$-vertex graph $G = (V, E)$ with $|M_n(G)| > 0$, the ratio $|M_{k-1}(G)|/|M_k(G)|$ increases monotonically with k in the range $0 < k \leqq n$; the maximum value of the ratio is $|M_{n-1}(G)|/|M_n(G)|$ and the minimum value is $|E|^{-1}$.* □

Armed with log-concavity, let us sketch how an algorithm that generates matchings from the weighted distribution of § 4 may be used to estimate the number of perfect matchings in a given *unweighted* $2n$-vertex graph $G = (V, E)$. Write m_k in place of $|M_k(G)|$, and assume that $m_n > 0$. The idea is to estimate the ratios m_{k+1}/m_k in turn in a sequence of n stages for $k = 0, \cdots, n-1$. Since $m_0 = 1$, an approximation to m_n is then obtained as the product of the estimated ratios.

In stage k we could in principle estimate m_{k+1}/m_k using an algorithm that generates matchings in G *uniformly*: just observe the relative numbers of $(k+1)$- and k-matchings in an independent sample produced by the generator. However, a very large sample may be necessary since these matchings might constitute only a tiny fraction of all matchings in G. This difficulty can be overcome by adding to every edge of G a weight c_k that is chosen so as to make the aggregated weight $m_k c_k^k$ of k-matchings maximal,

i.e., at least as large as that of matchings of any other size. That such a weight exists is a direct consequence of the log-concavity of the m_i. To see this, take $c_k = m_{k-1}/m_k$ and let $G(c_k)$ denote the graph G augmented with weight c_k on every edge. (We will not have available the exact value of m_{k-1}/m_k, but it will suffice to substitute the *estimate* of this quantity obtained in the previous stage.) Then if p_i is the probability of being at an i-matching in the stationary distribution of the Markov chain $\mathcal{MC}_{\mathrm{md}}(G(c_k))$, we have for $i \geqq k$

$$\frac{p_k}{p_i} = \frac{m_k c_k^k}{m_i c_k^i} = c_k^{k-i} \prod_{j=k}^{i-1} \frac{m_j}{m_{j+1}} \geqq \left(\frac{m_{k-1}}{m_k}\right)^{k-i} \left(\frac{m_{k-1}}{m_k}\right)^{i-k} = 1 \tag{14}$$

where the inequality comes from Corollary 5.2. An identical bound holds for $i < k$. Since $\sum p_i = 1$, we conclude that $p_k \geqq (n+1)^{-1}$. Moreover, the probability of being at a $(k+1)$-matching satisfies

$$p_{k+1} = \left(\frac{m_{k+1} c_k}{m_k}\right) p_k = \left(\frac{m_{k+1}}{m_k}\right)\left(\frac{m_{k-1}}{m_k}\right) p_k \geqq \frac{1}{|E|}\left(\frac{m_n}{m_{n-1}}\right) p_k. \tag{15}$$

Hence a lower bound of the form $1/\mathrm{poly}(n)$ holds for p_{k+1} also, provided the ratio m_{n-1}/m_n is polynomially bounded. These observations allow the ratio m_{k+1}/m_k to be estimated efficiently by sampling from the *weighted* distribution.

For the sampling itself we appeal to the Markov chain technique of § 4. (The algorithm is robust enough to cope with a small bias.) In view of Corollary 4.3, the generator will be efficient provided the various edge weights used in the algorithm are polynomially bounded. But each weight will be close to m_{k-1}/m_k for some k that by Corollary 5.2 lies in the range $[|E|^{-1}, m_{n-1}/m_n]$. This ensures rapid convergence of the Markov chain at all stages, provided again that m_{n-1}/m_n is polynomially bounded.

Notice how a polynomial bound on the ratio m_{n-1}/m_n plays a central role in the efficient operation of this algorithm. For an arbitrary function q of the natural number n, let us call a $2n$-vertex graph G *q-amenable* if either

(i) $|M_n(G)| = 0$, or

(ii) $|M_n(G)| > 0$ and $|M_{n-1}(G)|/|M_n(G)| \leqq q(n)$.

From the above discussion, we might expect to get an fpras for counting perfect matchings in q-amenable graphs for any fixed polynomial q.

The new approximation scheme for counting perfect matchings in q-amenable graphs G is spelled out in detail in Fig. 2. In line (4), $\mathcal{G}$ denotes the almost W-generator for matchings described in § 4, i.e., the call $\mathcal{G}(G(c), \cdot)$ invokes a simulation of the Markov chain $\mathcal{MC}_{\mathrm{md}}(G(c))$. The values of the sample size T and bias ξ will depend on n and the accuracy $0 < \varepsilon \leqq 1$ specified for the final estimate, as described below. The test in line (1) for the existence of a perfect matching may be implemented using any standard polynomial time algorithm.

THEOREM 5.3. *For an arbitrary polynomial q, the algorithm of Fig. 2 is an fpras for $|M_n(G)|$ in all q-amenable $2n$-vertex graphs G.*

Proof. In view of line (1), we need only consider graphs for which $m_n > 0$. Line (2) and the iterations of the **for**-loop correspond to the n stages of the computation mentioned above. Let c_{k+1} be the value of the weight parameter c at the end of stage k. We claim that, by making T a polynomial function of n and ε^{-1} and setting $\xi = \varepsilon/\alpha n$ for a suitable constant $\alpha > 1$, the following may be guaranteed:

$$\forall k \quad \Pr\left(c_{k+1} \text{ approximates } m_k/m_{k+1} \text{ within ratio } 1+\frac{\varepsilon}{2n}\right) \geqq \left(1-\frac{1}{4n^2}\right)^k. \tag{16}$$

```
(1)  if |M_n(G)| = 0 then halt with output 0
     else begin
(2)    c := |E|^{-1}; Π := |E|;
       for k := 1 to n − 1 do begin
(3)      if c > 2q(n) or c < (2|E|)^{-1} then halt with output 0
         else begin
(4)        make T calls of the form 𝒢(G(c), ξ)
             and let Y be the set of outputs;
(5)        p̃_k := T^{-1}|Y ∩ M_k(G)|; p̃_{k+1} := T^{-1}|Y ∩ M_{k+1}(G)|;
(6)        if p̃_k = 0 or p̃_{k+1} = 0 then halt with output 0
(7)        else begin c := cp̃_k/p̃_{k+1}; Π := Π/c end
         end
       end;
(8)    halt with output Π
     end
```

FIG. 2. *Approximation scheme for counting perfect matchings.*

This will imply that the product Π output in line (8) approximates m_n within ratio $(1+\varepsilon/2n)^n \leqq 1+\varepsilon$ with probability $(1-1/4n^2)^{n^2} \geqq 3/4$, as required. Moreover, the runtime of the procedure is bounded by a polynomial in n and ε^{-1}. (Note in particular that, by Corollary 4.3, the bounds on edge weights in line (3) ensure that each call to $\mathscr{G}$ is bounded in this way.) Hence the procedure is indeed an fpras.

The proof of (16) is a straightforward induction on k, the technical details of which are left to the reader. The important points to note are the following, assuming that c_k is a good estimate of m_{k-1}/m_k:

(i) In line (3), c will not violate the prescribed bounds because m_{k-1}/m_k lies in the range $[|E|^{-1}, q(n)]$.

(ii) From (14) and (15), the probabilities p_k, p_{k+1} of being at a k- and $(k+1)$-matching in the stationary distribution of the Markov chain $\mathscr{MC}_{\mathrm{md}}(G(c_k))$ used in stage $k+1$ are bounded below by a function of the form $1/\mathrm{poly}(n)$. Hence the modest sample size T in line (4) is enough to make the estimates $\tilde{p}_k$, $\tilde{p}_{k+1}$ of these quantities in line (5) good with high probability. (Note that the pathological cases of line (6) are therefore very unlikely to occur.)

The assignment to c in line (7) therefore makes c_{k+1} a good estimate of m_k/m_{k+1} with high probability. □

The algorithm of Theorem 5.3 is preferable to those described in § 3 in several respects. For a given input graph G, it makes use of a single Markov chain structure, the only manipulations being simple scaling of transition probabilities. It avoids any discussion of ad hoc processes with state space $M_k(G) \cup M_{k-1}(G)$, whose transition structure is not uniform over states. Moreover, the condition that the ratio $|M_{n-1}(G)|/|M_n(G)|$ should be polynomially bounded (if $|M_n(G)| > 0$) is seen to arise directly from the log-concavity of the matching sequence.

Indeed, this condition seems to be a true characterisation of those graphs that can be handled by the algorithm, or equivalently of those matrices whose permanent we can efficiently approximate by this method. Since the condition is rather unfamiliar, it deserves further investigation. One worthwhile activity is to come up with simpler deterministic criteria that guarantee the condition holds. We have already seen one such criterion in § 3, namely that the graph is dense. Another criterion, due to Dagum et al. [9], is that the graph is bipartite and contains an αn-regular subgraph for some real $\alpha > 0$. However, as we will see in the next section, it turns out that the condition is a rather weak one and is, in fact, satisfied by almost all (bipartite) graphs. In other

words, there exists a fixed polynomial q such that almost every graph is q-amenable. Thus of more practical interest is the problem of testing efficiently for any given graph whether the condition holds. Such a test would enable us not only to approximate the permanent in almost all cases, but also to reliably identify difficult instances.

We now present an efficient randomised algorithm that tests the condition in the following strong probabilistic sense. Let q be a polynomial. When given as input a $2n$-vertex graph G containing a perfect matching and a positive real $\delta > 0$, the algorithm

(i) Accepts with probability at least $1-\delta$ if $|M_{n-1}(G)|/|M_n(G)| \leqq q(n)$;

(ii) Rejects with probability at least $1-\delta$ if $|M_{n-1}(G)|/|M_n(G)| > 6q(n)$.

For intermediate values of the ratio, we do not care whether the algorithm accepts or rejects. (The value 6 here is used for illustrative purposes only and may be replaced by any fixed constant greater than 1.) Furthermore, the runtime of the algorithm will be bounded by a polynomial in n and $\lg \delta^{-1}$.

Before presenting the algorithm we make precise its implications for counting perfect matchings. Consider the following combined procedure, whose input is an *arbitrary* $2n$-vertex graph G:

(1) Using a standard polynomial time algorithm, test whether G contains a perfect matching. If not, output 0 and halt.

(2) Apply the above randomised test for the condition $|M_{n-1}(G)|/|M_n(G)| \leqq q(n)$, having set an error probability $\delta = 2^{-n}$. If the algorithm rejects, output "Graph is not q-amenable" and halt.

(3) Using the approximation scheme of Fig. 2 with $q(n)$ replaced by $6q(n)$ (and the test of line (1) omitted), estimate $|M_n(G)|$ and output the result.

This procedure will run in polynomial time for any desired polynomial q. There are two ways in which it may produce a misleading result. With probability at most δ it may falsely claim that the input graph G is not q-amenable. Or, again with probability at most δ, it may output an unreliable approximation to $|M_n(G)|$ obtained under the false assumption that $|M_{n-1}(G)|/|M_n(G)| \leqq 6q(n)$. Since δ decreases exponentially with n, the procedure will, with very high probability, produce a result that is *not* misleading. This will either be a statement that G is not q-amenable, or a reliable approximation of $|M_n(G)|$.

We now show how to construct the testing algorithm advertised above. It again makes use of the weighted Markov chain generator $\mathcal{G}$ for matchings and is extremely simple to describe:

(1) Make T calls of the form $\mathcal{G}(G(2q(n)), 1/16)$, and let $\tilde{p}$ be the proportion of perfect matchings among the outputs. (T will depend on the input δ as specified below.)

(2) If $\tilde{p} \geqq 3/8$ accept, otherwise reject.

To see that this algorithm works, consider the stationary distribution of the Markov chain $\mathcal{MC}_{\mathrm{md}}(G(2q(n)))$, and let p denote the probability of being at a perfect matching. Writing as usual m_k in place of $|M_k(G)|$, Corollary 5.2 implies that

$$\frac{m_k}{m_n} = \prod_{i=k}^{n-1} \frac{m_i}{m_{i+1}} \leqq \left(\frac{m_{n-1}}{m_n}\right)^{n-k}$$

for $0 \leqq k \leqq n$. Hence in the case that $m_{n-1}/m_n \leqq q(n)$ we have

$$p = \frac{m_n(2q(n))^n}{\sum_{k=0}^{n} m_k(2q(n))^k} \geqq \frac{2^n}{\sum_{k=0}^{n} 2^k} > \frac{1}{2}. \tag{17}$$

On the other hand, if $m_{n-1}/m_n > 6q(n)$ we have

$$p \leqq \frac{m_n(2q(n))^n}{m_n(2q(n))^n + m_{n-1}(2q(n))^{n-1}} < \frac{m_n}{m_n + 3m_n} = \frac{1}{4}.$$

An elementary statistical argument now shows that, by taking $T = \alpha \lg \delta^{-1}$ for a suitable constant α, we can arrange for

$$\Pr(\tilde{p} \geqq \tfrac{3}{8}) \begin{cases} \geqq 1-\delta & \text{if } m_{n-1}/m_n \leqq q(n), \\ \leqq \delta & \text{if } m_{n-1}/m_n > 6q(n). \end{cases}$$

The runtime of the algorithm is therefore bounded as required.

Remark. It is often desirable to be able to count matchings of any specified cardinality in a given graph. In the context of the monomer-dimer systems of the previous section, these correspond to configurations with a given number of dimers. Obviously, the procedure of Fig. 2 may be modified so as to yield an fpras for $|M_k(G)|$ in graphs G for which the ratio $|M_{k-1}(G)|/|M_k(G)|$ is polynomially bounded. Such graphs may again be identified efficiently using a minor variant of the above randomised test. Note also that, under the same condition on G, we can easily adapt the algorithm of Fig. 2 to produce an f.p. almost uniform generator for $M_k(G)$ for any desired k. □

We close this section with a slight digression. In recent years, a stochastic search heuristic for combinatorial optimisation known as *simulated annealing* [21] has received much attention. The basic idea is that a Markov chain explores a space of configurations (feasible solutions), each of which has an associated cost or "energy." In the stationary distribution of the chain, low cost solutions have large weight so the chain tends to favour them asymptotically. By progressively reducing a "temperature" parameter, the weights are scaled so as to accentuate the depths of the energy wells. (Thus the process is not in general time-homogeneous.) While such a process is known to converge asymptotically under fairly general conditions (see, e.g., [13], [24]), virtually nothing useful is known about its *rate* of convergence when applied to nontrivial problems.

Consider the problem of finding a maximum cardinality matching in a graph G, which is nontrivial in the sense that all known polynomial time algorithms for solving it are far from simple. For any $c \geqq 1$ we may take the Markov chain $\mathcal{MC}_{\mathrm{md}}(G(c))$ as the basis for a simulated annealing algorithm for this problem: maximum cardinality matchings will certainly have maximum weight, and "temperature" may be reduced by increasing the edge weight c.

In [27], Sasaki and Hajek study the performance of algorithms of this kind. In particular, they prove a positive result of the following form.

THEOREM 5.4. *Let $\varepsilon > 0$ be any constant, $G = (V, E)$ be an input graph, and k_0 the maximum cardinality of a matching in G. Then a simulated annealing algorithm of the above type, operated at a fixed temperature (which depends on G and ε), finds a matching in G of cardinality at least $(1-\varepsilon)k_0$ with high probability in polynomial time.*

(In the same paper they also prove a strong negative result that says that no simulated annealing algorithm in this or a fairly large related class can be relied on to find a maximum cardinality matching in polynomial time with high probability.)

Sasaki and Hajek's proof is lengthy and complex. In contrast, we offer the following argument which rests directly on our earlier results.

Proof. Define $c_\varepsilon = 2|E|^{(1-\varepsilon)/\varepsilon}$. We claim that, in the stationary distribution of the Markov chain $\mathcal{MC}_{\mathrm{md}}(G(c_\varepsilon))$, the probability of being at a matching of size $k = \lceil (1-\varepsilon)k_0 \rceil$ or more is greater than $\frac{1}{2}$. Note that the theorem then follows at once: by

Corollary 4.3 a polynomially bounded simulation of $\mathscr{MC}_{\mathrm{md}}(G(c_\varepsilon))$ suffices to ensure that we visit a matching of size k or more with probability at least (say) $\frac{1}{4}$. This can be boosted to $1-\delta$ by repeating the entire experiment $O(\lg \delta^{-1})$ times. (However, in common with [27], our time bound increases exponentially with ε^{-1}.)

To justify the claim, note from Corollary 5.2 that

$$m_{k-1} = m_{k_0} \prod_{j=k}^{k_0} \frac{m_{j-1}}{m_j} \geqq \left(\frac{m_{k-1}}{m_k}\right)^{k_0-k+1}. \tag{18}$$

(Here we are using the fact that $m_{k_0} \geqq 1$.) But since j-matchings in G are subsets of E of size j, there is also the crude upper bound $m_{k-1} \leqq |E|^{k-1}$. Hence from (18) we conclude that

$$\frac{m_{k-1}}{m_k} \leqq |E|^{(1-\varepsilon)/\varepsilon} = \frac{c_\varepsilon}{2}. \tag{19}$$

A further application of Corollary 5.2 now shows that $m_i/m_k \leqq (c_\varepsilon/2)^{k-i}$ for $0 \leqq i \leqq k$, so the aggregated weight of matchings of size less than k is

$$\sum_{i=0}^{k-1} m_i c_\varepsilon^i \leqq \left(\sum_{i=0}^{k-1} 2^{i-k}\right) m_k c_\varepsilon^k < m_k c_\varepsilon^k.$$

It is now immediate that the probability of being at a matching of size k or more is at least $\frac{1}{2}$, completing the proof. □

Remark. Inequality (19) provides a polynomial upper bound on the ratio m_{k-1}/m_k, so from our earlier observations we are able to count and generate matchings of any cardinality up to $(1-\varepsilon)k_0$ in *arbitrary* graphs which contain a k_0-matching. □

6. Random permanents. For any polynomial q, the algorithm presented in Fig. 2 of the previous section is an fpras for the number of perfect matchings in a q-amenable graph G. For a bipartite graph G with $2n$ vertices, and $q(n) = n^2$, we have observed a sufficient condition for q-amenability, namely that the minimum vertex degree of G should be at least $n/2$. We have also observed that this result is the best possible, in the sense that, for any $\delta > 0$, there exists a family of graphs of minimum vertex degree at least $n/(2+\delta)$ for which $|M_{n-1}(G)|/|M_n(G)| = \exp\{\Omega(n)\}$.

The aim of this section is to demonstrate that these counterexamples are pathological, and that a randomly selected bipartite graph with given edge density—even when that density is small—will almost surely be q-amenable for some suitably chosen (fixed) polynomial q.

Let n be a positive integer, and p a real number in the interval $(0, 1)$. We will work with the probability space of bipartite graphs $G_{n,p}$ constructed according to the following random graph model. The vertex set of $G_{n,p}$ is $U+V$ where $U = V = \{0, \cdots, n-1\}$, and each potential edge (i.e., element of $U \times V$) is included in the edge set of $G_{n,p}$ independently and with probability p. (In the sequel, $G_{n,p}$ will always denote a graph randomly selected according to this model.) We say that an event A in this model occurs *with overwhelming probability* if $1 - \Pr(A) = O(n^{-k})$ for all integer k. (The O-expression here is a function of n only, and is independent of p.)

The main result of the section (Theorem 6.4) is that for most values of p, and for a suitably chosen (fixed) polynomial q, the graph $G_{n,p}$ is q-amenable with overwhelming probability. Thus, in probabilistic terms, the approximation scheme of Fig. 2 has very

wide applicability. Recall also that the rare examples that the scheme cannot handle reliably may be identified using the randomised test of the previous section. We approach Theorem 6.4 via a sequence of three technical lemmas, the proofs of which are deferred.

LEMMA 6.1. *Let $\varepsilon > 0$ be a fixed constant, and let $p \leqq (1-\varepsilon)n^{-1} \ln n$. Then, with overwhelming probability, $G_{n,p}$ has no perfect matching.*

Call a graph (k, m)-*expanding* [7, p. 327] if every k-subset of U is adjacent to at least m vertices of V, and vice versa.

LEMMA 6.2. *Let $\varepsilon > 0$ be a fixed constant, $p \geqq (1+\varepsilon)n^{-1} \ln n$, and $\alpha = pn/\ln \ln n$. Then, with overwhelming probability, $G_{n,p}$ is $(k, m+1)$-expanding for all integers k and m that satisfy the inequalities $k \geqq \ln n/\ln pn$, $m \leqq \alpha k$, and $m \leqq n/2$.*

LEMMA 6.3. *Let $p \geqq n^{-1} \ln n$. Then, with overwhelming probability, the maximum vertex degree of $G_{n,p}$ does not exceed $pn \ln n$.*

THEOREM 6.4. *Let $\varepsilon > 0$ be a fixed constant, and let p lie outside the interval*

$$((1-\varepsilon)n^{-1} \ln n, (1+\varepsilon)n^{-1} \ln n).$$

Then, with overwhelming probability, the graph $G_{n,p}$ is q-amenable for $q(n) = n^{10}$.

Proof. Let A_1 denote the event $|M_n(G_{n,p})| = 0$, and A_2 the event

$$|M_n(G_{n,p})| > 0 \quad \text{and} \quad |M_{n-1}(G_{n,p})|/|M_n(G_{n,p})| \leqq n^{10}.$$

The event that the graph $G_{n,p}$ is q-amenable is the disjunction of the events A_1 and A_2. If $p \leqq (1-\varepsilon)n^{-1} \ln n$ then, by Lemma 6.1, event A_1 occurs with overwhelming probability. So from now on we assume that $p \geqq (1+\varepsilon)n^{-1} \ln n$.

Let B be the event that $G_{n,p}$ is $(k, m+1)$-expanding for all k, m in the ranges allowed in the statement of Lemma 6.2; let C be the event that the maximum degree of $G_{n,p}$ does not exceed $pn \ln n$. Suppose, as we will prove, that the event A_2 is a logical consequence of the events B, C, and $\bar{A}_1$ (the complement of A_1), that is to say, $A_2 \supseteq B \cap C \cap \bar{A}_1$. Then, by elementary set theory, $A_1 \cup A_2 \supseteq A_1 \cup (B \cap C \cap \bar{A}_1) \supseteq (B \cap C)$. Thus, $\Pr(A_1 \cup A_2) \geqq \Pr(B \cap C) \geqq 1 - \Pr(\bar{B}) - \Pr(\bar{C})$. The theorem follows from the estimates for $\Pr(\bar{B})$ and $\Pr(\bar{C})$ provided by Lemmas 6.2 and 6.3.

To complete the proof, we need to show that any graph $G = G_{n,p}$ that satisfies B, C, and $\bar{A}_1$ must also satisfy A_2. Our strategy is to demonstrate that every $(n-1)$-matching M in G can be extended to a perfect matching of G by augmentation along a *short* alternating path. (An *alternating path* is a path in G whose edges lie alternately inside and outside the matching M.) Since every $(n-1)$-matching is "close to" some perfect matching, the ratio of $(n-1)$-matchings to perfect matchings cannot be very large. (A similar technique was used in the proof of Theorem 3.2.)

So let M be any $(n-1)$-matching in G. For $s \in U+V$, $T \subset U+V$ and i a positive integer, denote by $\Gamma^i_{\text{alt}}(s, T)$ the set of vertices in T that can be reached from vertex s by an alternating path of length at most i. Set $L = (1+\varepsilon_1(n)) \ln n/\ln pn$, where ε_1 is a positive real function that tends to zero (and that will be defined implicitly later in the proof). We will prove that M can be extended to a perfect matching via a path of length at most $8L$.

Let $u \in U$, $v \in V$ be the vertices of G that are left uncovered by M. Consider the set $\Gamma^{2L-1}_{\text{alt}}(u, V)$. If $v \in \Gamma^{2L-1}_{\text{alt}}(u, V)$ then we are done, so assume the contrary. For any i in the range $1 \leqq i < L$, we have the inequality $|\Gamma^{2i}_{\text{alt}}(u, U)| > |\Gamma^{2i-1}_{\text{alt}}(u, V)|$. To see this, note that $|\Gamma^{2i-1}_{\text{alt}}(u, V)|$ vertices in U can be reached from vertices in $\Gamma^{2i-1}_{\text{alt}}(u, V)$ via a single edge in M, and that these vertices do not include u, which can be reached by the null alternating path. Moreover, $|\Gamma^{2i+1}_{\text{alt}}(u, V)| \geqq |\Gamma^{2i}_{\text{alt}}(u, U)|$ since G is assumed to contain a perfect matching. (This is the trivial direction of Hall's Theorem.) Putting

the inequalities together we obtain $|\Gamma_{\text{alt}}^{2i+1}(u, V)| > |\Gamma_{\text{alt}}^{2i-1}(u, V)|$, and, by iteration, $|\Gamma_{\text{alt}}^{2L-1}(u, V)| \geqq L$.

We continue the process of computing lower bounds on $|\Gamma_{\text{alt}}^{i}(u, V)|$ for increasing i, but now using the improved expansion factor provided by Lemma 6.2. (Note that $k = |\Gamma_{\text{alt}}^{2L-1}(u, V)| \geqq L > \ln n / \ln pn$, the threshold stipulated in the lemma.) For $i \geqq L$ we have $|\Gamma_{\text{alt}}^{2i+1}(u, V)| \geqq \min\{\alpha |\Gamma_{\text{alt}}^{2i-1}(u, V)|, \lceil n/2 \rceil\}$, where $\alpha = pn/\ln\ln n$. Thus, $|\Gamma_{\text{alt}}^{4L-1}(u, V)| \geqq \min\{|\Gamma_{\text{alt}}^{2L-1}(u, V)|\alpha^L, \lceil n/2 \rceil\}$. Since

$$\alpha^L = \exp\left\{(1+\varepsilon_1(n))\frac{\ln n}{\ln pn}(\ln pn - \ln\ln\ln n)\right\} \geqq n$$

for suitably chosen $\varepsilon_1(n) \to 0$, we deduce that $|\Gamma_{\text{alt}}^{4L-1}(u, V)| \geqq \lceil n/2 \rceil$. A symmetrical argument gives $|\Gamma_{\text{alt}}^{4L-1}(v, U)| \geqq \lceil n/2 \rceil$. Since some pair of vertices in $\Gamma_{\text{alt}}^{4L-1}(u, V)$ and $\Gamma_{\text{alt}}^{4L-1}(v, U)$ must be connected by an edge of M, there must exist an augmenting path for M of length not greater than $8L-1$.

Finally, associate with each $(n-1)$-matching M of G a perfect matching $\bar{M}$ that is reachable from M via an augmenting path of length at most $8L-1$. For each perfect matching $P \in M_n(G)$, let $\mathcal{K}(P) = \{M \in M_{n-1}(G): \bar{M} = P\}$ be the set of $(n-1)$-matchings associated with P; clearly, $\{\mathcal{K}(P): P \in M_n(G)\}$ is a partition of the set $M_{n-1}(G)$. To complete the proof, it is sufficient to show that the cardinality of $\mathcal{K}(P)$ is bounded above by n^{10}, for sufficiently large n.

Let M be an element of $\mathcal{K}(P)$. By definition, M can be reached from P by unwinding an alternating path of length less than $8L$. We can view the construction of such an alternating path as a sequence of choices. First select one of the n vertices of U as a starting point. Then, at each of at most $4L$ points during the tracing of the path, namely, each time the path visits a vertex v in V, select one of at most $pn \ln n$ possible next moves: either terminate the path at v, or extend it along one of the $pn \ln n - 1$ free edges incident at v. (Recall that G has maximum degree $pn \ln n$, and note that moves from U to V are forced.) Thus the number of possible augmenting paths, and hence the cardinality of $\mathcal{K}(P)$, is bounded above by

$$n(pn \ln n)^{4L} = n \exp\{4L(\ln pn + \ln\ln n)\} \leqq n \exp\{8L \ln pn\} = n^{1+8(1+\varepsilon_1(n))}.$$

Since $\varepsilon_1(n) \to 0$ as $n \to \infty$, the cardinality of $\mathcal{K}(P)$ is bounded by n^{10} for sufficiently large n. □

Remark. Neither event A_1 nor A_2 need, in isolation, occur with overwhelming probability, only their disjunction. This can be demonstrated by setting $p = \beta n^{-1} \ln n$, where β is any constant greater than 1. □

The condition on p in Theorem 6.4 is an unfortunate blemish. Alan Frieze [34] has indicated that the condition can be dropped at the expense of a slight weakening of the conclusion: q-amenability would now hold with probability tending to 1 as n tends to infinity, rather than with overwhelming probability.

We close the section by providing proofs of the three technical lemmas, using standard techniques from the theory of random graphs.

Proof of Lemma 6.1. The probability that $G_{n,p}$ has a perfect matching is certainly less than the probability that no vertex in U is isolated (has degree zero) so it is enough to bound the latter probability. Our calculations will make free use of the inequalities $1 - t \geqq \exp(-t - t^2)$ and $1 - t \leqq \exp(-t)$, the first of which is valid for $0 \leqq t < 0.69$, and the second valid unconditionally [7, p. 5]. First consider the probability that a *particular*

vertex $u \in U$ is isolated:

$$\begin{aligned}\Pr(u \text{ is isolated}) &= (1-p)^n \\ &\geqq \exp\{n(-p-p^2)\} \\ &\geqq \exp\{-(1-\varepsilon)\ln n - n^{-1}\ln^2 n\} \\ &= n^{-(1-\varepsilon)} + o(n^{-1}).\end{aligned}$$

Thus, the probability that *no* vertex in U is isolated is $\{1-(1-p)^n\}^n \leqq \exp\{-n(1-p)^n\} \leqq \exp\{-n^{\varepsilon} + o(1)\}$. □

Proof of Lemma 6.2. Denote by A_{km} the event that $G_{n,p}$ is $(k, m+1)$-expanding. It is clearly enough to show that, for *arbitrary* k, m satisfying the given inequalities, the event A_{km} occurs with overwhelming probability. Furthermore, since $\Pr(A_{km})$ increases monotonically with k, it is sufficient to show that the event A_{km} occurs with overwhelming probability for k and m satisfying $\ln n/\ln pn \leqq k \leqq \lceil n/2\alpha \rceil$, and $m \leqq \alpha k$.

Let U' be an arbitrary k-subset of U. The set of vertices in V which are adjacent to U' in $G_{n,p}$ may be modelled as a sequence of n Bernoulli trials with success probability $q = 1-(1-p)^k$. Thus the probability that U' is adjacent to at most m vertices in V is

$$\sum_{t=0}^{m} \binom{n}{t} q^t (1-q)^{n-t},$$

and the probability $\Pr(\bar{A}_{km})$ that $G_{n,p}$ *fails* to be $(k, m+1)$-expanding is bounded above by

$$2\binom{n}{k} \sum_{t=0}^{m} \binom{n}{t} q^t (1-q)^{n-t}. \tag{20}$$

By Chernoff's bound [12, p. 18] and using the inequality

$$\binom{n}{k} \leqq \frac{1}{\sqrt{2\pi k}} \left(\frac{en}{k}\right)^k \leqq \frac{1}{2}\left(\frac{en}{k}\right)^k$$

the failure probability (20) may be bounded as follows:

$$\Pr(\bar{A}_{km}) \leqq \exp\left\{(n-m)\ln\frac{(1-q)n}{n-m} + m\ln\frac{qn}{m} + k\ln\frac{en}{k}\right\}. \tag{21}$$

Since $q = 1-(1-p)^k$, we have the relations $1-q \leqq \exp(-pk)$ and $q \leqq pk$; employing these in inequality (21), we obtain

$$\Pr(\bar{A}_{km}) \leqq \exp\left\{-pk(n-m) + (n-m)\ln\frac{n}{n-m} + m\ln\frac{pkn}{m} + k\ln\frac{en}{k}\right\}.$$

Further simplification, using the fact that $\ln(n/(n-m)) \leqq m/(n-m)$, yields

$$\Pr(\bar{A}_{km}) \leqq f(p, k, m) = \exp\left\{-pk(n-m) + m\left(1 + \ln\frac{pkn}{m}\right) + k\ln\frac{en}{k}\right\}. \tag{22}$$

Our goal is to bound $\Pr(\bar{A}_{km})$ by maximising $f(p, k, m)$ (viewed as a function of three real variables) over the ranges $p \geqq (1+\varepsilon)n^{-1}\ln n$, $\ln n/\ln pn \leqq k \leqq \lceil n/2\alpha \rceil$, and $m \leqq \alpha k$.

By differentiating (22) with respect to m we discover that, with p, k fixed and n sufficiently large, $f(p, k, m)$ is an increasing function of m. (Indeed it is sufficient for n to be greater than 15, guaranteeing $m \leqq pkn/\ln\ln n < pkn$.) Thus, in attempting to bound $f(p, k, m)$, it is enough to consider those triples (p, k, m) for which the inequality

that governs m, namely $m \leqq \alpha k$, is actually an equality. Substituting $k = m/\alpha = m \ln\ln n/pn$ in (22), our task is now to bound

$$f_1(p, m) = \exp\left\{-m \ln\ln n\left(1-\frac{m}{n}\right) + m(\ln\ln\ln n + 1) + \frac{m \ln\ln n}{pn} \ln \frac{epn^2}{m \ln\ln n}\right\}$$

over feasible p, m. Now, for fixed m, the function f_1 decreases with p. (For this we need the inequality $m \leqq pkn/\ln\ln n$.) So making the substitution $p = (1+\varepsilon)n^{-1} \ln n$ we are further reduced to bounding the function

$$f_2(m) = \exp\left\{-m \ln\ln n\left(1-\frac{m}{n}\right) + m(\ln\ln\ln n + 1) + \frac{m \ln\ln n}{(1+\varepsilon) \ln n} \ln \frac{e(1+\varepsilon)n \ln n}{m \ln\ln n}\right\}$$

over feasible m. The argument to the exponential is a convex function of m, so we can bound $f_2(m)$ by considering its values at the extremes of m's range. A lower bound for m is given by the chain of inequalities

$$m = \alpha k = \frac{pkn}{\ln\ln n} \geqq \frac{pn \ln n}{\ln\ln n \ln pn} \geqq \left(\frac{\ln n}{\ln\ln n}\right)^2 = m_{\min},$$

and an upper bound by

$$m = \alpha k \leqq \alpha\left\lceil \frac{n}{2\alpha} \right\rceil \leqq \frac{3n}{4} = m_{\max}$$

where we have used the known bounds on k and p, and assumed n sufficiently large. Substituting these extreme values in the expression for $f_2(m)$, we obtain

$$f_2(m_{\min}) = \exp\left\{-\left(\frac{\varepsilon}{1+\varepsilon} - o(1)\right)\frac{\ln^2 n}{\ln\ln n}\right\},$$

$$f_2(m_{\max}) = \exp\left\{-\left(\frac{3}{16} - o(1)\right) n \ln\ln n\right\}.$$

The former bound is the weaker and hence is the one that determines the overall bound on $f(p, k, m)$. □

Proof of Lemma 6.3. It is clearly enough to show that, with overwhelming probability, the degree of an arbitrary vertex $u \in U$ is bounded by $pn \ln n$. The set of vertices adjacent to u may be modelled as a sequence of n Bernoulli trials with success probability p. The probability that $\delta(u)$, the degree of u, exceeds m can be estimated from Chernoff's bound, using manipulations similar to those in the proof of Lemma 6.2:

$$\begin{aligned}\Pr(\delta(u) > m) &\leqq \exp\left\{(n-m) \ln \frac{(1-p)n}{n-m} + m \ln \frac{pn}{m}\right\} \\ &\leqq \exp\left\{-p(n-m) + m + m \ln \frac{pn}{m}\right\} \\ &\leqq \exp\left\{-m\left(\ln \frac{m}{pn} - 1\right)\right\}.\end{aligned}$$

(Clearly we may assume $m < n$.) Now, substituting $pn \ln n$ for m and noting $p \geqq n^{-1} \ln n$, we obtain

$$\Pr(\delta(u) > pn \ln n) \leqq \exp\{-(1-o(1)) \ln^2 n \ln\ln n\},$$

which decays faster than the reciprocal of any polynomial in n. □

7. Miscellaneous remarks and open problems. (i) The existence of an fpras for the unrestricted permanent remains an intriguing open question. Whereas the requirement that the ratio $|M_{n-1}(G)|/|M_n(G)|$ be polynomially bounded arises very naturally from our methods, there seems to be no a priori reason to suspect that graphs that violate this condition are particularly hard to handle. Perhaps an fpras of a different kind can be found for graphs in which the ratio is large. Alternatively, it is conceivable that counting perfect matchings approximately in general graphs is hard, in the sense that the existence of an fpras for this problem would imply that NP = RP. (Hardness results of this kind for other structures appear in [16], [28].)

(ii) It would be interesting to know whether the ratio $|M_{n-1}(G)|/|M_n(G)|$ is polynomially bounded for other natural classes of graphs, immediately yielding an fpras for the number of perfect matchings. For example, this question is pertinent for families of regular lattices encountered in statistical physics. Much effort has been expended on counting perfect matchings in such graphs, and an elegant exact solution obtained for planar lattices (or indeed arbitrary planar graphs [19]). The three-dimensional case, however, remains open even in approximate form.

(iii) From a practical point of view, it would be interesting to know whether the conductance bounds we have derived can be significantly improved. We make no claim of optimality here, preferring to concentrate on giving a clear exposition of the rapid mixing property. The practical utility of our algorithms, however, is likely to depend on rather tighter bounds being available.

Similar considerations apply to our methods for estimating the expectation of a 0-1 random variable under the stationary distribution of a Markov chain. We have chosen to view the chain as a generator of independent samples, partly to simplify the statistical concepts involved and partly because the random generation problems are of interest in their own right. In contrast, Aldous [2] considers estimates derived by observing a Markov chain continuously and formulates the definition of rapid mixing directly in terms of the variance of such an estimate. This approach may lead to increased efficiency.

(iv) A wider issue is the extent to which the techniques of this paper can be applied to the analysis of natural Markov chains whose states are combinatorial structures other than matchings. Since these chains are usually time-reversible, the conductance characterisation of rapid mixing presented in § 2 can in principle be applied. We conjecture that this is possible in practice for other interesting chains, and that the path counting technique developed in this paper is a promising general approach for obtaining positive results. It is to be hoped that this will yield efficient random generation and approximate counting procedures for further structures. Moreover, it may lead to rigorous performance guarantees for Monte Carlo experiments in statistical physics, and a demystification of currently fashionable stochastic optimisation techniques such as simulated annealing.

Acknowledgment. The authors thank Leslie Valiant for bringing reference [25] to their attention.

REFERENCES

[1] D. ALDOUS, *Random walks on finite groups and rapidly mixing Markov chains*, Séminaire de Probabilités XVII, 1981/82, Springer Lecture Notes in Mathematics 986, Springer-Verlag, Berlin, New York, 1983, pp. 243-297.

[2] ———, *On the Markov chain simulation method for uniform combinatorial distributions and simulated annealing*, Probab. Engrg. Inform. Sci., 1 (1987), pp. 33-46.

[3] D. ALDOUS AND P. DIACONIS, *Shuffling cards and stopping times*, Amer. Math. Monthly, 93 (1986), pp. 333-348.
[4] N. ALON, *Eigenvalues and expanders*, Combinatorica, 6 (1986), pp. 83-96.
[5] N. ALON AND V. D. MILMAN, λ_1, *isoperimetric inequalities for graphs and superconcentrators*, J. Combin. Theory Ser. B, 38 (1985), pp. 73-88.
[6] K. BINDER, *Monte Carlo investigations of phase transitions and critical phenomena*, in Phase Transitions and Critical Phenomena, Volume 5b, C. Domb and M. S. Green, eds., Academic Press, London, 1976, pp. 1-105.
[7] B. BOLLOBÁS, *Random Graphs*, Academic Press, London, 1985.
[8] A. Z. BRODER, *How hard is it to marry at random? (On the approximation of the permanent)*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 50-58; Erratum in Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, p. 551, Association for Computing Machinery, New York.
[9] P. DAGUM, M. LUBY, M. MIHAIL, AND U. V. VAZIRANI, *Polytopes, permanents and graphs with large factors*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computing Society, Washington, DC, 1988.
[10] P. DIACONIS AND M. SHAHSHAHANI, *Generating a random permutation with random transpositions*, Z. Wahrsch. Verw. Gebiete, 57 (1981), pp. 159-179.
[11] J. EDMONDS, *Paths, trees and flowers*, Canad. J. Math., 17 (1965), pp. 449-467.
[12] P. ERDOS AND J. SPENCER, *Probabilistic Methods in Combinatorics.*, Academic Press, New York, 1974.
[13] B. HAJEK, *Cooling schedules for optimal annealing*, Math. Oper. Res., 13 (1988).
[14] O. J. HEILMANN AND E. H. LIEB, *Theory of monomer-dimer systems*, Comm. Math. Phys., 25 (1972), pp. 190-232.
[15] M. R. JERRUM, *Two-dimensional monomer-dimer systems are computationally intractable*, J. Statist. Phys., 48 (1987), pp. 121-134.
[16] M. R. JERRUM, L. G. VALIANT, AND V. V. VAZIRANI, *Random generation of combinatorial structures from a uniform distribution*, Theoret. Comput. Sci., 43 (1986), pp. 169-188.
[17] N. KARMARKAR, R. M. KARP, R. LIPTON, L. LOVÁSZ, AND M. LUBY, *A Monte Carlo algorithm to approximate the permanent*, preprint, 1988.
[18] R. M. KARP AND M. LUBY, *Monte-Carlo algorithms for enumeration and reliability problems*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computing Society, Washington, DC, 1983, pp. 56-64.
[19] P. W. KASTELYN, *Graph theory and crystal physics*, in Graph Theory and Theoretical Physics, F. Harary ed., Academic Press, London, 1967, pp. 43-110.
[20] J. KEILSON, *Markov Chain Models—Rarity and Exponentiality*, Springer-Verlag, Berlin, New York, 1979.
[21] S. KIRKPATRICK, C. D. GELLATT, AND M. P. VECCHI, *Optimisation by simulated annealing*, Science, 220 (1983), pp. 671-680.
[22] G. F. LAWLER AND A. D. SOKAL, *Bounds on the L^2 spectrum for Markov chains and Markov processes: A generalization of Cheeger's inequality*, Trans. Amer. Math. Soc., 309 (1988), pp. 557-580.
[23] L. LOVÁSZ AND M. D. PLUMMER, *Matching Theory*, North-Holland, Amsterdam, 1986.
[24] M. LUNDY AND A. I. MEES, *Convergence of an annealing algorithm*, Math. Programming, 34 (1986), pp. 111-124.
[25] M. MIHAIL, *On coupling and the approximation of the permanent*, Inform. Process. Lett., 30 (1989), pp. 91-96.
[26] H. MINC, *Permanents*, Addison-Wesley, Reading, MA, 1978.
[27] G. H. SASAKI AND B. HAJEK, *The time complexity of maximum matching by simulated annealing*, J. Assoc. Comput. Mach., 35 (1988), pp. 387-403.
[28] A. J. SINCLAIR, *Randomised algorithms for counting and generating combinatorial structures*, Ph.D. thesis, University of Edinburgh, Edinburgh, Scotland, June 1988.
[29] A. J. SINCLAIR AND M. R. JERRUM, *Approximate counting, uniform generation and rapidly mixing Markov chains*, Inform. Comput., 82 (1989), pp. 93-133.
[30] L. STOCKMEYER, *The complexity of approximate counting*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 118-126.
[31] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.
[32] L. G. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189-201.
[33] ———, *The complexity of enumeration and reliability problems*, SIAM J. Comput., 8 (1979), pp. 410-421.
[34] A. FRIEZE, *A note on computing random permanents*, unpublished manuscript.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1179–1187, December 1989

007

A CORRECTION NETWORK FOR N-SORTERS*

MANFRED SCHIMMLER† AND CHRISTOPH STARKE†

Abstract. A correction network C is introduced that can be added to an arbitrary N-input sorting net in order to achieve single-fault tolerance. Multiple (k) fault robustness is attained by adding C^k. For single-fault correction, C is proved to be asymptotically optimal.

Key words. sorting networks, comparators, fault tolerance, reliability, Hamming distance

AMS(MOS) subject classifications. 68C05, 68C25, 68E05

1. Introduction. The high integration density in VLSI technology has created an increasing probability of fabrication faults. Therefore, although it is cheap to produce large quantities of a single chip or wafer, only a small fraction of them can be expected to function correctly. One way of increasing the yield is by designing fault-tolerant algorithms.

Another aspect of fault tolerance is to increase the reliability of a chip. A useful technique is to introduce redundant components in order to keep the whole system reliable, even in the presence of several operating faults.

In this paper we present a method for adding a fault-correcting network to an arbitrary sorting net. It is very simple and needs little in the way of additional hardware and additional delay time. Furthermore, we will show that the number of additional comparators and the number of additional delay stages is asymptotically optimal for single-fault correction of arbitrary sorting nets.

The problem has already been investigated in [5]. The authors found a correction mechanism for k faults in an N-network, consisting of only $k(2N-3)$ additional comparators. Unfortunately, the number of additional delay stages is $k(2N-3)$, too. Therefore, even a single-fault tolerant N-sorter has time complexity $\Theta(N)$.

In [4] a multiple half-fault tolerant N-sorter is presented. For $N=2^n$ it consists of one block of the "balanced sorting network" [2], consisting of $\log N$ stages of $N/2$ comparators. The output of this block is recirculated back as input. If the block is fault free, sorting requires $\log N$ passes through the network. If it is not fault free, the number of necessary passes increases with the number of faulty comparators, but the network can still sort. There are n pairs of "critical" comparators. The network fails only if both comparators of such a pair are faulty.

2. Definitions. Let R denote the set of real numbers. An *N-comparator* (or comparator, if N is understood) is a pair $[i:j]$, with $0 \leqq i, j \leqq N-1$, $i \neq j$. Associate with the N-comparator $[i:j]$ the mapping from R^N to R^N defined by

$$\langle x_0, x_1, \cdots, x_{N-1}\rangle[i:j] := \langle x'_0, x'_1, \cdots, x'_{N-1}\rangle,$$

where $x'_p = x_p$ if $p \notin \{i, j\}$, $x'_i = \min(x_i, x_j)$ and $x'_j = \max(x_i, x_j)$. We call $[i:j]$ a *standard comparator* if $i < j$.

An *N-comparator stage* (or comparator stage if N is understood) s is a set of r N-comparators $\{[i_0:j_0], [i_1:j_1], \cdots, [i_{r-1}:j_{r-1}]\}$, where $i_0, j_0, i_1, j_1, \cdots, i_{r-1}, j_{r-1}$ are

* Received by the editors on September 28, 1987; accepted for publication (in revised form) January 30, 1989.

† Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, Olshausenstrasse 40, D-2300 Kiel 1, Federal Republic of Germany.

pairwise distinct. The N-comparator stage s defines a mapping from R^N to R^N which is the arbitrary sequential composition of the mappings associated with $[i_0:j_0], [i_1:j_1], \cdots, [i_{r-1}:j_{r-1}]$.

An *N-network* A is a sequence of N-comparator stages $s_1, s_2, \cdots, s_p$. The N-network A defines the mapping from R^N to R^N by successively applying the mappings induced by $s_1, s_2, \cdots, s_p$. A *standard N-network* is an N-network consisting of standard comparators only.

A vector $x = \langle x_0, x_1, \cdots, x_{N-1} \rangle \in \mathbf{R}^N$ is *sorted*, if and only if $x_{i-1} \leqq x_i$ for $1 \leqq i \leqq N-1$. An *N-sorter* A is an N-network that satisfies the following condition:

$$\forall x \in \mathbf{R}^N: xA = x' \Rightarrow x' \text{ is sorted.}$$

What follows is a very useful tool in the studies of sorting networks.

ZERO ONE PRINCIPLE. *Let A be an N-network. If xA is sorted for every* $x \in \{0, 1\}^N$, *then A is an N-sorter.*

Proof. See [3, 5.3.4, Thm. Z].

Using the Zero One Principle we can restrict our studies of sorting networks to input vectors $x \in \{0, 1\}^N$.

For $x \in \{0, 1\}^N$ the sorted version x_s denotes the nondecreasing sequence consisting of the same number of 0's and 1's as x:

$$x_s := xA \text{ for an arbitrary } N\text{-sorter } A.$$

Figure 1 shows, for example, the 4-sorter $\{[0:1], [3:2]\}, \{[0:2], [1:3]\}, \{[0, 1], [2:3]\}$, drawn as defined in [3, p. 222]: Comparators are drawn as vertical arrows between horizontal data lines. Elements to be sorted travel through the network from left to right, and whenever they meet a comparator they are compared and possibly interchanged such that the larger element appears at the line of the arrow head and the smaller one at the line of the arrow tail after passing the comparator. They arrive at the right end of the network in nondecreasing order.

3. Fault model. A correct comparator performs a comparison-exchange as depicted in Fig. 2a. We shall consider three different types of functional faults: Fig. 2b shows the *full-fault*, a comparator producing the maximum of its inputs instead of the minimum and vice versa. The *half-fault* comparator (Fig. 2c) leaves the inputs unchanged, and the *x-fault* (Fig. 2d) comparator exchanges the inputs independent of their values. Each of these faults is assumed to be static, i.e., the comparator always behaves in the same faulty way.

Of course, there are many other possible faults, as for example, dynamic faults, stuck-at faults, technology-dependent faults such as stuck-open faults in CMOS, etc. We do not want to discuss each of these fault types, but we are able to apply our correction method to arbitrary faults, if the considered sorter enables us to bypass faulty comparators.

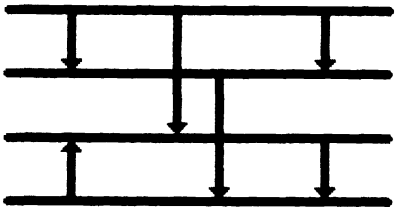


FIG. 1. *The bitonic 4-sorter* [1].

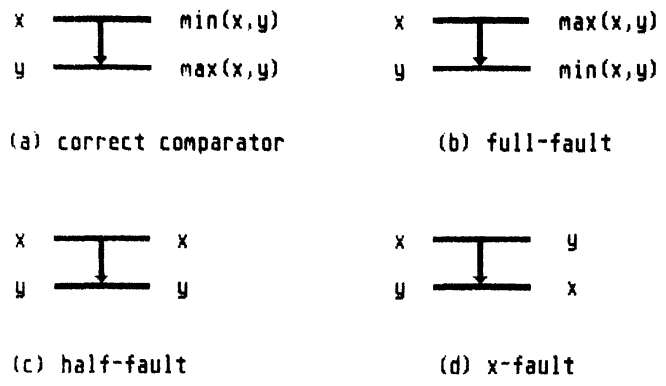


FIG. 2

Our aim is to find a solution for the half-fault correction problem. The reason for choosing this specific fault type is the fact that the mapping of a half-fault comparator is the identity (and thus it behaves like a simple storage cell). If we can locate the faults in a given sorting network, we can apply the following technique: we can bypass all of the faulty comparators by cutting out the comparator's logic and by connecting the input and the output lines, and we shall get a network that behaves exactly like the original sorting net, where every fault (of arbitrary type) is replaced by a half-fault. If we can add some half-fault correcting network (correcting an appropriate number of half-faults), the result is a fault-free sorter.

Of course, this construction is particularly useful, especially if the additional correction net is also allowed to contain faulty comparators, which can be bypassed if necessary. The same construction is not possible with full-faults or x-faults instead of half-faults, because the last stage of the resulting network could be faulty. In this case, a full-fault as well as an x-fault produces unsorted output sequences, which means that the network is not a correct sorter. In the remainder of the paper, a *k-fault N-sorter* denotes a network obtained from an N-sorter by replacing k comparators with half-fault comparators.

4. Lower bounds. Let C be an N-network that corrects any 1-fault N-sorter A', i.e.,

$$xA'C = x_s \quad \text{for every } x \in \{0,1\}^N.$$

In this section we will prove two lower bounds for C.

LEMMA 1. *The number of comparators of C is $\Omega(N)$.*

LEMMA 2. *The number of comparator stages of C is $\Omega(\log N)$.*

The proofs follow.

PROPOSITION 1. *For any i, j, $0 \leqq i < j \leqq N-1$ there is a 1-fault N-sorter A'_{ij} satisfying the following conditions for every $x \in \{0,1\}^N$:*

$$(x_s)_p = (xA'_{ij})_p \quad \text{for } p \notin \{i, j\},$$
$$(x_s)_i = (xA'_{ij})_j,$$
$$(x_s)_j = (xA'_{ij})_i.$$

In other words, A'_{ij} sorts every x except the items on positions i and j, which are exchanged.

Proof of Proposition 1. Take an arbitrary N-sorter S. If $i<j$, we define $A_{ij} = S$, $\{[j:i]\}, \{[i:j]\}$. A_{ij} is an N-sorter. Let A'_{ij} be A_{ij} with a half fault in the last comparator $[i:j]$. Obviously, A'_{ij} satisfies the conditions of Proposition 1. □

Proof of Lemma 1. From Proposition 1 we know the N-network A'_{0j} for every j, $1 \leqq j \leqq N-1$. An N-network C correcting an arbitrary A'_{0j} must be able to move the wrong item from position 0 to any position between 1 and $N-1$. Therefore, every position must appear at least once in a comparator of C. Hence, C consists of at least $N/2$ comparators. □

Proof of Lemma 2. Again we consider the N-networks A'_{0j} for $1 \leqq j \leqq N-1$. The correcting N-network C must be able to move the wrong item from position 0 to any position between 1 and $N-1$. Since every comparator connects only two lines, at least $\log N$ comparator stages are needed to obtain a path from position 0 to any other position. □

In § 6, the N-network C is presented, correcting every 1-fault N-sorter. C meets the lower bounds in the number of comparators and the number of comparator stages.

5. Symmetric networks. To simplify the proof of the Theorem in § 7 we need some more notations and a few lemmata.

For $x = \langle x_0, x_1, \cdots, x_{N-1}\rangle \in \{0,1\}^N$ we define the *complement*

$$\bar{x} := \langle \bar{x}_0, \bar{x}_1, \cdots, \bar{x}_{N-1}\rangle, \qquad \bar{x}_i := 1 - x_{N-1-i}, \qquad i \in \{0, 1, \cdots, N-1\}.$$

For a comparator $[i:j]$ the complement is defined by

$$\overline{[i:j]} = [N-1-j : N-1-i].$$

The complement $\bar{s}$ of a comparator stage $s = \{[i_0:j_0], [i_1:j_1], \cdots, [i_{r-1}:j_{r-1}]\}$ is defined by $\bar{s} := \{\overline{[i_0:j_0]}, \overline{[i_1:j_1]}, \cdots, \overline{[i_{r-1}:j_{r-1}]}\}$ and the complement of an N-network $A = s_1, s_2, \cdots, s_p$ is $\bar{A} = \bar{s}_1, \bar{s}_2, \cdots, \bar{s}_p$.

An N-network A is called *symmetric* if and only if $\bar{A} = A$.

LEMMA 3. *For* $x \in \{0,1\}^N$ *and* $0 \leqq i, j \leqq N-1$ *the following equations hold*:

(i) $\bar{\bar{x}} = x$,

(ii) $\overline{\overline{[i:j]}} = [i:j]$,

(iii) $\overline{x[i:j]} = \bar{x}[i:j]$.

Proof. (i) and (ii) are obvious.

(iii) Consider the pth component of $\overline{\bar{x}[i:j]}$:

$$(\overline{\bar{x}[i:j]})_p = (\overline{\bar{x}[N-1-j:N-1-i]})_p = 1 - \bar{x}[N-1-j:N-1-i]_{N-1-p}.$$

If $p \neq i$ and $p \neq j$, we have

$$1 - \bar{x}[N-1-j:N-1-i]_{N-1-p} = 1 - \bar{x}_{N-1-p} = x_p = x[i:j]_p.$$

If $p = i$, we have

$$\begin{aligned} 1 - \bar{x}[N-1-j:N-1-i]_{N-1-p} &= 1 - \max(\bar{x}_{N-1-j}, \bar{x}_{N-1-i}) \\ &= \min(1-\bar{x}_{N-1-j}, 1-\bar{x}_{N-1-i}) = \min(x_j, x_i) = x[i:j]_p. \end{aligned}$$

If $p = j$, we have

$$\begin{aligned} 1 - \bar{x}[N-1-j:N-1-i]_{N-1-p} &= 1 - \min(\bar{x}_{N-1-j}, \bar{x}_{N-1-i}) \\ &= \max(1-\bar{x}_{N-1-j}, 1-\bar{x}_{N-1-i}) = \max(x_j, x_i) = x[i:j]_p. \end{aligned}$$

Therefore, we know for every p, $0 \leqq p \leqq N-1$,

$$(\overline{\bar{x}[i:j]})_p = x[i:j]_p, \text{ which means } \overline{\bar{x}[i:j]} = x[i:j]. \qquad \square$$

LEMMA 4. *For every* N*-network* A *and every* $x \in \{0,1\}^N$: $\overline{\bar{x}A} = xA$.

Proof. Induction on the total number m of comparators in A:

$m = 0$: $\bar{x}\bar{A} = \bar{\bar{x}} = x = xA$;

$m > 0$: Assume Lemma 4 to be true for every N-network consisting of no more than $m-1$ comparators.

Let A consist of the comparators $[i_1:j_1]$, $[i_2:j_2]$, $\cdots$, $[i_m:j_m]$. Without loss of generality, $[i_m:j_m]$ is the last stage of A. Define G to be the N-network obtained by removing $[i_m:j_m]$ from A. Then

$$\overline{\bar{x}\bar{A}} = \overline{\overline{\bar{x}\bar{G}[i_m:j_m]}} = \overline{\overline{\bar{x}\bar{G}}[i_m:j_m]} = \overline{\bar{x}\bar{G}}[i_m:j_m] = xG[i_m:j_m] = xA. \qquad \square$$

SYMMETRIC NETWORK LEMMA. *For every symmetric N-network A and every* $x \in \{0,1\}^N$, $\overline{\bar{x}A} = xA$.

Proof. Definition of symmetric networks and Lemma 4. $\square$

6. The correction network. Let n be an integer greater than 1 and let $N = 2^n$. In this section we shall define the N-network C that can be used for single half-fault correction. We show that C consists of $2\log N - 1$ comparator stages, and of $3.5N - 2\log N - 5$ comparators.

DEFINITION. Let $N = 2^n$. The *correcting N-network* C is defined by

$$C := s_1, s_2, \cdots, s_{2n-1} \quad \text{with}$$

$$s_1 := \{[2i:2i+1] \mid 0 \leqq i \leqq 2^{n-1}-1\}$$

$$s_j := \{[2^j i : 2^j i + 2^{j-1}] \mid 0 \leqq i \leqq 2^{n-j}-1\} \cup$$

$$\{[N-1-(2^j i + 2^{j-1}) : N-1-2^j i] \mid 0 \leqq i \leqq 2^{n-j}-1\}$$

$$\text{for } 2 \leqq j \leqq n-1$$

$$s_j := \{[2^{2n-j-1} i - 2^{2n-j-2} : 2^{2n-j-1} i] \mid 1 \leqq i \leqq 2^{j-n+1}-1\} \cup$$

$$\{[N-1-2^{2n-j-1} i : N-1-(2^{2n-j-1} i - 2^{2n-j-2})] \mid 1 \leqq i \leqq 2^{j-n+1}-1\}$$

$$\text{for } n \leqq j \leqq 2n-3$$

$$s_{2n-2} := \{[2i-1:2i] \mid 1 \leqq i \leqq 2^{n-1}-1\}$$

$$s_{2n-1} := \{[2i:2i+1] \mid 0 \leqq i \leqq 2^{n-1}-1\}$$

As an example, Fig. 3 shows the network C for $N = 32$ $(n = 5)$.

To assure that C is well defined, we need to prove the following proposition.

PROPOSITION 2. *For every* $r \in \{1, 2, \cdots, 2n-1\}$ *and for all comparators* $[i_0:j_0]$ *and* $[i_1:j_1]$ *occurring in* s_r, i_0, i_1, j_0, j_1 *are pairwise distinct.*

Proof. For stages 1, $2n-2$, and $2n-1$ the proposition is obviously true. For $2 \leqq p \leqq n-1$ the comparators of the pth comparator stage are of the form

(*) $\quad [(2i)2^{p-1} : (2i+1)2^{p-1}]$ or

(*) $\quad [N-1-(2i+1)2^{p-1} : N-1-(2i)2^{p-1}]$ for $0 \leqq i \leqq 2^{n-p}-1$.

For two different comparators of the form (*) the proposition is true. The same holds for two comparators of the form (**). Now let $[i_0:j_0]$ be a (*) type comparator and $[i_1:j_1]$ a (**) type one. We have $i_0 \neq j_0$ as well as $i_1 \neq j_1$. Since i_0 and j_0 are even, whereas i_1 and j_1 are odd, they must be pairwise distinct. The same argument holds for the pth comparator stage where $n \leqq p \leqq 2n-2$, which completes the proof. $\square$

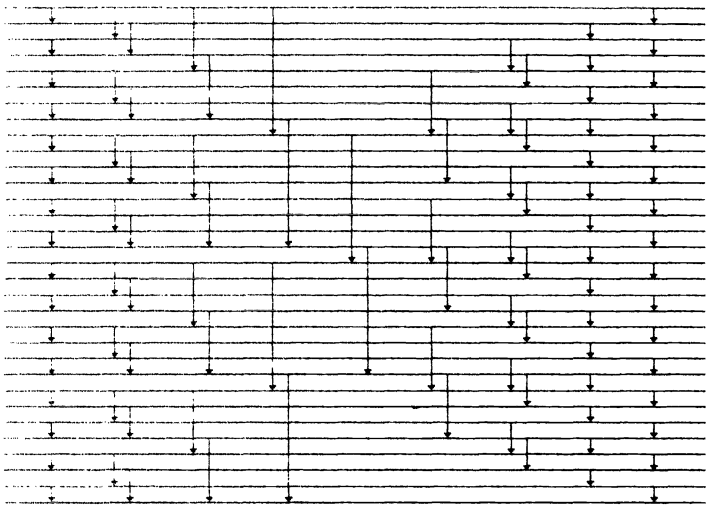


FIG. 3. *The correction* N-network C *for* $N = 32$.

Now we count the total number of comparators in C. The first stage consists of 2^{n-1} comparators. For $2 \leqq p \leqq n-1$ comparator stage p has $2 * 2^{n-p}$ comparators. For $n \leqq p \leqq 2n-3$ stage p consists of $2 * (2^{p-n+1}-1)$ comparators. Stages $2n-2$ and $2n-1$ have $2^{n-1}-1$ and 2^{n-1} comparators, respectively. Therefore, the total number of comparators in C is

$$m = 2^{n-1} + 2\sum_{p=2}^{n-1} 2^{n-p} + 2\sum_{p=n}^{2n-3} (2^{p-n+1}-1) + 2^{n-1} - 1 + 2^{n-1} = 3.5 * 2^n - 2n - 5.$$

Observe that C is a symmetric standard network. In particular each comparator stage s_j of C is symmetric and so is every subnetwork $s_1, s_2, \cdots, s_j$, for $1 \leqq j \leqq 2n-1$.

7. Correction property of C. The *Hamming distance* of two N bit vectors x and y is the number of bits in which they differ: $D(x, y) = \sum_{i=1}^{N-1} |x_i - y_i|$.

LEMMA 5. *Let* $x \in \{0,1\}^N$ *and let* $[i:j]$ *be a standard comparator in an* N*-network. Then* $D(x[i:j], x_s) \leqq D(x, x_s)$.

Proof. See [5, Lem. 2].

LEMMA 6. *Let* A' *be a* k*-fault* N*-sorter. Then for any* $x \in \{0,1\}^N$, $D(xA', x_s) \leqq 2k$.

Proof. See [5, Lem. 4].

THEOREM. *Let* $x \in \{0,1\}^N$, *with* $D(x, x_s) > 0$. *For the correcting* N*-network* $C = (s_1, s_2, \cdots, s_{2n-1})$ *the following relation holds*: $D(xC, x_s) \leqq D(x, x_s) - 2$.

Proof. Let b be the number of 0's in x, i.e., $x_s = 0^b 1^{N-b}$. Since $D(x, x_s) > 0$, b must be greater than 0 and smaller than N. We call the set of positions $\{0, 1, \cdots, b-1\}$ the *zero area*, and the set $\{b, b+1, \cdots, N-1\}$ the *one area*. A *wrong* 1 is a 1 positioned in the zero area and a *wrong* 0 a 0 in the one area.

If there is any comparator in C that compares a wrong 1 with a wrong 0, these two elements change their positions and the Hamming distance to the sorted sequence is reduced by two. Since the Hamming distance cannot be enlarged again by standard comparators (Lemma 5), after passing the network C we would have $D(xC, x_s) \leqq D(x, x_s) - 2$.

Therefore, it is sufficient to construct a contradiction to the following assumption:

(*) There is no wrong 1 compared with a wrong 0 in C.

Let P_0 be the position of the first wrong 0 in x, i.e.,

$$P_0 = \min\{p \in \{b, b+1, \cdots, N-1\} \mid x_p = 0\}.$$

We want to construct the sequence of positions $P_0, P_1, \cdots, P_{2n-2}$, which we shall show to be the path of this 0 through the network C:

Let $p_{n-1}p_{n-2}\cdots p_0$ be the binary representation of P_0, i.e.,

$$P_0 = \sum_{i=0}^{n-1} p_i 2^i.$$

Since $P_0 \geqq b$, we can define the constant K:

$$K := \max\left\{ j \;\middle|\; 0 \leqq j \leqq n-1 \quad \text{and} \quad \sum_{i=j}^{n-1} p_i 2^i \geqq b \right\}.$$

Now we define the sequence of positions P_j:

$$\text{for } 0 \leqq j \leqq K, \quad P_j := \sum_{i=j}^{n-1} p_i 2^i;$$

$$\text{for } K+1 \leqq j \leqq 2n-K-2, \quad P_j := P_K;$$

$$\text{for } 2n-K-1 \leqq j \leqq 2n-2, \quad d_j := \max\{i \in \{0,1\} \mid P_{j-1} - i * 2^{2n-2-j} \geqq b\},$$

$$P_j := P_{j-1} - d_j * 2^{2n-2-j}.$$

d_j is well defined because $P_{j-1} \geqq b$.

We need some observations concerning the P_j:

(1) For $0 \leqq j \leqq K$, P_j is a multiple of 2^j.

(2) For $0 \leqq j \leqq K$, $P_j = P_{j-1} - p_{j-1}2^{j-1}$.

The definition of K implies

$$P_K = \sum_{i=K}^{n-1} p_i 2^i \geqq b > \sum_{i=K+1}^{n-1} p_i 2^i.$$

Therefore, $0 \leqq P_K - b < p_K * 2^K$, which implies $p_K = 1$ and

(3) $$0 \leqq P_K - b < 2^K.$$

For $j = 2n-K-2$ we have $P_j = P_K$. From (1) we know that P_j is a multiple of 2^{2n-2-j}. For $2n-K-1 \leqq j \leqq 2n-2$ we have

$$\begin{aligned} P_j &= P_K - \sum_{i=2n-K-1}^{j} d_i 2^{2n-2-i} \\ &= 2^K \sum_{i=K}^{n-1} p_i 2^{i-K} - 2^{2n-2-i} * \sum_{i=2n-K-1}^{j} d_i 2^{j-i} \\ &= 2^{2n-2-j} * \left(2^{K-2n+2+j} * \sum_{i=K}^{n-1} p_i 2^{i-K} - \sum_{i=2n-K-1}^{j} d_i 2^{j-i} \right). \end{aligned}$$

Therefore, we know

(4) $$\forall j, \quad 2n-K-2 \leqq j \leqq 2n-2 \quad \exists \text{ integer } m_j : P_j = m_j * 2^{2n-2-j},$$
$$\text{where } 1 \leqq m_j \leqq 2^{j-n+2}, \quad \text{because } 0 < P_j = m_j * 2^{2n-2-j} < 2^n.$$

Now we want to prove that the sequence of positions P_j is the path of the first wrong 0 through the network:

$$\forall j,\quad 0 \leqq j \leqq 2n-2\colon x(s_1, s_2, \cdots, s_j)_{P_j} = 0. \tag{5}$$

Proof of (5) *by induction on j.*

$j=0$: By definition of P_0 we have $x_{P_0}=0$.

$j>0$: Assume $x(s_1, s_2, \cdots, s_{j-1})_{P_{j-1}}=0$. We have to consider three cases:

Case 1. $1 \leqq j \leqq K$; s_j contains the comparator $[P_j : P_j + 2^{j-1}]$, because P_j is a multiple of 2^j (see (1)). Since $P_j = P_{j-1} - P_{j-1} * 2^{j-1}$ (see (2)) and since $x(s_1, s_2, \cdots, s_{j-1})_{P_{j-1}}=0$, at least one input into the comparator is 0. Therefore, $x(s_1, s_2, \cdots, s_j)_{P_j}=0$.

Case 2. $K+1 \leqq j \leqq 2N-K-2$; by the definition of C, for every comparator $[r:s]$ in stage s_j we know $s-r>2^K$. On the other hand, we know from (3) that $2^K > P_K - b$. Since $P_K = P_{j-1} = P_j$, this implies that a comparator of the form $[r : P_{j-1}]$ in s_j compares an element from the zero area with a wrong 0 in position P_{j-1} in the one area. Assumption (*) forces this element to be a 0, which implies $x(s_1, s_2, \cdots, s_j)_{P_j}=0$. If there is a comparator of the form $[P_{j-1} : s]$ in s_j, then the 0 remains in P_j and again we have $x(s_1, s_2, \cdots, s_j)_{P_j}=0$.

Case 3. $2n-K-1 \leqq j \leqq 2n-2$; if $d_j=0$, then $P_j = P_{j-1}$ and $P_{j-1} - 2^{n-2-j} < b$. If there is a comparator $[r : P_{j-1}]$ in s_j, then $r<b$ and $x(s_1, s_2, \cdots, s_{j-1})_r$ must be 0, due to assumption (*). Therefore, $x(s_1, s_2, \cdots, s_j)_{P_j}=0$. If $d_j=1$, then $P_j = P_{j-1} - 2^{2n-2-j}$. By (4), there is an integer m_{j-1}, $1 \leqq m_{j-1} \leqq 2^{j-n+1}$, with $P_{j-1} = m_{j-1} * 2^{2n-j-1}$. Therefore, in stage s_j there is a comparator $[P_j : P_{j-1}]$, which implies that $x(s_1, s_2, \cdots, s_j)_{P_j}=0$.

For $2n-K-2 \leqq j \leqq 2n-2$ we need an additional statement concerning the distance of P_j from the zero area:

$$\forall j,\quad 2n-K-2 \leqq j \leqq 2n-2\colon 0 \leqq P_j - b < 2^{2n-2-j}. \tag{6}$$

Proof of (6) *by induction on j.*

$j=2n-K-2$: from (3) we know $0 \leqq P_K - b < 2^K = 2^{2n-2-j}$.

$j>2n-K-2$: assume (6) to be true for $j-1$. $P_j = P_{j-1} - d_j * 2^{2n-2-j} \geqq b$. If $d_j=0$, then $P_j = P_{j-1}$ and $P_{j-1} - 2^{2n-2-j} < b$. Thus we have $0 \leqq P_j - b \leqq 2^{2n-2-j}$.

If $d_j=1$, then we know by induction hypothesis

$$P_j = P_{j-1} - 2^{2n-j-2} < b + 2^{2n-1-j} - 2^{2n-2-j} = b + 2^{2n-2-j}.$$

This implies $0 \leqq P_j - b \leqq 2^{2n-2-j}$.

Equation (6), in particular, implies that for $j=2n-2$ holds $0 \leqq P_{2n-2} - b < 1$, or in other terms $P_{2n-2}=b$. From (5) we know $x(s_1, s_2, \cdots, s_{2n-2})_{P_{2n-2}}=0$, and thus we get

$$x(s_1, s_2, \cdots, s_{2n-2})_b = 0. \tag{7}$$

Since the complement $\bar{x}$ consists of $N-b$ 0's and b 1's, we get from equation (7)

$$\bar{x}(s_1, s_2, \cdots, s_{2n-2})_{N-b}=0, \quad \text{and therefore,} \quad \overline{\bar{x}(s_1, s_2, \cdots, s_{2n-2})}_{N-1-(N-b)}=1.$$

Because of the Symmetric Network Lemma, this is equivalent to

$$x(s_1, s_2, \cdots, s_{2n-2})_{b-1}=1, \tag{8}$$

since $s_1, s_2, \cdots, s_{2n-2}$ is a symmetric network.

Now (7) and (8) show that there is a 0 in position b and a 1 in position $b-1$ after $2n-2$ stages. Since in stage $2n-2$ there are the comparators $[2i-1 : 2i]$, they would have been interchanged if b is even. Hence b must be odd. But then in stage

s_{2n-1}, there is a comparator $[b-1:b]$. This comparator compares a wrong 1 with a wrong 0, a contradiction to (*). □

COROLLARY. *An N-sorter S is given. If in the network SC^k at most k comparators are replaced by half-fault comparators, then the resulting network is still an N-sorter.*

Proof. Let r be the number of comparators replaced in S. Then at most $k-r$ copies of C are faulty and at least r copies of C are fault free. Lemma 6 ensures that for the network S' obtained from S by replacing the r comparators the following condition holds: $D(xS', x_s) \leqq 2r$.

By the theorem, each of the fault-free copies of C reduces the Hamming distance by at least 2. Therefore, after passing the r fault-free copies, the sequence is sorted. □

8. Conclusions. We have introduced an efficient way to achieve fault tolerance in sorting networks. We have shown the asymptotic optimality of our method for single half-fault correction. It is certainly of further interest to find lower bounds for multiple half-fault correction and correction networks that meet these bounds.

9. Acknowledgments. We would like to thank the referees for helpful suggestions and remarks.

REFERENCES

[1] K. E. BATCHER, *Sorting networks and their applications*, AFIPS Spring Joint Computer Conference, Montvale, NJ, 23 (1968), pp. 307–314.

[2] M. DOWD, Y. PERL, L. RUDOLPH, AND M. SAKS, *The balanced sorting network*, in Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983, pp. 161–172.

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

[4] L. RUDOLPH, *A Robust Sorting Network*. IEEE Trans. Comput. C-34, (1985), pp. 326–335.

[5] A. C. YAO AND F. E. YAO, *On fault tolerant networks for sorting*, SIAM J. Comput., 14 (1985), pp. 120–128.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1188–1200, December 1989

008

SOLUTION OF A DIVIDE-AND-CONQUER MAXIMIN RECURRENCE*

ZHIYUAN LI†‡ AND EDWARD M. REINGOLD†

Abstract. The solution of the divide-and-conquer recurrence

$$M(n) = \max_{1 \leq k < n} (M(k) + M(n-k) + \min(f(k), f(n-k)))$$

is found for a variety of functions f. Asymptotic bounds on $M(n)$ are found for arbitrary nondecreasing f, and the exact form of $M(n)$ is determined for f nondecreasing and weakly concave. As a corollary to the asymptotic bounds, it is shown that $M(n)$ remains linear even when f is almost linear. Among the exact forms determined: For $f(x) = \lfloor \lg x \rfloor$, the solution is $M(n) = (M(1)+1)n - \lfloor \lg n \rfloor - \nu(n)$ where $\nu(n)$ is the number of 1-bits in the binary representation of n. For $f(x) = \lceil \lg x \rceil$, the solution is $M(n) = (M(1)+1)n - \lceil \lg n \rceil - 1$, while for $f(x) = \lceil \lg (x+1) \rceil$, the solution is $M(n) = (M(1)+2)n - \lfloor \lg n \rfloor - \nu(n) - 1$.

Key words. recurrence relations, divide and conquer, algorithmic analysis

AMS(MOS) subject classifications. 68Q25, 68R05, 05A20, 26A12

1. Introduction.

Let $M(n)$ be defined by the recurrence

$$M(n) = \max_{1 \leq k < n} (M(k) + M(n-k) + \min(f(k), f(n-k))), \tag{1}$$

with $M(1)$ given. Divide-and-conquer recurrence relations of this type, for various functions f, occur in a variety of problems in the analysis of algorithms. For instance, the number of element interchanges in the worst case of quicksort (see, for example, [15] or [16]) is given by $M(n)$ when $f(x) = x$ and $M(1) = 0$; this $M(n)$ counts the worst-case number of interchanges used by quicksort in sorting $n-1$ elements, provided we make the simplifying assumption that the element used for partitioning is *always* interchanged to put it between the left and right parts of the array. The identical recurrence relation describes the worst-case behavior of a certain permutation algorithm (see [8]) and a merging process (see [14]).

The case $f(x) = \Theta(\log x)$ is well known. The recurrence (1) occurs, with such an f, in the construction of binary search trees [6], [15], [16], in the finding of common ancestors [10], and several computational-geometric problems [3], [5], [7], [9], [12], [21]. $M(n)$ is $\Theta(n)$ by an inductive argument [15], but the behavior of $M(n)$ in this case has not been investigated more precisely.

There is even a version of the recurrence (1) over trees. In this version, which occurs in the analysis of a certain tree-drawing algorithm, the function f is the height of the tree. See [18] or [22] for details.

Our purpose in this paper is to obtain precise solutions to the recurrence (1) under a number of choices of f. These solutions will be instructive for several reasons. First, the order-of-magnitude approximations conceal the precise behavior of the algorithms being analyzed. Second, and most important, by giving a more precise analysis of (1), we will discover that $f(x) = \Theta(\log x)$ is a *much* stronger condition than is necessary to guarantee that $M(n)$ is linear—any $f(x) = O(x/(\log x)^{1+\varepsilon})$, $\varepsilon > 0$, will do. This

* Received by the editors May 31, 1988; accepted for publication (in revised form) December 12, 1988.

† Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, Illinois 61801.

‡ This author is affiliated with the Center for Supercomputing Research and Development, University of Illinois and his work was supported in part by National Science Foundation grant NSF MIP-8410110, by Department of Energy grant DOE DE-FG02-85ER25001, and by the Control Data Corporation.

means that the "merge steps" in the divide-and-conquer algorithms with this behavior can use almost linear time without sacrificing the overall linearity of the algorithm. Third, there is the challenge of explicitly solving a hard recurrence relation.

The published literature of exact solutions to (1) is scant—only the case $f(x)=x$, $M(1)=0$ has come under scrutiny. This case arises in the algorithmic problems outlined above, and also in at least one combinatorial setting [11]. The solution turns out to be

$$M(n)=\sum_{i=1}^{n-1}\nu(i)$$

where $\nu(i)$ is the number of 1-bits in the binary representation of i. This $M(n)$ has been well studied; see [14] for a detailed discussion and [4] for an extensive analysis of its asymptotic behavior.

Our analysis is aided considerably by the work of [2] on subadditive inequalities. In this paper (and in [1]), the determination of the value of k giving the maximum in (1) is given for some general classes of f. In particular, [2] considers the cases of f decreasing, f increasing and concave, and f increasing and convex. These cases, while of obvious interest, do not help in determining the exact behavior of $M(n)$ when f is a step function. Of course, step functions such as $\lfloor \lg x \rfloor$, $\lceil \lg (x+1) \rceil$, or $\lfloor x/2 \rfloor$ occur often as the cost of the merge step in divide-and-conquer algorithms, but the results of [2] do not apply to such functions. We will show how to handle many such cases.

To begin, we note that [2] observed that when f is nonincreasing, a simple induction on n verifies that $M(n)=nM(1)+(n-1)f(1)$. In the next section we treat the case in which f is nondecreasing and satisfies a certain concavity condition that is weak enough to encompass many step functions of interest. Similarly, in § 3 of this paper, we cover the case in which f is nondecreasing and satisfies a certain weak convexity condition that is less restrictive than general convexity. In these two sections, the main theorems are reformulated versions of results in [2], rewritten with emphasis on the less restrictive concavity/convexity condition. The corollaries show how these results apply to various step functions of interest. Section 4 develops some general bounds for arbitrary nondecreasing f. Finally, we summarize these results and give some unresolved questions in § 5.

2. The concave case.

DEFINITION. A real-valued function $f(n)$ is *concave* if $\Delta^2 f(n)\leqq 0$ for all $n\geqq 0$, that is, if

$$f(n+2)-f(n+1)\leqq f(n+1)-f(n) \quad \text{for all } n\geqq 0.$$

THEOREM 1. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and satisfies the inequality*

$$f(2^{m-1}+j)-f(2^{m-1})\leqq f(i)-f(i-j), \qquad 1\leqq j\leqq i/2\leqq 2^{m-1}. \tag{2}$$

Then $M(n)$ satisfies

$$M(2^m+i)=M(2^m)+M(i)+f(i), \qquad 1\leqq i\leqq 2^m, \quad m\geqq 0. \tag{3}$$

Proof. The proof is by induction on $n=2^m+i$. First, observe that since f is nondecreasing, the original recurrence simplifies to

$$M(n)=\max_{1\leqq k\leqq \lfloor n/2\rfloor}(M(k)+M(n-k)+f(k)). \tag{4}$$

Direct computation gives

$$M(2) = M(1) + M(1) + f(1), \qquad M(3) = M(1) + M(2) + f(1),$$
$$\begin{aligned} M(4) &= \max\,(M(1) + M(3) + f(1),\, M(2) + M(2) + f(2)) \\ &= \max\,(4M(1) + 3f(1),\, 4M(1) + 2f(1) + f(2)) \\ &= 4M(1) + 2f(1) + \max\,(f(1), f(2)) \\ &= 4M(1) + 2f(1) + f(2) \\ &= M(2) + M(2) + f(2). \end{aligned}$$

This establishes the theorem for $n \leqq 4$.

Suppose, by induction, that the theorem is true for values less than n; we will show that (3) holds for $n = 2^m + i$ as well. In particular, we will show that for $m \geqq 2$, $1 \leqq i \leqq 2^m$, (that is, $n \geqq 5$),

$$M(2^m) + M(i) + f(i) \geqq M(k) + M(2^m + i - k) + f(k), \qquad 1 \leqq k \leqq 2^{m-1} + i/2.$$

Since equality occurs at $k = i$, it follows that

$$\begin{aligned} M(2^m) + M(i) + f(i) &= \max_{1 \leqq k \leqq \lfloor (2^m + i)/2 \rfloor} (M(k) + M(2^m + i - k) + f(k)) \\ &= M(2^m + i). \end{aligned}$$

The range $1 \leqq k \leqq 2^{m-1} + i/2$ is broken into four subranges that are handled in separate cases.

Case 1. $1 \leqq k < i$. This is the simplest case; we have

$$\begin{aligned} M(k) + M(2^m + i - k) + f(k) &= M(k) + (M(2^m) + M(i - k) + f(i - k)) + f(k) \\ &\qquad \text{(by induction)} \\ &= M(2^m) + (M(k) + M(i - k) + \min\,(f(k), f(i - k))) \\ &\quad + \max\,(f(k), f(i - k)) \\ &\leqq M(2^m) + M(i) + \max\,(f(k), f(i - k)) \\ &\qquad \text{(by the definition of } M) \\ &\leqq M(2^m) + M(i) + f(i), \end{aligned}$$

since f is nondecreasing.

Case 2. $i + 1 \leqq k < 2^{m-2} + i/2$.

$$\begin{aligned} M(k) + M(2^m + i - k) + f(k) &= M(k) + M(2^{m-1} + 2^{m-1} + i - k) + f(k) \\ &= M(k) + (M(2^{m-1}) + M(2^{m-1} + i - k) \\ &\quad + f(2^{m-1} + i - k)) + f(k) \qquad \text{(by induction)} \\ &= M(2^{m-1}) + (M(k) + M(2^{m-1} + i - k) + f(k)) \\ &\quad + f(2^{m-1} + i - k) \\ &\leqq M(2^{m-1}) + M(2^{m-1} + i) + f(2^{m-1} + i - k) \\ &\qquad \text{(by the definition of } M) \\ &= M(2^{m-1}) + (M(2^{m-1}) + M(i) + f(i)) + f(2^{m-1} + i - k) \\ &\qquad \text{(by induction)} \\ &\leqq (M(2^{m-1}) + M(2^{m-1}) + f(2^{m-1})) + M(i) + f(i), \end{aligned}$$

because $i < k$ and f is nondecreasing.

$$= M(2^m) + M(i) + f(i) \quad \text{(by induction).}$$

Case 3. $2^{m-2} + i/2 \leqq k \leqq 2^{m-1}$.

$$\begin{aligned} M(k) + M(2^m + i - k) + f(k) &= M(k) + M(2^{m-1} + 2^{m-1} + i - k) + f(k) \\ &= M(k) + (M(2^{m-1}) + M(2^{m-1} + i - k) + f(2^{m-1} + i - k)) \\ &\quad + f(k), \end{aligned}$$

by induction, since $i < k$.

$$\begin{aligned} &= M(2^{m-1}) + (M(k) + M(2^{m-1} + i - k) + f(2^{m-1} + i - k)) \\ &\quad + f(k) \\ &\leqq M(2^{m-1}) + M(2^{m-1} + i) + f(k), \end{aligned}$$

by the definition of M, since $k \geqq 2^{m-1} + i - k$.

$$\begin{aligned} &= M(2^{m-1}) + (M(2^{m-1}) + M(i) + f(i)) + f(k) \\ &\qquad\qquad \text{(by induction)} \\ &= (M(2^{m-1}) + M(2^{m-1})) + M(i) + f(i) + f(k) \\ &= M(2^m) - f(2^{m-1}) + M(i) + f(i) + f(k) \quad \text{(by induction)} \\ &= M(2^m) + M(i) + f(i) + (f(k) - f(2^{m-1})) \\ &\leqq M(2^m) + M(i) + f(i), \end{aligned}$$

since $k \leqq 2^{m-1}$ and f is nondecreasing.

Case 4. $2^{m-1} < k \leqq 2^{m-1} + i/2$. Let $k = 2^{m-1} + j$, $1 \leqq j \leqq i/2 \leqq 2^{m-1}$; if $i - j > 2^{m-1}$ then $i - (k - 2^{m-1}) > 2^{m-1}$, so that $i > k$ and we have Case 1. Hence we may assume that $i - j \leqq 2^{m-1}$ and we have

$$\begin{aligned} M(k) + M(2^m + i - k) + f(k) &= M(2^{m-1} + j) + M(2^{m-1} + i - j) + f(2^{m-1} + j) \\ &= (M(2^{m-1}) + M(j) + f(j)) + (M(2^{m-1}) + M(i - j) \\ &\quad + f(i - j)) + f(2^{m-1} + j), \end{aligned}$$

by induction, since $j \leqq 2^{m-1}$ and $i - j \leqq 2^{m-1}$.

$$\begin{aligned} &= (M(2^{m-1}) + M(2^{m-1})) + (M(j) + M(i - j) + f(j)) \\ &\quad + f(i - j) + f(2^{m-1} + j) \\ &\leqq (M(2^m) - f(2^{m-1})) + M(i) + f(i - j) + f(2^{m-1} + j), \end{aligned}$$

by induction and the definition of M.

$$\begin{aligned} &= M(2^m) + M(i) + f(i) + ((f(2^{m-1} + j) - f(2^{m-1})) \\ &\quad - (f(i) - f(i - j))) \\ &\leqq M(2^m) + M(i) + f(i), \end{aligned}$$

because for $1 \leqq j \leqq i/2 \leqq 2^{m-1}$ the bracketed term is negative by hypothesis. □

A detailed examination of [2] reveals that Batty and Rogers actually proved Theorem 1. However, all they state is the following corollary.

COROLLARY 1 [2]. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and concave. Then $M(n)$ satisfies*

$$M(2^m+i)=M(2^m)+M(i)+f(i), \qquad 1\leqq i\leqq 2^m, \quad m\geqq 0.$$

Proof. The concavity of f guarantees that it satisfies the inequality (2) in the hypothesis of Theorem 1. □

We can use Theorem 1 to obtain an explicit solution to recurrence (1) for nondecreasing functions f that satisfy inequality (2). First, we get the solution for powers of two, then we extend it to arbitrary n.

COROLLARY 2. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and satisfies inequality* (2). *Then*

$$M(2^k)=2^kM(1)+2^{k-1}\sum_{i=0}^{k-1}\frac{f(2^i)}{2^i}.$$

Proof. The proof is by repeated application of (3). □

COROLLARY 3. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and satisfies inequality* (2). *Let* $n=2^{k_1}+2^{k_2}+\cdots+2^{k_l}$, $0\leqq k_1<k_2<\cdots<k_l$, $l=\nu(n)\geqq 1$; *then*

$$M(n)=nM(1)+\sum_{j=1}^{l}2^{k_j-1}\sum_{i=0}^{k_j-1}\frac{f(2^i)}{2^i}+\sum_{j=1}^{l-1}f\left(\sum_{i=1}^{j}2^{k_i}\right) \tag{5}$$

$$=nM(1)+\sum_{j=1}^{\infty}\left\lfloor\frac{n}{2^j}\right\rfloor f(2^{j-1})+\sum_{j=1}^{l-1}f\left(\sum_{i=1}^{j}2^{k_i}\right). \tag{6}$$

Proof. Equation (5) follows from repeated application of (3), together with Corollary 2. Equation (6) follows from (5) by observing that the coefficient of $f(2^{t-1})$ in (5) is $\sum_{1\leqq j\leqq l,\,k_j\geqq t}2^{k_j-t}=\lfloor n/2^t\rfloor$. □

Corollary 3 can be applied to a variety of step functions f, including $f(x)=\lceil\lg(x+1)\rceil$, $f(x)=\lfloor x/2\rfloor$, and so on. The evaluations of $M(n)$ in such cases are straightforward algebraic calculations. We give one example of such a calculation in the next corollary. We chose to give this example, $f(x)=\lfloor\lg n\rfloor$, because this particular choice of f is what initiated our investigation [16].

COROLLARY 4. *Let $M(n)$ be defined by the recurrence* (1) *with $f(x)=\lfloor\lg x\rfloor$. Then* $M(n)=(M(1)+1)n-\lfloor\lg n\rfloor-\nu(n)$.

Proof. Since $\lfloor\lg x\rfloor$ is nondecreasing and satisfies (2), Theorem 1 and its corollaries apply. Equation (5) in Corollary 3 tells us that

$$\begin{aligned}M(n)&=M(1)n+\sum_{j=1}^{l}2^{k_j-1}\sum_{i=0}^{k_j-1}\frac{i}{2^i}+\sum_{j=1}^{l-1}k_j\\&=M(1)n+\sum_{j=1}^{l}\sum_{i=0}^{k_j-1}i2^{k_j-1-i}+\sum_{j=1}^{l}k_j-\lfloor\lg n\rfloor,\end{aligned}$$

since $k_l=\lfloor\lg n\rfloor$. It remains to show that

$$\sum_{j=1}^{l}\sum_{i=0}^{k_j-1}i2^{k_j-1-i}+\sum_{j=1}^{l}k_j=n-\nu(n).$$

We have

$$\sum_{j=1}^{l}\sum_{i=0}^{k_j-1}i2^{k_j-1-i}+\sum_{j=1}^{l}k_j=\sum_{j=1}^{l}(2^{k_j}-k_j-1)+\sum_{j=1}^{l}k_j=\sum_{j=1}^{l}2^{k_j}-\sum_{j=1}^{l}1=n-\nu(n),$$

since $\sum_{i=0}^{t}i2^{t-i}=2^{t+1}-t-2$. □

The most famous instance of (1) is with $f(x)=x$ and $M(1)=0$ [4], [8], [11], [14]. We have Corollary 5.

COROLLARY 5 [8], [11], [14]. *Let $M(n)$ be defined by the recurrence* (1), *with $f(x)=x$ and $M(1)=0$. Then $M(n)=\sum_{k=0}^{n-1}\nu(k)$.*

Proof. Since $f(x)=x$ satisfies inequality (2), we apply Corollary 3 and find that

$$M(n)=\sum_{j=1}^{\infty}\left\lfloor\frac{n}{2^j}\right\rfloor 2^{j-1}+\sum_{j=1}^{l-1}\sum_{i=1}^{j}2^{k_i}$$

from equation (6). Now, imagine the binary representations of $0, 1, 2, \cdots, n-1$ written in an array, one above the other with the columns aligned. The number of 1-bits can be counted row by row to give $\sum_{k=0}^{n-1}\nu(k)$. Alternatively, the number of 1-bits can be counted column by column as follows. In the jth column from the right there will be $\lfloor n/2^j\rfloor$ complete groups of 2^{j-1} 1-bits, for a total of $\lfloor n/2^j\rfloor 2^{j-1}$ 1-bits in those complete groups; summing this value from one to infinity counts the 1-bits in complete groups, column by column. Now we count the fragmentary groups of 1-bits column by column as follows. If the jth column of n is a 1-bit (that is, if 2^{j-1} appears in the binary representation of n), there will be a fragmentary group of 1-bits in column j. The fragmentary group in column j will have a number of 1-bits equal to the value of the binary number formed by the bits of n to the right of the jth column, that is, $\sum_{i=1}^{j}2^{k_i}$ 1-bits. Summing this for all the 1-bits in n (except the first which can have no fragmentary group), we obtain precisely $\sum_{j=1}^{l-1}\sum_{i=1}^{j}2^{k_i}$. □

3. The convex case. The case of nondecreasing, convex f is not nearly as interesting as the concave case. As in the concave case, we state and prove a result that holds for a relatively weak form of convexity, and then state as a corollary the result for usual convexity. Also as in the concave case, Batty and Rogers [2] proved a more general result than they state.

DEFINITION. A real-valued function $f(n)$ is *convex* if $\Delta^2 f(n)\geqq 0$ for all $n\geqq 0$, that is, if

$$f(n+2)-f(n+1)\geqq f(n+1)-f(n) \quad \text{for all } n\geqq 0.$$

THEOREM 2. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and satisfies*

$$f(\lfloor(p+q)/2\rfloor)-f(\lfloor q/2\rfloor)\geqq f(p)-f(\lceil p/2\rceil) \tag{7}$$

for all integers p and q, $0\leqq p\leqq q$. Then $M(n)$ satisfies

$$M(n)=M(\lfloor n/2\rfloor)+M(\lceil n/2\rceil)+f(\lfloor n/2\rfloor). \tag{8}$$

Proof. The proof is by induction on n. As in the proof of Theorem 1, the original recurrence simplifies to

$$M(n)=\max_{1\leqq k\leqq\lfloor n/2\rfloor}(M(k)+M(n-k)+f(k))$$

and direct computation gives

$$M(2)=M(1)+M(1)+f(1),$$

so the theorem holds for $n\leqq 2$. Suppose it holds for all values less than n; we will show that (8) holds for n as well. In particular, we will show that for all p, q, $0<p\leqq q$, $p+q=n$,

$$M(\lfloor n/2\rfloor)+M(\lceil n/2\rceil)+f(\lfloor n/2\rfloor)\geqq M(p)+M(q)+f(p).$$

Since equality occurs for $p = \lfloor n/2 \rfloor$, $q = \lceil n/2 \rceil$, it follows that

$$M(\lfloor n/2 \rfloor) + M(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) = \max_{1 \leqq k \leqq \lfloor n/2 \rfloor} (M(k) + M(n-k) + f(k)).$$

If p is odd and q is even, we have $\lfloor p/2 \rfloor + \lceil q/2 \rceil = \lfloor (p+q)/2 \rfloor$ and $\lceil p/2 \rceil + \lfloor q/2 \rfloor = \lceil (p+q)/2 \rceil$. Since $p+q=n$ we have

$$\begin{aligned}
M(p)+M(q)+f(p) &= M(\lfloor p/2 \rfloor)+M(\lceil p/2 \rceil)+f(\lfloor p/2 \rfloor)+M(\lfloor q/2 \rfloor)+M(\lceil q/2 \rceil) \\
&\quad +f(\lfloor q/2 \rfloor)+f(p) \quad \text{(by induction)} \\
&= (M(\lfloor p/2 \rfloor)+M(\lceil q/2 \rceil)+f(\lfloor p/2 \rfloor))+(M(\lceil p/2 \rceil) \\
&\quad +M(\lfloor q/2 \rfloor)+f(\lceil p/2 \rceil))+f(\lfloor q/2 \rfloor)-f(\lceil p/2 \rceil)+f(p) \\
&\leqq M(\lfloor n/2 \rfloor)+M(\lceil n/2 \rceil)+f(\lfloor q/2 \rfloor) \\
&\quad -f(\lceil p/2 \rceil)+f(p) \quad \text{(by the definition of } M) \\
&= M(\lfloor n/2 \rfloor)+M(\lceil n/2 \rceil)+f(\lfloor n/2 \rfloor) \\
&\quad -((f(\lfloor (p+q)/2 \rfloor)-f(\lfloor q/2 \rfloor))-(f(p)-f(\lceil p/2 \rceil))) \\
&\leqq M(\lfloor n/2 \rfloor)+M(\lceil n/2 \rceil)+f(\lfloor n/2 \rfloor),
\end{aligned}$$

because f satisfies inequality (7). On the other hand, if either p is even and q is odd, or if both p and q have the same parity, we have $\lfloor p/2 \rfloor + \lceil q/2 \rceil = \lceil (p+q)/2 \rceil$ and $\lceil p/2 \rceil + \lfloor q/2 \rfloor = \lfloor (p+q)/2 \rfloor$ and the proof continues exactly as above. □

COROLLARY 6 [2]. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and convex. Then $M(n)$ satisfies*

$$M(n) = M(\lfloor n/2 \rfloor) + M(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor).$$

Proof. Since

$$\lfloor (p+q)/2 \rfloor - \lfloor q/2 \rfloor \geqq p - \lceil p/2 \rceil$$

for all nonnegative integers p, q, it follows that inequality (7) holds by the convexity of f. □

COROLLARY 7. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing and satisfies inequality* (7); *we have*

$$M(2^k) = 2^k M(1) + 2^{k-1} \sum_{i=0}^{k-1} \frac{f(2^i)}{2^i}.$$

Proof. The proof is by repeated application of equation (8). □

COROLLARY 8 [8], [14]. *Let $M(n)$ be defined by the recurrence* (1), *with $f(x) = x$; then $M(n) = M(\lfloor n/2 \rfloor) + M(\lceil n/2 \rceil) + \lfloor n/2 \rfloor$.* □

4. The general case. Corollaries 2 and 7 suggest that the general asymptotic character of $M(n)$ does not depend on the particular shape of f, only on it being nondecreasing. In this section we verify this observation and derive bounds on $M(n)$ for arbitrary nondecreasing f.

In studying the function $M(n)$ as defined by recurrence (1) for arbitrary nondecreasing f, it will be convenient to use binary trees to represent the recursive evaluation of (1) for a given n. We define a *partition tree of n* as a rooted binary tree T containing $n-1$ internal nodes and n external nodes (see, for example, [16]) in which every node

x in T is labeled with a natural number: the root is labeled with n, every internal node is labeled with the sum of the labels of its two children, and every external node is labeled with one. Furthermore, the label of a left child must never be larger than the label of its sibling. When no confusion results, we will refer to a node by its label. We denote by $\mathcal{T}(n)$ the set of all partition trees of n.

We will want to consider partition trees in which parts of the tree have been pruned away. Given a particular partition tree T in $\mathcal{T}(n)$, we define the sequence of truncated partition trees $T_0 = T$, T_1, T_2, $\cdots$ as follows. T_t, $t \geqq 0$, is the tree that results when all nodes $x < 2^t$ in T are delected.

Finally, we define the function $F(T)$, for a truncated partition tree T, to be

$$F(T) = \sum_{\text{left children } x \text{ in } T} f(x).$$

The formation rule for partition trees and the nondecreasing property of f make the relationship between the recurrence (1) and partition trees

$$M(n) = nM(1) + \max_{T \text{ in } \mathcal{T}(n)} F(T).$$

We will, therefore, be able to bound $M(n)$ by bounding $F(T)$. Specifically, we will show below (in the proof of Theorem 3) that

$$F(T) \leqq \sum_{i=1}^{k-1} (\lfloor n/2^i \rfloor + b_{i-1}) f(2^i) \quad \text{for all } T \text{ in } \mathcal{T}(n)$$

where $k-1 = \lfloor \lg n \rfloor$ and $n = (\cdots b_3 b_2 b_1 b_0)_2$. First, however, we need a preliminary result.

Given a partition tree T, let $L(T)$ denote the number of left children in T. Then, we define

$$L_t(n) = \max_{T \text{ in } \mathcal{T}(n)} L(T_t);$$

that is, $L_t(n)$ denotes the largest number of left children remaining in any partition tree of n that has been pruned of all nodes with labels less than 2^t. Obviously, $L_t(1) = 0$ and $L_t(n)$ can be defined recursively by

$$L_t(n) = \max_{1 \leqq k \leqq \lfloor n/2 \rfloor} (L_t(k) + L_t(n-k) + g(k)) \tag{9}$$

where

$$g(x) = \begin{cases} 0 & \text{if } x < 2^t, \\ 1 & \text{otherwise;} \end{cases}$$

the $g(k)$ term in the recurrence counts the root of the left subtree.

LEMMA.

$$L_t(n) = \begin{cases} \lfloor n/2^t \rfloor - 1 & \text{if } n \geqq 2^t, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Recurrence (9) is an instance of recurrence (4) and the function g is nondecreasing and satisfies inequality (2), so that Theorem 1 and its corollaries apply.

Corollary 3 tells us that

$$L_t(n) = nL_t(1) + \sum_{\substack{1 \le j \le l \\ k_j > t}} 2^{k_j - 1} \sum_{i=t}^{k_j - 1} 2^{-i} + \sum_{j=1}^{l-1} \begin{cases} 1 & \text{if } k_j \ge t, \\ 0 & \text{otherwise} \end{cases}$$

$$= \sum_{\substack{1 \le j \le l \\ k_j > t}} (2^{k_j - t} - 1) + \begin{cases} \nu(\lfloor n/2^t \rfloor) - 1 & \text{if } n \ge 2^t, \\ 0 & \text{otherwise} \end{cases}$$

$$= \lfloor n/2^t \rfloor - \nu(\lfloor n/2^t \rfloor) + \begin{cases} \nu(\lfloor n/2^t \rfloor) - 1 & \text{if } n \ge 2^t, \\ 0 & \text{otherwise} \end{cases}$$

$$= \begin{cases} \lfloor n/2^t \rfloor - 1 & \text{if } n \ge 2^t, \\ 0 & \text{otherwise.} \end{cases}$$

□

THEOREM 3. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing. Then $M(n)$ satisfies*

$$M(n) \le nM(1) + \sum_{i=0}^{\lfloor \lg n \rfloor} (\lfloor n/2^i \rfloor + b_{i-1}) f(2^i)$$

where $n = (\cdots b_3 b_2 b_1 b_0)_2$.

Proof. Let $2^{k-1} \le n < 2^k$ so that $k - 1 = \lfloor \lg n \rfloor$. The theorem will follow from the lemma and the inequality

$$F(T) \le \sum_{i=1}^{k-1} [L_{k-(i+1)}(n) - L_{k-i}(n)] f(2^{k-i}) \quad \text{for all } T \text{ in } \mathcal{T}(n),$$

because

$$M(n) = nM(1) + \max_{T \text{ in } \mathcal{T}(n)} F(T).$$

In fact, for any given partition tree T in $\mathcal{T}(n)$ we will establish, by induction on t, the more general inequality

$$F(T_{k-t}) \le \left[\sum_{i=1}^{t-1} (L_{k-(i+1)}(n) - L_{k-i}(n)) f(2^{k-i}) \right] - (L_{k-t}(n) - L(T_{k-t})) f(2^{k-t+1}), \tag{10}$$

for $1 \le t \le k$.

We need to establish the basis. Because $k - 1 = \lfloor \lg n \rfloor$, there is no left child $x \ge 2^{k-1}$ in any T in $\mathcal{T}(n)$; thus $F(T_{k-1}) = 0$, for all T in $\mathcal{T}(n)$. Similarly, $L_{k-1}(n) = L(T_{k-1}) = 0$, so (10) holds for $t = 1$.

Suppose (10) holds and $t < k$; we will show that it holds for $t + 1$ as well. We have

$$F(T_{k-(t+1)}) = F(T_{k-t}) + \sum_{\substack{2^{k-(t+1)} \le x < 2^{k-t} \\ x \text{ is a left child in } T}} f(x)$$

$$\le F(T_{k-t}) + \sum_{\substack{2^{k-(t+1)} \le x < 2^{k-t} \\ x \text{ is a left child in } T}} f(2^{k-t})$$

because f is nondecreasing.

$$= F(T_{k-t}) + \begin{pmatrix} \text{the number of left children} \\ x \text{ in } T \text{ such that } 2^{k-(t+1)} \le x < 2^{k-t} \end{pmatrix} f(2^{k-t})$$

$$= F(T_{k-t}) + (L(T_{k-(t+1)}) - L(T_{k-t})) f(2^{k-t}).$$

Using (10), we have

$$\leqq \left[\sum_{i=1}^{t-1} (L_{k-(i+1)}(n) - L_{k-i}(n)) f(2^{k-i})\right] - (L_{k-t}(n) - L(T_{k-t})) f(2^{k-t+1})$$
$$+ (L(T_{k-(t+1)}) - L(T_{k-t})) f(2^{k-t})$$
$$\leqq \left[\sum_{i=1}^{t-1} (L_{k-(i+1)}(n) - L_{k-i}(n)) f(2^{k-i})\right] - (L_{k-t}(n) - L(T_{k-t})) f(2^{k-t})$$
$$+ (L(T_{k-(t+1)}) - L(T_{k-t})) f(2^{k-t}),$$

since f is nondecreasing.

$$= \left[\sum_{i=1}^{t-1} (L_{k-(i+1)}(n) - L_{k-i}(n)) f(2^{k-i})\right] - (L_{k-t}(n) - L(T_{k-(t+1)})) f(2^{k-t})$$
$$= \left[\sum_{i=1}^{t} (L_{k-(i+1)}(n) - L_{k-i}(n)) f(2^{k-i})\right] - (L_{k-(t+1)}(n) - L(T_{k-(t+1)})) f(2^{k-t}),$$

completing the induction.

Now, $T = T_0$, so

$$M(n) = nM(1) + \max_{T \text{ in } \mathcal{T}(n)} F(T_0),$$

and then

$$M(n) \leqq nM(1) + \sum_{i=1}^{k-1} (L_{k-(i+1)}(n) - L_{k-i}(n)) f(2^{k-i}), \tag{11}$$

by (10) with $t = k$ since $L_0(n) = L(T_0) = n - 1$.

The lemma tells us that

$$L_{k-(i+1)}(n) - L_{k-i}(n) = \begin{cases} \lfloor n/2^{k-(i+1)} \rfloor - 1 - (\lfloor n/2^{k-i} \rfloor - 1), & n \geqq 2^{k-i}, \\ \lfloor n/2^{k-(i+1)} \rfloor - 1 - 0, & 2^{k-(i+1)} \leqq n < 2^{k-i}, \\ 0 - 0 & n < 2^{k-(i+1)}, \end{cases}$$
$$= \begin{cases} \lfloor n/2^{k-(i+1)} \rfloor - \lfloor n/2^{k-i} \rfloor, & n \geqq 2^{k-i}, \\ \lfloor n/2^{k-(i+1)} \rfloor - 1, & 2^{k-(i+1)} \leqq n < 2^{k-i}, \\ 0, & n < 2^{k-(i+1)}, \end{cases}$$
$$= \begin{cases} \lfloor n/2^{k-i} \rfloor + b_{k-(i+1)}, & n \geqq 2^{k-i}, \\ 0, & \text{otherwise}, \end{cases}$$

because $\lfloor n/2^m \rfloor = 2\lfloor n/2^{m+1} \rfloor + b_m$. Inequality (11) thus becomes

$$M(n) \leqq nM(1) + \sum_{i=1}^{k-1} (\lfloor n/2^{k-i} \rfloor + b_{k-(i+1)}) f(2^{k-i})$$
$$= nM(1) + \sum_{i=1}^{k-1} (\lfloor n/2^{i} \rfloor + b_{i-1}) f(2^{i}),$$

as desired. □

COROLLARY 9. *Let $M(n)$ be defined by the recurrence* (1), *where $M(1)$ is given and f is nondecreasing. Then $M(n)$ satisfies*

$$(12)\quad nM(1)+\sum_{j=1}^{\lceil \lg n\rceil}\lfloor n/2^j\rfloor f(2^{j-1})+\sum_{j=1}^{l-1} f\left(\sum_{i=1}^{j}2^{k_i}\right)\leqq$$

$$M(n)\leqq nM(1)+\sum_{j=1}^{\lfloor \lg n\rfloor}\lfloor n/2^j\rfloor f(2^j)+\sum_{j=1}^{l-1} f(2^{k_j+1}),$$

where $n=2^{k_1}+2^{k_2}+\cdots+2^{k_l}$, $0\leqq k_1<k_2<\cdots<k_l$, and $l=\nu(n)\geqq 1$.

Proof. The upper bound follows directly from Theorem 3. The lower bound follows from equation (6) in Corollary 3, interpreted as the evaluation of $F(T(n))$, where $T(n)$ in $\mathscr{T}(n)$ is defined recursively as follows: $T(1)$ is a single leaf labeled one. If $n=2^m+i$, $1\leqq i\leqq 2^m$, then $T(n)$ has the label n at the root, $T(i)$ as its left subtree, and $T(2^m)$ as its right subtree. □

5. Conclusions and open problems. The most striking conclusion we can make is that divide-and-conquer algorithms whose time requirements are given by (1) can use far more than logarithmic time for their merge step, without losing overall linear-time behavior: Inequality (12) guarantees that $M(n)=O(n)$, even for $f(x)$ as large as $O(x/(\log x)^{1+\varepsilon})$, $\varepsilon>0$, that is, even when f is almost linear! This observation should make possible the use of more sophisticated merge steps in the divide-and-conquer algorithms.

Table 1 gives some examples of the comparative growth rates of $M(n)$ versus f, based on inequality (12). All of the entries in Table 1 are straightforward to verify, except those for

$$f(x)=\Theta\left(x\Big/\prod_{1\leqq i<\log_b^* n}\log_b^{(i)}x\right);$$

TABLE 1

Relative growth rates of f versus $M(n)$, based on inequality (12). *For any particular base, $\log^{(k)}x$ is the kth iterated logarithm defined by $\log^{(0)}x=x$, $\log^{(k+1)}x=\log(\log^{(k)}x)$ and $\log^* x$ is the least k such that $\log^{(k)}\leqq 1$. Unless otherwise indicated, the base of the logarithms is arbitrary.*

$f(x)$	$M(n)$
$O(x/(\log x)^{1+\varepsilon})$, $\varepsilon>0$	$\Theta(n)$
$\Theta\left(x\Big/\prod_{1\leqq i<\log_b^* x}\log_b^{(i)}x\right)$, $b<e$	$\Theta(n)$
$\Theta\left(x\Big/\prod_{1\leqq i<\ln^* x}\ln^{(i)}x\right)$	$\Theta(n\ln^* n)$
$\Theta\left(x\Big/\prod_{1\leqq i<\log_b^* x}\log_b^{(i)}x\right)$, $b>e$	$\Theta(n(\ln b)^{\log_b^* n})$
$\Theta\left(x\Big/\prod_{1\leqq i\leqq k}\log^{(i)}x\right)$, $k\geqq 0$	$\Theta(n\log^{(k+1)}n)$
$\Theta(x/\log\log x)$	$\Theta((n\log n)/\log\log n)$
$\Theta(x(\log x)^k)$, $k\geqq 0$	$\Theta(n(\log n)^{k+1})$
$\Theta(x^{1+\varepsilon}(\log x)^k)$, $\varepsilon>0$, $k\geqq 0$	$\Theta(n^{1+\varepsilon}(\log n)^k)$

those interesting entries follow from the proof of Lemma B.1 in [13] (see also [19]). We may also ask for what function f will $M(n) = n\alpha(n)$, where $\alpha(n)$ is a functional inverse of Ackermann's function (see [20]); the answer, which is fairly complex, can be found in [17].

The specific case of $f(x) = \lceil \lg x \rceil$ is very strange. Inequality (2) in Theorem 1 fails (with, for example, $m = i = 4$ and $j = 1$), as does inequality (7) in Theorem 2 (with $p = 31$ and $q = 34$, for instance). However, the *conclusions* of both theorems hold! That is, the solution to recurrence (1) in this case is $M(n) = (M(1)+1)n - \lceil \lg n \rceil - 1$ which satisfies both equations (3) and (8) as is easily shown by induction on n. This anomaly suggests that both Theorems 1 and 2 can be proved under weaker assumptions than we have used. The seemingly insignificant change of f from $f(x) = \lceil \lg x \rceil$ to $f(x) = \lceil \lg (x+1) \rceil$, which does satisfy inequality (2), yields the solution $M(n) = (M(1)+2)n - \lfloor \lg n \rfloor - \nu(n) - 1$. When $M(1) = 0$, this is roughly twice as large as when $f(x) = \lceil \lg x \rceil$, demonstrating a somewhat unstable dependence of $M(n)$ on f.

The bound in Theorem 3, and hence the upper bound in Corollary 9, is fairly tight when f is a constant, but becomes progressively looser as the rate of growth of f increases. For $f(x) = \Theta(x^w)$, $w \geqq 0$, for example, the upper bound is approximately 2^w times the lower bound. We believe that the lower bound is more nearly correct, but an improved upper bound has eluded us.

We should mention that the solution to the minimax form of the recurrence (1), namely,

$$M(n) = \min_{1 \leqq k < n} (M(k) + M(n-k) + \max(f(k), f(n-k))),$$

is precisely parallel to the maximin form that we have considered. In the minimax form, the roles of nondecreasing and nonincreasing are interchanged, as are the roles of concavity and convexity. It is the minimax form that was studied in [2].

Finally, there are two natural generalizations of recurrence (1) that can also be studied:

$$M(n) = \max_{k_1 + k_2 + \cdots + k_t = n} \left(\sum_{i=1}^{t} M(k_i) + \min_{1 \leqq i \leqq t} f(k_i) \right),$$

and the identical definition with "min" replaced by "sum-of-all-but-the-max." Little is known about either generalized form. The generalization with "min" has been considered in [1], while the generalization with "sum-of-all-but-the-max" occurs in [22] and [23].

Acknowledgments. We thank Herbert Edelsbrunner for asking us about the behavior of the recurrence for f other than $\Theta(\log n)$, for pointing out references [3], [5], [7], [9], [12], and [23], and for his helpful comments. We thank Ronald Rivest for pointing out reference [19].

REFERENCES

[1] C. J. K. BATTY, M. J. PELLING, AND D. G. ROGERS, *Some recurrence relations of recursive minimization*, SIAM J. Algebraic Discrete Methods, 3 (1982), pp. 13-29.

[2] C. J. K. BATTY AND D. G. ROGERS, *Some maximal solutions of the generalized subadditive inequality*, SIAM J. Algebraic Discrete Methods, 3 (1982), pp. 369-378.

[3] B. CHAZELLE AND H. EDELSBRUNNER, *An optimal algorithm for intersecting line segments in the plane*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, Washington, DC, 1988, pp. 590-600.

[4] H. DELANGE, *Sur la fonction sommatoire de la fonction "somme des chiffres"*, L'Enseignement Math., 21 (1975), pp. 31-47.

[5] D. DOBKIN, L. GUIBAS, J. HERSHBERGER, AND J. SNOEYINK, *An efficient algorithm for finding the CSG representation of a simple polygon*, Computer Graphics, 22 (1988), pp. 31-40.

[6] M. L. FREDMAN, *Two applications of a probabilistic search technique: sorting $X + Y$ and building balanced search trees*, in Proc. 7th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1975, pp. 240-244.

[7] 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, IEEE Computer Society Press, Washington, DC, 1987, pp. 11-19.

[8] D. H. GREENE AND D. E. KNUTH, *Mathematics for the Analysis of Algorithms*, Second edition, Birkhäuser, Boston, 1982.

[9] 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.

[10] D. HAREL AND R. E. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338-355.

[11] S. HART, *A note on the edges of the n-cube*, Discrete Math., 14 (1976), pp. 157-163.

[12] K. HOFFMANN, K. MEHLHORN, P. ROSENSTIEHL, AND R. E. TARJAN, *Sorting Jordan sequences in linear time*, in Proc. Symposium on Computational Geometry, Association for Computing Machinery, New York 1985, pp. 196-202.

[13] 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.

[14] M. D. MCILROY, *The number of 1's in binary integers: Bounds and extremal properties*, SIAM J. Comput., 3 (1974), pp. 255-261.

[15] K. MEHLHORN, *Data Structures and Algorithms* 1: *Sorting and Searching*, Springer-Verlag, Berlin, New York, 1984.

[16] E. M. REINGOLD AND W. J. HANSEN, *Data Structures in Pascal*, Little, Brown, Boston, 1986.

[17] E. M. REINGOLD AND X.-J. SHEN, *More nearly optimal algorithms for unbounded searching*, Report UIUCDCS-R-88-1471, Department of Computer Science, University of Illinois, Urbana, IL, 1988.

[18] E. M. REINGOLD AND J. S. TILFORD, *Tidier drawings of trees*, IEEE Trans. Software Engrg., 7 (1981), pp. 223-228.

[19] J. RISSANEN, *A universal prior for integers and estimation by minimum description length*, Ann. Statist., 11 (1983), pp. 416-431.

[20] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., 22 (1975), pp. 215-225.

[21] R. E. TARJAN AND C. J. 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.

[22] J. S. TILFORD, *Tree drawing algorithms*, Report UIUCDCS-R-81-1055, Department of Computer Science, University of Illinois, Urbana, IL, 1981.

[23] P. VAIDYA, *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem*, Disc. Comput. Geometry, 4 (1989), pp. 101-115.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1201-1225, December 1989

009

GEOMETRY HELPS IN MATCHING*

PRAVIN M. VAIDYA†

Abstract. A set of $2n$ points on the plane induces a complete weighted undirected graph as follows. The points are the vertices of the graph, and the weight of an edge between any two points is the distance between the points under some metric. The problem of finding a minimum weight complete matching (MWCM) in such a graph is studied. An $O(n^{2.5}(\log n)^4)$ algorithm is given for finding an MWCM in such a graph, for the L_1 (*manhattan*), the L_2 (*Euclidean*), and the L_∞ metrics. The bipartite version of the problem is also studied, where half the points are painted with one color and the other half with another color, and the restriction is that a point of one color may be matched only to a point of another color. An $O(n^{2.5} \log n)$ algorithm for the bipartite version, for the L_1, L_2, and L_∞ metrics, is presented. The running time for the bipartite version can be further improved to $O(n^2(\log n)^3)$ for the L_1 and L_∞ metrics.

Key words. weighted matching, computational geometry, optimization

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U05

1. Introduction.

Given a complete weighted undirected graph on a set of $2n$ vertices, a complete matching is a set of n edges such that each vertex has exactly one edge incident on it. The weight of a set of edges is the sum of the weights of the edges in the set, and a minimum weight complete matching (MWCM) is a complete matching that has the least weight among all the complete matchings.

We study the problem of finding an MWCM in the complete graph induced by a set of $2n$ points on the plane. The points are the vertices of the graph, and the weight of an edge between any two points is the distance between the points under some metric. We shall investigate two common metrics: the L_1 (*manhattan*) metric, and the L_2 (*Euclidean*) metric. (We note that the L_∞ metric can be converted to the L_1 metric by rotating the coordinate system by 45°, and so any algorithm for the L_1 metric can be trivially modified to work for the L_∞ metric.) The input consists of $2n$ points that specify the locations of the vertices on the plane. Each point p is given as an ordered pair (p_x, p_y), where p_x and p_y denote the x and y coordinates of p, respectively. The L_r distance between two points p and q is given by $(|p_x - q_x|^r + |p_y - q_y|^r)^{1/r}$. (Note that the L_1 distance between p and q is given by $|p_x - q_x| + |p_y - q_y|$.) We shall assume that the metric defining the edge weights is fixed.

We also study the bipartite version of the MWCM problem for points on the plane. In the bipartite version, half the points are painted with one color and the other half another color, and the restriction is that a point of one color can be matched only to a point of the other color.

The complete graph induced by a set of $2n$ points on the plane is entirely specified by the locations of the vertices. So the problem of finding an MWCM in such a graph differs from the problem of finding an MWCM in a general complete graph in that the size of the input is $O(n)$ rather than $\Omega(n^2)$. The input is sparse since the edge weights are implicitly defined by the underlying geometry. It is interesting to investigate if the geometric nature of the MWCM problem for points on the plane can be exploited

* Received by the editors November 30, 1987; accepted for publication (in revised form) December 5, 1988.

† Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Avenue, Urbana, Illinois 61801. Present address, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

to obtain an algorithm for its solution that is faster than the $\Theta(n^3)$ algorithm [6], [11] for general graphs. At this point we note that several heuristics for finding a good complete matching (not necessarily of minimum weight) on points on the plane have been developed [2], [9], [17], but the only known way to find an MWCM on $2n$ points on the plane is to run the MWCM algorithm for general graphs that requires $\Theta(n^3)$ time. In this paper we show that geometry does help to obtain a faster algorithm. We give an $O(n^{2.5}(\log n)^4)$ algorithm for finding an MWCM in the complete graph induced by a set of $2n$ points on the plane, for the L_1 and L_2 metrics. For the bipartite version of the MWCM problem for $2n$ points on the plane, we give an $O(n^{2.5} \log n)$ algorithm for the L_1 and L_2 metrics. For the bipartite case, the running time of the MWCM algorithm can be further improved to $O(n^2(\log n)^3)$ for the L_1 metric. The space requirement of all the algorithms is $O(n \log n)$ in the case of L_2 metric and $O(n(\log n)^2)$ in the case of L_1 metric.

The algorithms described in this paper will be essentially the well-studied primal-dual algorithms for weighted matching, namely, the Hungarian method [10], [11], [14] for bipartite matching, and Edmond's algorithm [4], [11], [14] for general matching. The primal-dual algorithms for weighted matching associate a dual variable with each vertex of the given graph, and the *slack* associated with an edge is the weight of the edge minus the sum of the dual variables associated with the end vertices of the edge. The algorithms can be substantially speeded up for points in the plane by the application of two key ideas. First, associating a weight with each vertex (point) that is suitably related to the dual variable corresponding to the vertex and that changes much less frequently than the dual variable, and implicity maintaining the dual variable using the weight. Second, reducing the computation of the minimum slack for certain subsets of edges to geometric query problems that involve the weights associated with the vertices and that can be efficiently solved using known data structures in computational geometry.

In § 2 we shall discuss some geometric query problems that arise naturally in the implementation of the primal-dual weighted matching algorithm for points on the plane, and see how known data structures in computational geometry can be used to solve them efficiently. In § 3 we shall give the algorithm for the bipartite version of the MWCM problem for points in the plane. The bipartite case is easier, and serves to illustrate the main ideas that are used in developing the algorithm for the general case. In § 4 we describe the algorithm for finding an MWCM in the complete graph induced by a set of points on the plane.

We shall assume a real RAM model of computation [15] standard in computational geometry, so arithmetic operations (i.e., addition, subtraction, multiplication, division), memory access operations, and comparison operations, on real numbers require constant time. (Actually, it is not necessary to assume that division requires constant time; the restricted assumption that division by two takes constant time is adequate.) For the case of the L_2 metric, we must make the additional assumption that either square roots can be computed in constant time (so that edge weights can be obtained in constant time) or that the edge weights have been precomputed and are available at the start of the algorithm.

Next, we introduce some notation and definitions.

A *priority queue* [1] is an abstract data structure consisting of a collection of elements, each element being associated with a real-valued *priority*. A priority queue supports the three operations—insert an element with some priority, delete an element, and find an element with the minimum priority—in time proportional to the logarithm of the number of elements in the priority queue.

We let $d(p, q)$ denote the distance between points p and q. (The metric under consideration will be clear from the context.) Let $H_x(a, b)$ denote the set of all points p on the plane such that $a \leqq p_x \leqq b$. Similarly, let $H_y(a, b)$ denote the set of all points p on the plane such that $a \leqq p_y \leqq b$. For a pair of real numbers a and b, $[a, b)$ denotes the set of all real numbers c such that $a \leqq c < b$. With respect to an ordered sequence $a_1 < a_2 < \cdots < a_m$, $[a_i, a_j)$ denotes the set of all a_k such that $i \leqq k < j$.

With respect to a set of points P such that there is a weight $w(p)$ associated with each point p in P, we define the following terms. For subsets P_1, P_2 of P, *shortest*$[P_1, P_2]$ denotes an edge (p_1^*, p_2^*), $p_1^* \in P_1$, $p_2^* \in P_2$, such that

$$d(p_1^*, p_2^*) - w(p_1^*) - w(p_2^*) = \min_{p_1 \in P_1,\, p_2 \in P_2} \{d(p_1, p_2) - w(p_1) - w(p_2)\}.$$

For a point q, *nearest*$[q, P]$ denotes a point $p^* \in P$ such that

$$d(q, p^*) - w(p^*) = \min_{p \in P} \{d(q, p) - w(p)\},$$

and *shortest*$[q, P]$ denotes the edge $(q,$ *nearest*$[q, P])$.

Finally, we shall use the term *vertex* when referring to a graph, and the term *node* when referring to a data structure.

2. Geometric query problems arising in matching on the plane. The primal-dual algorithms for weighted matching associate a dual variable with each vertex of the given graph, and the slack associated with an edge is the weight of the edge minus the sum of the dual variables associated with the end vertices of the edge. During the execution of the matching algorithm, we are repeatedly required to compute the minimum slack for certain subsets of edges. To perform this computation efficiently we shall need a good solution to the following query problems.

PROBLEM 1. Given a set of points P and a weight $w(p)$ for each point p in P, preprocess P so that for a given query point q, *nearest*$[q, [p_i, p_j)]$ can be found quickly.

PROBLEM 2. Given a set of points P, an ordering $p_1 < p_2 < \cdots < p_{|P|}$ of the points in P, and a weight $w(p)$ associated with each point p in P, preprocess P, so that given a query point q, and an interval $[p_i, p_j)$ such that $1 \leqq i < j \leqq |P| + 1$, *nearest*$[q, [p_i, p_j)]$ can be computed quickly.

In Problems 1 and 2 the set P is static. We shall also require a solution to the *semidynamic* version of Problems 1 and 2. In the semidynamic version, a new point can be added to P but a point can never be deleted from P. Furthermore, P is totally ordered by the following rule. For a pair of points $p, p' \in P$, $p < p'$ if and only if p was added to P before p'.

Problem 1 comes up in the bipartite case as well as the general case, and its solution enables us to quickly compute the minimum slack for various subsets of edges. Problem 2 and the semidynamic versions of the two problems arise because of certain subsets of vertices of odd cardinality, called blossoms, in Edmond's algorithm for general weighted matching. One type of blossom corresponds to intervals in some ordering on the set of vertices (points), and given a vertex q and a blossom B of this type, we are required to compute the minimum slack over all edges between q and vertices in B. This leads to Problem 2. The semidynamic versions arise because of blossoms merging to form bigger blossoms.

In §§ 2.1 and 2.2 we describe solutions to Problems 1 and 2, respectively. In § 2.3 we shall describe how to suitably modify the data structures for Problems 1 and 2 to handle the semidynamic case where new points may be inserted into P but no point

may be deleted from P. The solutions use segment trees [15], a data structure common in computational geometry. So we shall first briefly discuss segment trees. The segment tree for the interval $[i, j)$ is a rooted binary tree defined as follows:

(1) $j > i+1$. The interval $[i, j)$ is associated with the root of the segment tree. The left subtree at the root is the segment tree for $[i, \lceil (i+j)/2 \rceil)$ and the right subtree at the root is the segment tree for $[\lceil (i+j)/2 \rceil, j)$.

(2) $j = i+1$. With the root is associated the interval $[i, i+1)$, and the left and right subtrees are both empty.

(3) $j \leqq i$. The segment tree is empty.

The segment tree has $O(\log(|j-i|))$ levels, and the disjoint union of all the intervals at a specific level in the tree is the interval $[i, j)$. Any subinterval of $[i, j)$ with integer endpoints can be expressed as the disjoint union of $O(\log(|j-i|))$ intervals in the segment tree. The segment tree data structure extends naturally to an ordered sequence $a_1 < a_2 < \cdots < a_m$ via the correspondence between the interval $[i, j)$ and the interval $[a_i, a_j)$.

2.1. Problem 1. For the case of the Euclidean (L_2) metric the weighted Voronoi diagram (WVD) [5], [16] of the points in P provides an adequate solution to Problem 1. Such a Voronoi diagram divides the plane into $|P|$ regions (some possibly empty), there being a region $Vor(p)$ for each point $p \in P$. $Vor(p)$ is the region given by

$$Vor(p) = \{p'': \forall p' \in P,\ d(p'', p) - w(p) \leqq d(p'', p') - w(p')\}.$$

The WVD of P can be constructed in $O(|P| \log(|P|))$ time [5]. Furthermore, in $O(|P| \log(|P|))$ additional time it can be preprocessed, so that given a query point q, in $O(\log(|P|))$ time we can find a point $\bar{p}$ in P such that $q \in Vor(\bar{p})$ [3], [12]. So once the WVD of P is available, $nearest[q, P]$ can be obtained in $O(\log(|P|))$ time for any point q.

For the case of the L_1 metric we shall use the Willard–Lueker modification of the two-dimensional range tree [15] to provide a suitable solution to Problem 1. The range tree (RT) for P is as follows. At the top level is a segment tree for the nondecreasing sequence of x-coordinates of the points in P. At a segment tree node ψ associated with the interval $[a, b)$ of the x-axis is stored an ordered list of points in $P \cap H_x(a, b)$, with the points being ordered by y-coordinate. The segment tree also contains extra pointers from each internal node to its two children for efficient searching. The entire RT for P can be constructed in $O(|P| \log(|P|))$ time. To facilitate the search for $nearest[q, P]$ we shall store some additional information at each node. Let ψ be a node in the top level segment tree, and let $[a, b)$ be the interval associated with ψ. Let $p \in H_x(a, b) \cap P$. In addition to storing p in the ordered list at ψ, we store along with p the following points:

(1) $nearest[(a, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$.

(2) $nearest[(a, p_y), P \cap H_x(a, b) \cap H_y(-\infty, p_y)]$.

(3) $nearest[(b, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$.

(4) $nearest[(b, p_y), P \cap H_x(a, b) \cap H_y(-\infty, p_y)]$.

For a vertical strip $H_x(a, b)$ these additional points may be computed in $O(|P \cap H_x(a, b)|)$ time leading to a total of $O(|P| \log(|P|))$ for all the nodes in the tree. Suppose p, p' are adjacent points in the ordered list of points in $P \cap H_x(a, b)$ stored at node ψ. Also suppose that the query point q is such that $p_y \geqq q_y \geqq p'_y$. Then q satisfies the following two conditions:

(1) If $q_x \leqq a$ then one of the two points $nearest[(a, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$, and $nearest[(a, p'_y), P \cap H_x(a, b) \cap H_y(-\infty, p'_y)]$, is $nearest[q, P \cap H_x(a, b)]$.

(2) If $q_x \geqq b$ then one of the two points *nearest*$[(b, p_y), P \cap H_x(a, b) \cap H_y(p_y, \infty)]$, and *nearest*$[(b, p'_y), P \cap H_x(a, b) \cap H_y(-\infty, p'_y)]$ is *nearest*$[q, P \cap H_x(a, b)]$.
So given a query point q, we visit the $O(\log(|P|))$ nodes in the segment tree corresponding to the decomposition of the two segments $[-\infty, q_x)$, and $[q_x, \infty)$, locate q in the ordered lists at these nodes, and compute *nearest*$[q, P]$ in $O(\log(|P|))$ time.

LEMMA 1. *Given a set of points P on the plane, and a weight $w(p)$ associated with each point p in P, P can be preprocessed in $O(|P| \log(|P|))$ time, so that given a query point q, nearest$[q, P]$ and shortest$[q, P]$ can be found in $O(\log(|P|))$ time.* □

2.2. Problem 2. The data structure for Problem 2 has two levels. At the top level is a segment tree for the ordered sequence $p_1 < p_2 < \cdots < p_{|P|}$ of the points in P. At a segment tree node associated with the interval $[p_k, p_l)$ is stored the WVD for the set $[p_k, p_l)$ in the case of L_2 metric, and the RT for the set $[p_k, p_l)$ in the case of L_1 metric. Given an interval $[p_i, p_j)$, we visit the $O(\log(|P|))$ nodes in the segment tree corresponding to the decomposition of $[p_i, p_j)$, and search the WVD/RT at each of these nodes, and thereby compute *nearest*$[q, [p_i, p_j)]$ in $O((\log(|P|))^2)$ time. The data structure may be easily constructed in $O(|P|(\log(|P|))^2)$ time.

LEMMA 2. *Let $P = \{p_1 < p_2 < \cdots < p_{|P|}\}$ be an ordered set of points on the plane, and let there be a weight $w(p)$ associated with each point p in P. P can be preprocessed in $O(|P|(\log(|P|))^2)$ time, so that given a query point q and an interval $[p_i, p_j)$ such that $1 \leqq i < j \leqq |P|+1$, nearest$[q, [p_i, p_j)]$ and shortest$[q, [p_i, p_j)]$ can be found in $O((\log(|P|))^2)$ time.* □

2.3. Dynamizing the data structures for insertion. In this section we shall describe how to dynamize the data structures for Problems 1 and 2 (described in §§ 2.1 and 2.2, respectively) to handle the semidynamic case where new points may be inserted into P, but no points may be deleted from P. P is totally ordered by the following rule. For a pair of points p, p' in P, $p < p'$ if and only if p was added to P before p'.

There are standard techniques [13] for dynamizing a static data structure to support insertion, and we shall briefly describe how to apply one of them to the weighted Voronoi diagram and the range tree.

Let

$$|P| = \sum_{0 \leqq i \leqq \log_2(|P|)} a_i 2^i, \qquad a_i \in \{0, 1\},$$

be the binary representation of $|P|$. Let $P_1, P_2, \cdots$ be a partition of P such that:

(1) $|P_i| = a_i 2^i$, $0 \leqq i \leqq \log_2(|P|)$.

(2) If $i > j$ then each point in P_i was inserted into P before any of the points in P_j.

The *semidynamic* WVD (RT) is just a collection of WVDs (RTs), one for each nonempty P_i in the partition of P. Using the semidyanmic WVD (RT), given a query point q, *nearest*$[q, P]$ may be obtained $O((\log(|P|))^2)$ time. Furthermore, if we start with $P = \phi$ and there are a total of m insertions into P, then the total cost of maintaining the semidynamic WVD (RT) for P is $O(m(\log m)^2)$ operations [13].

The static data structure for Problem 2 described in § 2.2 may be dynamized to allow insertions into P in a similar manner. The dynamization increases the query time and the total time for all the insertions by a factor of at most $2 \log_2(|P|)$ [13].

LEMMA 3. *Let P be a set of points such that new points may be added to P, but no point may be deleted from P. With each point $p \in P$ is associated a weight $w(p)$. Suppose in the beginning $P = \phi$, and in the end P contains m points.*

(1) *We can maintain a semidynamic* WVD/RT *for P such that:*
  (1.1) *The total time for all the m insertions and thereby the total time for maintaining for the semidynamic* WVD/RT *is* $O(m(\log m)^2)$.
  (1.2) *Given a query point q, nearest*$[q, P]$ (*and shortest*$[q, P]$) *can be found in* $O((\log m)^2)$ *time.*

(2) *Suppose P is totally ordered by the following rule. For a pair of points p, p' in P,* $p < p'$ *if and only if p was added to P before p'. Also, let* $p_1 < p_2 < \cdots < p_{|P|}$ *be the ordered sequence of the points in P. Then we can maintain a semidynamic data structure for P such that:*
  (2.1) *The total time for inserting all the m points and thereby the total time for maintaining the semidynamic data structure is* $O(m(\log m)^3)$.
  (2.2) *Given a query point q, and an interval* $[p_i, p_j)$ *such that* $1 \leqq i < j \leqq |P|+1$, *nearest*$[q, [p_i, p_j)]$ (*and shortest*$[q, [p_i, p_j)]$) *can be found in* $O((\log m)^3)$ *time.* □

3. Weighted bipartite matching on the plane. We are given two sets U and V each consisting of n points on the plane. U and V induce a complete bipartite graph whose vertices are the points in U and V, and the weight of an edge (u_i, v_j), $u_i \in U$, $v_j \in V$, is the distance between u_i and v_j under some metric. We consider two metrics, the L_1 metric and the L_2 metric. The problem is to find a minimum weight complete matching in the complete bipartite graph on U and V.

Let $u_1, \cdots, u_n$ be an enumeration of the vertices in U, and let $v_1, \cdots, v_n$ be an enumeration of the vertices in V. A linear programming formulation [11] of the above bipartite weighted matching problem is:

$$\min \sum_{i,j} d(u_i, v_j) x_{ij}$$

$$\text{subject to} \quad \sum_j x_{ij} = 1, \qquad i = 1, \cdots, n,$$

$$\sum_i x_{ij} = 1, \qquad j = 1, \cdots, n,$$

$$x_{ij} \geqq 0$$

with the understanding that (u_i, v_j) is in the matching X if and only if $x_{ij} = 1$. The dual linear program is

$$\max \sum_i \alpha_i + \sum_j \beta_j$$

$$\text{subject to} \quad \alpha_i + \beta_j \leqq d(u_i, v_j), \qquad 1 \leqq i \leqq n, \quad 1 \leqq j \leqq n,$$

$$\alpha_i, \beta_j, \quad \text{unconstrained}$$

α_i and β_j are the dual variables associated with u_i and v_j, respectively. Orthogonality conditions that are necessary and sufficient for optimality of primal and dual solutions are:

$$x_{ij} > 0 \Rightarrow \alpha_i + \beta_j = d(u_i, v_j) \tag{3.1}$$

$$\alpha_i \neq 0 \Rightarrow \sum_j x_{ij} = 1, \qquad i = 1, \cdots, n, \tag{3.2}$$

$$\beta_j \neq 0 \Rightarrow \sum_i x_{ij} = 1, \qquad j = 1, \cdots, n. \tag{3.3}$$

We will use the version of the Hungarian method [10] for weighted bipartite matching given in [11]. The Hungarian method maintains dual feasibility at all times, and in addition maintains satisfaction of all orthogonality conditions except (3.3). Initially, we start with the empty matching $X = \phi$, and a feasible dual solution $\beta_j = \min_i \{d(u_i, v_j)\}$, $\alpha_i = 0$. The method proceeds in phases, and during each phase the matching X is augmented by an edge. Thus during each phase the number of violations of (3.3) is decreased by one.

During each phase we focus on those edges (u_i, v_j) such that $\alpha_i + \beta_j = d(u_i, v_j)$. These are the *admissible* edges. The *exposed* vertices are those that are not matched by the current matching X. An alternating path is one that alternately traverses an edge in the matching X, and an edge not in the matching X. An augmenting path is one that is between two exposed vertices. A phase consists of searching for an augmenting path among the admissible edges.

For each exposed vertex in V, we grow an alternating tree rooted at the vertex. Each vertex in $U \cup V$ that is in an alternating tree is reachable from the root of the tree via an alternating path that uses only admissible edges. Each vertex in an alternating tree is *labelled*, and each vertex that is not in any of the alternating trees is *free*. The root of an alternating tree is given an s-label of the form $[s, root]$. A matched vertex $v_j \in V$ in an alternating tree is given an s-label of the form $[s, u_k]$ where u_k is the vertex to which v_j is matched. A vertex $u_i \in U$ in an alternating tree is given a t-label of the form $[t, v_j]$, where v_j is the vertex from which u_i was labelled. An s-vertex (t-vertex) is a vertex with an s-label (t-label). Furthermore, an s-vertex (t-vertex) is reachable from an exposed vertex in V by an even (odd) length alternating path that uses only admissible edges.

Let $S(T)$ denote the set of all the s-vertices (t-vertices), and let F denote the set of all the free vertices in U. (Thus $F = U - T$.) Initially all the vertices are free, and to start with each exposed vertex in V is given an s-label. Thus at the beginning of a phase, S consists of the exposed vertices in V and $F = U$. Let δ be defined as

$$\delta = \min_{u_i \in F, v_j \in S} \{d(u_i, v_j) - \alpha_i - \beta_j\}.$$

We use a variable Δ to keep track of the sum of dual changes δ, and associate a weight $w(v)(w(u))$ with each vertex v in V (u in U). The weights are used to implement a phase efficiently. At the beginning of a phase $\Delta = 0$; for each $u_i \in U$, $w(u_i) = \alpha_i$; and for each $v_j \in V$, $w(v_j) = \beta_j$. Depending on whether δ equals zero or exceeds zero, the alternating trees are grown or there is a dual variable change.

Case 1. $\delta = 0$ (add to alternating trees or augment). Let (u_i, v_j), $u_i \in F$, $v_j \in S$ be an admissible edge, i.e., $d(u_i, v_j) - \alpha_i - \beta_j = 0$.

If u_i is exposed, then we have discovered an augmenting path among the admissible edges, and we can construct such a path by backtracking from v_j to the root of the alternating tree containing v_j using the labels on the vertices in the tree.

If u_i is matched to v_k, then give u_i the t-label $[t, v_j]$, and give v_k the s-label $[s, u_i]$,

$$F := F - \{u_i\}, \quad T := T \cup \{u_i\}, \quad S := S \cup \{v_k\},$$

$$w(u_i) := \alpha_i + \Delta, \qquad w(v_k) := \beta_k - \Delta. \qquad \square$$

Case 2. $\delta > 0$ (dual variable change). $\Delta := \Delta + \delta$;

For each vertex $u_i \in T$, $\alpha_i := \alpha_i - \delta$;
For each vertex $v_j \in S$, $\beta_j := \beta_j + \delta$. □

We note that for each s-vertex v_j, β_j equals $w(v_j)+\Delta$, and for each t-vertex u_i, α_i equals $w(u_i)-\Delta$. So during a phase, it suffices to maintain Δ and the weights associated with all the vertices, and there is no need to explicitly update the dual variables α_i, β_j, every time δ exceeds zero. At the end of a phase the correct values of the dual variables may be computed using Δ and the weights. A useful property of the weights is that the weight of a vertex changes only when it is labelled, and once it has been labelled its weight does not change during the remainder of the phase. Thus we may write

$$\delta = \min_{u \in F, v \in S} \{d(u, v) - w(u) - w(v)\} - \Delta.$$

We will use the data structures used in the solution of Problem 1 in § 2, namely, the weighted Voronoi diagram (WVD) and the range tree (RT), to efficiently compute δ, and an edge (u, v), $u \in F$, $v \in S$, such that $d(u, v) - w(u) - w(v) = \delta + \Delta$. Throughout a phase, S is partitioned into S_1 and S_2 such that $|S_2| \leqq \sqrt{n}$. Also, F is partitioned into $F_1, F_2, \cdots, F_{\lceil n^{0.5} \rceil}$, (some of the F_i's possibly empty) such that $|F_i| \leqq \lceil n^{0.5} \rceil$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. We maintain following data structures:

(1) A priority queue containing the edge *shortest*$[u, S_1]$ for each $u \in F$. The priority of an edge (u, v) in this queue is $d(u, v) - w(u) - w(v)$.

(2) A priority queue containing the edges *shortest*$[v, F_i]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$, for each vertex v in S_2. The priority of an edge (u, v) in this queue is also $d(u, v) - w(u) - w(v)$.

(3) The WVD/RT for each of the sets $F_1, F_2, \cdots, F_{\lceil n^{0.5} \rceil}$.

δ and an edge for which δ is achieved can be obtained in $O(\log n)$ time by examining an edge with minimum priority in (1) and (2) above. A new vertex added to S always gets inserted into S_2. In order to maintain the condition that $|S_2| \leqq \sqrt{n}$, whenever the size of S_2 reaches the threshold of $\sqrt{n}$ we add all the vertices in S_2 to S_1 and reset S_2 to the null set. Then *shortest*$[u, S_1]$ must be recomputed for every $u \in F$. From Lemma 1 in § 2, this recomputation may be done in $O(n \log n)$ time using a WVD/RT for S_1, leading to total of $O(n^{1.5} \log n)$ operations for the recomputation for the entire phase. Next we shall see that an insertion into S_2, and a deletion from F, each cost $O(n^{0.5} \log n)$ operations. As there are $O(n)$ such insertions and deletions in a phase, this leads to a total of $O(n^{1.5} \log n)$ operations per phase.

(i) Suppose a vertex v is inserted into S_2. Then we must compute *shortest*$[v, F_i]$, and insert it into the priority queue in (2) above, for $i = 1, \cdots, \lceil n^{0.5} \rceil$. Using the maintained WVD/RT for F_i, *shortest*$[v, F_i]$ can be found in $O(\log n)$ time. Hence an insertion costs $O(n^{0.5} \log n)$ operations.

(ii) Suppose a vertex u is deleted from F, and suppose $u \in F_i$. Then recomputing the WVD/RT for F_i requires $O(n^{0.5} \log n)$ time, and recomputing *shortest*$[v, F_i]$ for all the vertices v in S_2 and maintaining the priority queue also requires $O(n^{0.5} \log n)$ time. Thus a deletion costs $O(n^{0.5} \log n)$ operations.

Finally, since a phase takes $O(n^{1.5} \log n)$ time, and as there are at most n phases, the total running time of the weighted bipartite matching algorithm for points in the plane is $O(n^{2.5} \log n)$. In § 3.1 we shall see how to further improve the running time of the bipartite matching algorithm to $O(n^2(\log n)^3)$ for the case of L_1 metric.

3.1. Improving the complexity for the L_1 metric. In this section we shall show that for the case of the L_1 metric, the running time of the weighted bipartite matching algorithm for points on the plane can be further improved to $O(n^2(\log n)^3)$. Since we shall be dealing with the L_1 metric only in this section, $d(p, q)$ will denote the L_1

distance between p and q throughout this section. Note that maintaining the dual variables can be accomplished in $O(n)$ time per phase. To efficiently compute δ, we shall maintain a data structure containing all the points in $F \cup S$ such that:

(1) *shortest*(F, S) can be obtained in $O(1)$ time.

(2) A point in S can be inserted (or deleted) in $O((\log n)^3)$ time, and a point in F can be deleted (or inserted) in $O((\log n)^3)$ time.

Suppose that *shortest*$(F, S) = (u, v)$. Depending on whether $d(u, v) - w(u) - w(v) - \Delta$ equals or exceeds zero there is a dual variable change, and then (u, v) becomes admissible. If u is exposed then there is an augmentation and the phase ends. Otherwise, both u and the vertex to which u is matched get labelled and enter an alternating tree. u is deleted from F and the vertex to which u is matched is inserted into S. Correspondingly, there is a deletion from and an insertion into the above-mentioned data structure. Thus, using the above data structure it takes $O((\log n)^3)$ operations to increase the number of labelled vertices by 2, thereby leading to $O(n(\log n)^3)$ operations per phase. This leads to a running time of $O(n^2(\log n)^3)$ for the L_1 case.

Next, we describe a data structure for maintaining *shortest*$[F', S']$ for the special case when F' and S' are separated by a vertical line. Then using the data structure for this special case we shall show how to implement the above-mentioned data structure for maintaining *shortest*$[F, S]$. Let U' and V' be fixed subsets of U and V, respectively, such that all the points in U' lie on one side of the vertical line $x = a$, and all the points in V' line on the other side of the line $x = a$. (Points in U' and V' could lie on the line $x = a$.) Let $F' = F \cap U'$, and let $S' = S \cap V'$. Let $m = |U' \cup V'|$. Let $b_1 \leqq b_2 \leqq \cdots \leqq b_m$ be the nondecreasing sequence of the y-coordinates of the points in $U' \cup V'$. The data structure consists of a segment tree for this sequence of y-coordinates together with extra information stored at each node in the segment tree. Consider a node ψ in the segment tree associated with the interval $[b_i, b_j)$ of the y-axis. Let $k = \lceil (i+j)/2 \rceil$. At node ψ we store the following:

(i) Four priority queues, one for each of the four sets $F' \cap H_y(b_i, b_k)$, $F' \cap H_y(b_k, b_j)$, $S' \cap H_y(b_i, b_k)$, and $S' \cap H_y(b_k, b_j)$. The priority of a point p in any of the four queues is given by $d(p, (a, b_k)) - w(p)$.

(ii) The edge *shortest*$[F' \cap H_y(b_i, b_j), S' \cap H_y(b_i, b_j)]$.

Next, we shall see that once the two edges *shortest*$[F' \cap H_y(b_i, b_k), S' \cap H_y(b_i, b_k)]$ and *shortest*$[F' \cap H_y(b_k, b_j), S' \cap H_y(b_k, b_j)]$ are available, using the priority queues stored at node ψ, *shortest*$[F' \cap H_y(b_i, b_j), S' \cap H_y(b_i, b_j)]$ may be computed in $O(\log m)$ time. This is seen as follows. First, note that for a pair of points u and v, $u \in F' \cap H_y(b_i, b_k)$, and $v \in S' \cap H_y(b_k, b_j)$,

$$d(u, v) = d(u, (a, b_k)) + d(v, (a, b_k)),$$

and hence

$$d(u, v) - w(u) - w(v) = d(u, (a, b_k)) - w(u) + d(v, (a, b_k)) - w(v).$$

Thus

$$\min_{u \in F' \cap H_y(b_i, b_k),\, v \in S' \cap H_y(b_k, b_j)} \{d(u, v) - w(u) - w(v)\}$$

$$= \min_{u \in F' \cap H_y(b_i, b_k)} \{d(u, (a, b_k)) - w(u)\} + \min_{v \in S' \cap H_y(b_k, b_j)} \{d(v, (a, b_k)) - w(v)\}.$$

The same relationship holds for $u \in F' \cap H_y(b_k, b_j)$ and $v \in S' \cap H_y(b_i, b_k)$. Thus

shortest$[F' \cap H_y(b_i, b_k),\ S' \cap H_y(b_k, b_j)]$ and *shortest*$[F' \cap H_y(b_k, b_j),\ S' \cap H_y(b_i, b_k)]$ may be computed by examining the four points with minimum priority in the four priority queues described in (i) above. This requires $O(\log m)$ time. And once, *shortest*$[F' \cap H_y(b_i, b_k),\ S' \cap H_y(b_i, b_k)]$, *shortest*$[F' \cap H_y(b_k, b_j),\ S' \cap H_y(b_k, b_j)]$, *shortest*$[F' \cap H_y(b_i, b_k),\ S' \cap H_y(b_k, b_j)]$, and *shortest*$[F' \cap H_y(b_k, b_j),\ S' \cap H_y(b_i, b_k)]$, are available, *shortest*$[F' \cap H_y(b_i, b_j),\ S' \cap H_y(b_i, b_j)]$ is computable in constant time.

We now show that a point may be inserted into or deleted from the data structure for maintaining *shortest*$[F', S']$ in $O((\log m)^2)$ time. When a point p is added to S' (or F'), it is inserted into the priority queues at the nodes in the segment tree that lie on the path from the root to the leaf corresponding to the point p. So there are $O(\log m)$ insertions, each costing $O(\log m)$ operations. We then start from the leaf corresponding to the inserted point p and work our way up toward the root, updating in sequence the edge (as specified by (ii) above) that is stored at each of the $O(\log m)$ nodes on the path from the leaf to the root. Let ψ be a node on this path. Once the updates at nodes on the subpath from the leaf to ψ have been performed, the update at node ψ may be performed in $O(\log m)$ time using the priority queues stored at ψ. Thus, whenever there is an insertion, *shortest*$[F', S']$ is updated in $O((\log m)^2)$ operations. The same bound holds when a point is deleted from F' (or S').

The data structure for maintaining *shortest*$[F, S]$ is a two level data structure. At the top level is a segment tree for the nondecreasing sequence $a_1 \leqq a_2 \leqq \cdots \leqq a_{2n}$ of the x-coordinates of the points in $U \cup V$, and at each node is a data structure for the special case mentioned above. Specifically, let ψ be a node in the top level segment tree and let $[a_i, a_j)$ be the interval of the x-axis associated with ψ. Let $k = \lceil (i+j)/2 \rceil$. At node ψ we store the following:

(I). A data structure as described above for maintaining each of the two edges *shortest*$[F \cap H_x(a_i, a_k),\ S \cap H_x(a_k, a_j)]$ and *shortest*$[F \cap H_x(a_k, a_j),\ S \cap H_x(a_i, a_k)]$. This is the secondary data structure stored at ψ.

(II). The edge *shortest*$[F \cap H_x(a_i, a_j),\ S \cap H_x(a_i, a_j)]$.

We note that the edge stored at the root (as specified by (II) above) is *shortest*$[F, S]$. When a point is inserted, we start at the leaf in the top level segment tree corresponding to the inserted point, and perform updates in sequence at the nodes on the path from this leaf to the root. Suppose ψ is a node on this path. Once the updates at nodes on the subpath from the leaf to ψ have been performed, the edges *shortest*$[F \cap H_x(a_i, a_k),\ S \cap H_x(a_i, a_k)]$ and *shortest*$[F \cap H_x(a_k, a_j),\ S \cap H_x(a_k, a_j)]$, are available. The updates in the secondary data structure at ψ are performed in $O((\log n)^2)$ time, and then *shortest*$[F \cap H_x(a_i, a_k),\ S \cap H_x(a_k, a_j)]$, and *shortest*$[F \cap H_x(a_k, a_j),\ S \cap H_x(a_i, a_k)]$, are also available. Then *shortest*$[F \cap H_x(a_i, a_j),\ S \cap H_x(a_i, a_j)]$ is computable in constant time. Since there are $O(\log n)$ nodes on the path from a leaf in the top level segment tree to the root, the total cost of an insertion is $O((\log n)^3)$ operations. The same bound holds for deleting a point.

4. Weighted general matching on points in the plane. We are given a set V of $2n$ points in the plane. The set of points V induces a complete graph whose vertices are the points in V, and the weight of an edge between two points is the distance between the points under some metric. We shall consider two cases, one where the weight of an edge is given by the Euclidean (L_2) distance between the two endpoints of the edge, and the other where the weight of an edge is given by the L_1 distance between the endpoints of the edge. We let $v_1, v_2, \cdots, v_{2n}$ denote the $2n$ vertices (points) in V. Let O denote the set of all those subsets of V having cardinality that is odd and greater than one.

The MWCM problem in the complete graph on V is formulated as a linear program [4], [11], [14]:

$$\min \sum_{1 \leqq i < j \leqq 2n} d(v_i, v_j) x_{ij},$$

$$\text{subject to} \quad \sum_{1 \leqq j \leqq 2n, j \neq i} x_{ij} = 1, \qquad i = 1, \cdots, 2n$$

$$\forall B \in O, \quad \sum_{v_i, v_j \in B} x_{ij} \leqq \frac{|B| - 1}{2},$$

$$x_{ij} \geqq 0, \qquad 1 \leqq i < j \leqq 2n$$

with the understanding that $x_{ij} = 1$ if and only if (v_i, v_j) is in the matching X.

The dual linear program is given by

$$\max \sum_{i=1}^{2n} \alpha_i + \sum_{B \in O} z(B)$$

$$\text{subject to} \quad \alpha_i + \alpha_j + \sum_{v_i, v_j \in B, B \in O} z(B) \leqq d(v_i, v_j), \qquad 1 \leqq i < j \leqq 2n$$

$$\forall B \in O, \quad z(B) \leqq 0.$$

α_i is the dual variable associated with vertex v_i, and $z(B)$ is the dual variable associated with the odd set B. Orthogonality conditions that are necessary and sufficient for optimality of primal and dual solutions are:

$$x_{ij} > 0 \Rightarrow \alpha_i + \alpha_j + \sum_{v_i, v_j \in B, B \in O} z(B) = d(v_i, v_j), \tag{4.1}$$

$$\alpha_i \neq 0 \Rightarrow \sum_{1 \leqq j \leqq 2n, j \neq i} x_{ij} = 1, \tag{4.2}$$

$$z(B) < 0 \Rightarrow \sum_{v_i, v_j \in B} x_{ij} = \frac{|B| - 1}{2}. \tag{4.3}$$

Edmond's algorithm [4], [11], [14] for finding a minimum weight complete matching maintains dual feasibility at all times, and in addition maintains the satisfaction of all orthogonality conditions, except conditions (4.2). The algorithm starts with the empty matching $X = \phi$, and a feasible dual solution given by $\alpha_i = \frac{1}{2} \min_{v_j \in V - \{v_i\}} d(v_i, v_j)$, $i = 1, \cdots, 2n$, and $z(B) = 0$, for all $B \in O$. The algorithm proceeds in phases. During each phase the cardinality of the matching X increases by one thereby decreasing the number of violations of (4.2) by one.

An $O(nm \log n)$ implementation of Edmond's algorithm for finding a minimum weight complete matching in a graph with n vertices and m edges is given in [8]. We shall utilize the underlying geometry together with some of the ideas in [8] to obtain an $O(n^{2.5}(\log n)^4)$ algorithm for finding an MWCM on $2n$ points on the plane. First, we shall sketch Edmond's algorithm, and then show how to efficiently implement it for points in the plane.

4.1. Blossoms and their representation. As the algorithm proceeds, it discovers certain subsets of V (of odd size) called *blossoms*. It is convenient to consider the vertices of V as (trivial) blossoms of size one. For a pair of blossoms B, B', either $B \cap B' = \phi$, $B \subseteq B'$, or $B' \subseteq B$, and a blossom that is not a subset of any other blossom is a *maximal* blossom. The algorithm has $z(B) < 0$ only for those $B \in O$ that are nontrivial blossoms. Consequently, the number of nonzero $z(B)$'s is $O(n)$.

The discussion below about blossoms follows [8]. The representation that we use for blossoms is identical to the one in [8] but we shall describe it here for completeness.

An edge (v_i, v_j) is *admissible* if and only if

$$\alpha_i + \alpha_j + \sum_{v_i, v_j \in B, B \in O} z(B) = d(v_i, v_j).$$

An *alternating path* from v_0 to v_r is a sequence of edges $\{e_i = (v_{i-1}, v_i)\}_{i=1}^{r}$ such that for $i = 1, \cdots, r-1$, $e_i \in X$ if and only if $e_{i+1} \notin X$. An *alternating path* from a maximal blossom B_0 to a maximal blossom B_r (possibly $B_0 = B_r$) is a sequence of edges $\{e_i = (u_{i-1}, v_i)\}_{i=1}^{r}$ such that for $i = 0, \cdots, r$, $u_i, v_i \in B_i$, where $B_0, \cdots, B_r$ are distinct maximal blossoms, and for $i = 1, \cdots, r-1$, $e_i \in X$ if and only if $e_{i+1} \notin X$. When the algorithm discovers an odd length alternating path, that uses only admissible edges, from a maximal blossom B_0 to itself (i.e., $B_0 = B_r$, $e_1, e_r \notin X$), a new blossom B is formed. $B_1, \cdots, B_r$ are the subblossoms of the new blossom B. (Note: A blossom that is a proper subset of B_i, for some i, $1 \leqq i \leqq r$, is not a subblossom of B.) Each blossom has a *base* vertex. The base of a trivial blossom is the unique vertex in it. The base of the blossom B defined above is the base of B_0 $(= B_r)$.

A nontrivial blossom B is represented by the cyclic double-linked list $\{(B_i, e_i)\}_{i=1}^{r}$, and by its base. (From an entry (B_i, e_i) in the list there are bidirectional links to (B_l, e_l) where $l = (i \bmod r) + 1$.) B_i, $i = 1, \cdots, r$ are the subblossoms of B, and B_r is the distinguished subblossom that contains the base of B. For each i, $1 \leqq i \leqq r-1$, $(e_1, e_2, \cdots, e_i)$ and $(e_r, e_{r-1}, \cdots, e_{i+1})$ are alternating paths that use admissible edges only from B_r to B_i. The one of even length is the one whose last edge is in X.

A blossom B of size $2k+1$ has the following two properties. First, there are exactly k matched edges (i.e., edges in X) both of whose endpoints are in B, and the base of B is that vertex in B not an endpoint of one of these k edges. Second, there is an even length alternating path, which uses only admissible edges, from the base of B to every vertex in B. As a consequence an alternating path, which uses admissible edges only, from a maximal blossom B to a maximal blossom B' can be expanded into an alternating path, which uses admissible edges only, from the base of B to the base of B'.

The structure of a maximal blossom B can be represented by a tree. In this tree the sons of a blossom B are its subblossoms $B_1, \cdots, B_r$, and the leaves are the vertices of B. This tree will be denoted as the *structure tree* of B. The structure tree of B is implicitly represented by the cyclic double-linked lists $\{(B_i, e_i)\}_{i=1}^{r}$ corresponding to B, its subblossoms, the subblossoms of its subblossoms, and so on. The tree implies a total order on the vertices in B: $v_i < v_j$ if and only if v_i is to the left of v_j in the tree. The base of B is largest vertex in this ordering. Furthermore, a blossom B' that is a subset of B corresponds to an interval in this ordering. Also note that the base of a blossom changes only when the matching X is augmented, and so during a phase of the algorithm the ordering of the vertices in an existing blossom does not change.

During the course of the matching algorithm, given a vertex v_j we shall need to know the identity of the maximal blossom containing v_j. To be able to obtain this identity efficiently, we also represent the maximal blossoms as ordered sets of vertices. These ordered sets are implemented as *concatenable queues* [1]. Such queues support the operations of find, split, and concatenate [1] in time proportional to the logarithm of the number of items in the queue. So given a vertex v_j, the base of the maximal blossom containing v_j and thereby the identity of the maximal blossom containing v_j can be computed in $O(\log n)$ time. When a blossom B splits into r subblossoms, the splitting of the concatenable queue for B into the concatenable queues for the subblossoms of B may be carried out in $O(r \log n)$ time. Similarly, when a new blossom

B is formed from r blossoms, the concatenation of the r concatenable queues of the subblossoms of B to obtain the concatenable queue for B may also be carried out in $O(r \log n)$ time.

4.2. A phase of the algorithm. An edge in the matching X is said to be matched, and an edge not in X is said to be unmatched. A vertex is *matched* if there is an edge in X incident to it. A vertex is *exposed* if there is no edge in X incident to it. A blossom is exposed (matched) if and only if its base is exposed (matched). An *augmenting path* is an alternating path between two distinct exposed maximal blossoms or between two distinct exposed vertices. A phase consists of looking for an augmenting path, that uses admissible edges only, between two exposed maximal blossoms.

We grow an alternating tree rooted at each exposed maximal blossom. The nodes in an alternating tree are themselves maximal blossoms. Each node in an alternating tree is reachable from the root by an alternating path that uses only admissible edges. Each maximal blossom that is a node in an alternating tree is *labelled*, and each maximal blossom that is not in any of the alternating trees is *free*. (Note that only a maximal blossom can be labelled or free.) A node in an alternating tree that is reachable by an even (odd) length alternating path from the root is given an s-label (t-label). An s-label (t-label) given to a node B is of the form $[s, (v_i, v_j)]([t, (v_i, v_j)])$ where $v_j \in B$ and v_i is the first vertex (point) on the alternating path from B to the root of the alternating tree containing B. A node with an s-label (t-label) is called an s-blossom (t-blossom). A vertex in an s-blossom (t-blossom) is referred to as an s-vertex (t-vertex). A vertex in a free blossom is called a free vertex. We let $S(T)$ denote the set of all the s-vertices (t-vertices), and let F denote the set of free vertices. (Note that $V = S \cup T \cup F$.)

At the start of a phase each maximal exposed blossom is given an s-label, and each maximal matched blossom is free. (So initially, $T = \phi$.) A phase consists of several steps and at each step one of following four things must happen. Either there is an admissible edge between two s-vertices not in the same s-blossom or there is an admissible edge between an s-vertex and a free vertex or the dual variable corresponding to a t-blossom is zero or there is a dual variable change. In the first case an alternating tree can be grown, in the second case an augmenting path is found or a new blossom is discovered, in the third case a t-blossom expands, and an occurrence of the fourth case is always followed by an occurrence of one the first three cases in the next step. Let

$$\delta_1 = \min_{v_i \in S, v_j \in F} \{d(v_i, v_j) - \alpha_i - \alpha_j\},$$

$$\delta_2 = \min_{v_i, v_j \in S, v_i, v_j \text{ not in the same } s\text{-blossom}} \left\{\frac{d(v_i, v_j) - \alpha_i - \alpha_j}{2}\right\},$$

$$\delta_3 = \min_{B \text{ a } t\text{-blossom}} \left\{\frac{-z(B)}{2}\right\},$$

and

$$\delta = \min\{\delta_1, \delta_2, \delta_3\}.$$

A detailed description of the four cases that can occur at a step during a phase is given below. Δ accumulates the dual variable changes δ during a phase, and at the start of a phase $\Delta = 0$.

Case 1. $\delta_1 = 0$ (grow an alternating tree). In this case there is an admissible edge between an s-vertex and a free vertex. Let (v_i, v_j) be such an edge where v_i is an

s-vertex and v_j is in a free blossom B. Let b be the base of B and let (b, v_k) be the matched edge incident on b. Then v_k is also a free vertex, and is in some free blossom B'. B gets the label $[t, (v_i, v_j)]$, and B' gets the label $[s, (b, v_k)]$.

By using the representations of the blossoms as ordered sets (i.e., the concatenable queues), the maximal blossoms containing v_i, v_j, and v_k, may be found in $O(\log n)$ time, and so once the edge (v_i, v_j) is available the labelling takes $O(\log n)$ time. □

Case 2. $\delta_2 = 0$ (discover a new blossom or augment). In this case there is an admissible edge between two s-vertices not in the same s-blossom. Let (v_i, v_j) be such an edge, and let v_i, v_j, be in distinct s-blossoms B, B', respectively. Using the labels on blossoms, we backtrack along the alternating path from B to the root of the alternating tree containing B. Simultaneously, we also backtrack along the alternating path from B' to the root of the alternating tree containing B'. We make a careful backtrack by backtracking one blossom on both paths each time, marking blossoms along the way. Either we discover a new blossom or find an augmenting path.

(i) A new blossom is found. Suppose the new blossom has r subblossoms. Then we will visit at most $2r$ blossoms before finding the first common blossom C on both paths. We use the parts of the paths from C to B and from C to B' to generate the cyclic double-linked list $\{(B_i, e_i)\}_{i=1}^{r}$ for the new blossom, where $B_r = C$ and e_i are taken from the labels on the two paths. The base of the new blossom is the base of C. The new blossom gets an s-label, and the subblossoms of the new blossom get unlabelled and stop being s-blossoms/t-blossoms (since they are no longer maximal blossoms). The dual variable associated with the new blossom is initialized to zero. Using the concatenable queues and the labels, finding the common blossom C and constructing the new cyclic double-linked list requires $O(r \log n)$ operations. The concatenable queue for the new blossom is obtained by concatenating together the concatenable queues for the r subblossoms in $O(r \log n)$ time.

(ii) An augmenting path is discovered between two exposed blossoms. If we discover an augmenting path π between two exposed blossoms, then we construct an augmenting path between the base vertices of the two exposed blossoms as follows. For each blossom on the path π, we recursively find an even length alternating path between the base of the blossom and the vertex by which the path π leaves the blossom. Once the even length alternating paths within all the blossoms that lie on the path π are available, we connect up these paths using the edges on π to give an augmenting path between the base vertices of the two exposed blossoms. The resulting augmenting path uses admissible edges only. We switch the status of the edges on the resulting augmenting path from matched to unmatched and vice versa. This augments the matching X and the current phase ends. We note that for each blossom through which the augmenting path passes, the base vertex changes and the ordering of the vertices in the blossom that is implied by the structure tree also changes. □

Case 3. $\delta_3 = 0$ (expand a t-blossom). Let B be a t-blossom such that the dual variable $z(B)$ associated with B is zero, and let B be labelled $[t, (v_j, v_i)]$. Suppose that $v_i \in B_i$ where $B_1, \cdots, B_r$ are the subblossoms of B and B_r contains the base of B. The blossom B expands, the blossoms on the odd length alternating path from B_i to B_r become free, and the blossoms on the even length alternating path from B_i to B_r get alternating labels starting and ending with a t-label. The labels are generated using the edges e_i in the cyclic double-linked list for B. The concatenable queues for the subblossoms of B are obtained by splitting for $i = 1, \cdots, r-1$ each B_i from B according to its base, which is its largest element.

Using the cyclic double-linked lists and the concatenable queues, the expansion of a t-blossom B with r subblossoms can be carried out in $O(r \log n)$ time. □

Case 4. $\delta > 0$ (dual variable change). In this case the dual variables are changed as follows:

For every s-vertex v_i, $\alpha_i := \alpha_i + \delta$;
For every t-vertex v_j, $\alpha_j := \alpha_j - \delta$;
For every s-blossom B, $z(B) := z(B) - 2\delta$;
For every t-blossom B', $z(B') := z(B') + 2\delta$;
$\Delta := \Delta + \delta$. □

At the end of a phase we construct the augmenting path, and augment the matching X by switching the status of the edges on the augmenting path from matched to unmatched and vice versa. We then change the base of each blossom through which the augmenting path passes, and rebuild the concatenable queues for all the maximal blossoms through which the augmenting path passes, since changing the base changes the ordering of the vertices in the blossom. Finally, we expand all maximal blossoms B such that $z(B) = 0$. (This ensures that at the start of the next phase the dual variable associated with each maximal blossom has nonzero value.)

We have the following useful lemma concerning the behavior of blossoms in a phase, which follows directly from Cases 1–4 described above.

LEMMA 4. (a) *Each blossom that gets an s-label some time during the phase, corresponds to a unique node in the structure tree of some maximal blossom at the end of the phase. Each blossom that gets a t-label or becomes free some time during the phase, corresponds to a unique node in the structure tree of some maximal blossom at the beginning of the phase.*

(b) *A free blossom is either free from the beginning of the phase or becomes free because it is the subblossom of some t-blossom that expands. Once a blossom receives a t-label, either it stays as a t-blossom until the phase ends, it expands, or it loses its label because it becomes the subblossom of a new s-blossom, in which case it stays unlabelled until the phase ends. Once a blossom receives an s-label, it either remains an s-blossom, until the phase ends, or it loses its label because it becomes the subblossom a new s-blossom, in which case it stays unlabelled until the phase ends.*

(c) *The total number of free blossoms plus s-blossoms plus t-blossoms in a phase is* $O(n)$. □

The remainder of § 4 will be devoted to the efficient implementation of a phase. We shall assign a weight $w(v_i)$, suitably related to the dual variable α_i, to each vertex v_i in V, a weight $w(B)$ to each s-blossom/t-blossom B, and also an offset $\mu(B)$ to each blossom B, which measures the change in the value of the dual variable of a vertex in B. In § 4.3 we shall describe how to compute these weights and offsets, and show how to efficiently maintain the dual variables using these weights and offsets. In §§ 4.4 and 4.5 these weights and offsets will be used along with the underlying geometry to construct a scheme for efficiently maintaining δ_1 and δ_2, respectively. The running time per phase may be broken down as follows:

(1) The computation at the end of a phase (i.e., constructing the augmenting path, etc.) requires $O(n \log n)$ time.

(2) In § 4.3 it is shown that the time for computing the weights and the offsets and for maintaining the dual variables using the weights and the offsets is $O(n^{1.5})$ per phase.

(3) In § 4.4 we show that the time required for maintaining δ_1, and an appropriate edge between an s-vertex and a free vertex for which δ_1 is achieved, is $O(n^{1.5}(\log n)^2)$ per phase. Furthermore, since Case 1 can occur $O(n)$ times during one phase, $O(n \log n)$ extra time is spent in Case 1 per phase in addition to the time required for maintaining δ_1. So the total time spent in Case 1 is $O(n^{1.5}(\log n)^2)$ per phase.

(4) In § 4.5 we show that the time required to maintain δ_2, and an appropriate edge between two s-vertices not in the same s-blossom for which δ_2 is achieved, is $O(n^{1.5}(\log n)^4)$ per phase. From Lemma 4 above it follows that $O(n \log n)$ extra time is spent in Case 2 per phase in addition to the time required for maintaining δ_2. So the total time spent in Case 2 is $O(n^{1.5}(\log n)^4)$ per phase.

(5) To maintain δ_3 the t-blossoms are kept in a priority queue, and the priority of a t-blossom B in this queue is given by $-w(B)$. Each t-blossom B satisfies the condition $z(B) = w(B) + 2\Delta$, and so the t-blossom with the largest weight is also the t-blossom with largest value for the dual variable $z(B)$. Then from Lemma 4 above it follows that $O(n \log n)$ time is spent in Case 3 per phase in addition to the time for computing the weights $w(B)$ of blossoms.

Thus the time required by the algorithm per phase is $O(n^{1.5}(\log n)^4)$. Since the number of phases is at most n, the total running time is $O(n^{2.5}(\log n)^4)$.

4.3. Weights, offsets, and dual variables. It is too time consuming to explicitly update the dual variables whenever there is a dual variable change (i.e., whenever Case 4 in § 4.2 occurs), so they are implicitly maintained by associating weights w with the vertices and the blossoms, and offsets μ with the blossoms. Similar ideas are also used in [8] to implicitly maintain the dual variables. Let $w(v_i)$ denote the weight associated with vertex v_i, and let $\mu(B)$ denote the offset associated with blossom B. We divide blossoms into *large* and *small* blossoms according to their size: a large blossom is one that contains at least $\sqrt{n}$ vertices, and a small blossom is one that contains less than $\sqrt{n}$ vertices. As mentioned in § 4.2, Δ denotes the sum of the dual variable changes δ since the beginning of the phase. The weights associated with vertices and the offsets associated with blossoms are updated so that after each step of a phase (that consists of executing one of the four cases in § 4.2) the following four relationships are maintained:

(1) For each s-vertex v_i, $\alpha_i = w(v_i) + \Delta$.

(2) For each vertex v_j in a t-blossom B, $\alpha_j = w(v_j) + \mu(B) - \Delta$.

(3) For each vertex v_k in a large free blossom B', $\alpha_k = w(v_k) + \mu(B')$.

(4) For each vertex v_l in a small free blossom B'', $\alpha_l = w(v_l)$.

At any time during a phase, to compute the value of the dual variable associated with a t-vertex or a free vertex we find the maximal blossom containing the vertex and then use the above relations. So at any time during the phase, the value of the dual variable associated with a vertex is computable in $O(\log n)$ time. At the end of a phase the values of the dual variables associated with all the vertices are explicitly computed.

At the beginning of a phase the weights and offsets are initialized as follows.

For each $v_i \in V$, $w(v_i) := \alpha_i$;

For each blossom B, $\mu(B) := 0$.

Note that $\Delta = 0$ at the beginning of a phase.

Next we shall describe how to update the weights and offsets during each of the four cases that occur at a step during a phase. (These cases were described in § 4.2.)

Case 1. $\delta_1 = 0$. In this case a free blossom B gets a t-label, and a free blossom B', such that a matched edge joins the base of B to a vertex in B', gets an s-label. The weights and offsets are updated as follows.

If newly labelled t-blossom B is a large blossom

then $\mu(B) := \mu(B) + \Delta$

else $\mu(B) := \Delta$;

For each vertex v_i in newly labelled s-blossom B',

If B' is a large blossom

then $w(v_i) := w(v_i) + \mu(B') - \Delta$
else $w(v_i) := w(v_i) - \Delta$. □

Case 2. $\delta_2 = 0$. The weights and offsets need to be updated only in the case when a new blossom is discovered in Case 2. Let B be the new blossom that is discovered. Note that B gets an s-label, and all the vertices in each subblossom of B, which had a t-label, switch status from t-vertices to s-vertices.

For each subblossom B' of B which had a t-label,
For each vertex v_i in B', $w(v_i) := w(v_i) + \mu(B') - 2\Delta$. □

Case 3. $\delta_3 = 0$. In this case a t-blossom B expands, and each of its subblossoms either becomes free or gets an s- or a t-label. We perform the following updates:

For each subblossom B' of B,
If B' gets a t-label then $\mu(B') := \mu(B)$;
If B' gets an s-label then
For each vertex v_i in B', $w(v_i) := w(v_i) + \mu(B) - 2\Delta$;
If B' is a large blossom and B' becomes free then $\mu(B') := \mu(B) - \Delta$;
If B' is a small blossom and B' becomes free then
For each vertex v_i in B', $w(v_i) := w(v_i) + \mu(B) - \Delta$. □

Case 4. $\delta > 0$. $\Delta := \Delta + \delta$. □

It is easily verified that the updates to the weights and the offsets maintain the above-mentioned four relationships between the weights, the offsets, and the dual variables. A blossom switches status whenever one of three things happens: (1) it is free and gets labelled; (2) it is labelled and gets unlabelled because it becomes the subblossom of a new blossom; or (3) it becomes free or labelled because it is the subblossom of some blossom that expands. From Lemma 4 in § 4.2, the total number status switches of blossoms is $O(n)$ per phase. So the total number of updates to the offsets μ is $O(n)$, since $\mu(B)$ is updated only when B switches status. The weight of a vertex is updated when it switches status from t-vertex/free vertex to s vertex, and whenever it becomes free and is in a small free blossom. Since a vertex switches status to s-vertex at most once during a phase, since the total number of status switches of blossoms is $O(n)$ per phase, and since a small blossom contains at most $\sqrt{n}$ vertices, we get that the total number of updates to weights w of vertices is $O(n^{1.5})$ per phase. So maintaining the weights of all the vertices and the offsets of all the blossoms requires $O(n^{1.5})$ time per phase.

The values of the dual variables associated with s-blossoms and t-blossoms are also implicitly maintained by weights associated with these blossoms. Let $w(B)$ denote the weight associated with blossom B. When a blossom B receives a t-label we initialize its weight as

$$w(B) := z(B) - 2\Delta.$$

As long as B remains a t-blossom the value of $z(B)$ is implicitly given by the relation

$$z(B) = w(B) + 2\Delta.$$

When B stops being a t-blossom (by becoming a subblossom of a new blossom and thereby getting unlabelled) we explicitly compute the value of the dual variable $z(B)$. When a blossom B' receives an s-label we initialize its weight as

$$w(B') := z(B') + 2\Delta,$$

and for the duration of the period for which B' is an s-blossom the value of $z(B')$ is implicitly given by

$$z(B') = w(B') - 2\Delta.$$

When B' stops being an s-blossom (by becoming a subblossom of a new blossom and thereby getting unlabelled) we explicitly compute the value of the dual variable $z(B')$. Since the total number of times blossoms are labelled is $O(n)$ per phase, the weights of the blossoms can be maintained in $O(n)$ time per phase.

4.4. Maintaining δ_1 during a phase. In this section we shall describe how to maintain δ_1, and an edge between an s-vertex and a free vertex such that δ_1 is achieved for that edge. The slack associated with an edge (v_i, v_j) is given by the quantity $d(v_i, v_j) - \alpha_i - \alpha_j$, and denoted by $slack[(v_i, v_j)]$. Let

$$\delta_{11} = \min_{v_i \in S, v_j \text{ in a large free blossom}} \{slack[(v_i, v_j)]\},$$

and

$$\delta_{12} = \min_{v_i \in S, v_j \text{ in a small free blossom}} \{slack[(v_i, v_j)]\}.$$

Then clearly,

$$\delta_1 = \min\{\delta_{11}, \delta_{12}\}.$$

In §§ 4.4.1 and 4.4.2 we shall describe how to maintain δ_{11} and δ_{12} in $O(n^{1.5}(\log n)^2)$ and $O(n^{1.5} \log n)$ time per phase, respectively.

4.4.1. Maintaining δ_{11}. During the computation of δ_{11}, we will be required to quickly find $shortest[v_j, B]$ for an s-vertex v_j and a large free blossom B. A data structure for doing this is obtained as follows. From Lemma 4 (§ 4.2), a free blossom B is a blossom that existed at the beginning of the phase. Consider the maximal blossom B' that contained B at the start of the phase, and the ordering imposed on the vertices of B' by the structure tree of B'. The free blossom B corresponds to an interval in this ordering, in other words if $v_i, v_l \in B$ and $v_i < v_l$ then each vertex v_k in B' such that $v_i < v_k < v_l$ is also in B. So the problem is to preprocess each maximal blossom B' at the start of a phase, so that later on in the phase, given a vertex v_j and a blossom B that is a subset of B' (and therefore corresponds to an interval in the ordering on B'), $shortest[v_j, B]$ may be found quickly. This is precisely Problem 2 given in § 2. So, by Lemma 2 (§ 2.2) we can preprocess all the maximal blossoms B' that exist at the start of a phase in $O(n(\log n)^2)$ time, so that given a free blossom B later in the phase together with a vertex v_j, $shortest[v_j, B]$ can be found in $O((\log n)^2)$ time.

As in the bipartite case, the set of s-vertices S is partitioned into S_1, S_2, such that $|S_2| \leqq \sqrt{n}$. We maintain the following edges:

(1) For each large free blossom B, the edge $shortest[B, S_1]$. All these edges are in a priority queue, with the priority of an edge (v_i, v_j) being given by $slack[(v_i, v_j)]$.

(2) For each large blossom B' that is either a subset of a t-blossom or a subset of a free blossom, the edge $shortest[B', S_1]$.

(3) For each pair v_j, B, where $v_j \in S_2$ and B is a large free blossom, the edge $shortest[v_j, B]$. All these edges are also in a priority queue, with the priority of an edge being given by its slack.

We note that for each vertex v_i in a large free blossom B, $\alpha_i = w(v_i) + \mu(B)$ where $\mu(B)$ is the offset associated with B, and that for each s-vertex v_j, $\alpha_j = w(v_j) + \Delta$. Thus if B is a large free blossom and $S' \subseteq S$, then

$$slack[shortest[B, S']] = \min_{v_i \in B, v_j \in S'} \{slack[(v_i, v_j)]\}.$$

So, δ_{11} is achieved either for the edge in (1) above with minimum priority or for the edge in (3) above with minimum priority, and may be computed in $O(\log n)$ time by examining these two edges.

At the beginning of a phase, S_1 consists of the vertices in exposed blossoms and S_2 is empty. Note that a vertex is never deleted from S. A new s-vertex is always inserted into S_2. Whenever the size of S_2 reaches the threshold of $\lceil n^{0.5} \rceil$, we add all the vertices in S_2 to S_1 and reset S_2 to the empty set. When vertices are moved from S_2 to S_1 the edges in (1) and (2) above must be recomputed, and this may be accomplished in $O(n \log n)$ time as follows. We utilize the data structures for Problem 1 described in § 2.1. We construct the weighted Voronoi diagram/range tree (WVD/RT) for the points in S_1, and for each vertex v_i that is a free vertex or a t-vertex, we compute $shortest[v_i, S_1]$ by querying the WVD/RT for S_1. This requires $O(n \log n)$ time by Lemma 1 in § 2.1. Then using the structure trees of the t-blossoms and the free blossoms, the edges in (1) and (2) above can be computed in $O(n)$ extra time. As the recomputation is done at most $\sqrt{n}$ times per phase, the time for the recomputation is $O(n^{1.5} \log n)$ for the entire phase.

Note that for each vertex v_i in a large t-blossom or in a large free blossom, the weight $w(v_i)$ equals the value of dual variable α_i at the beginning of the phase, and so the edges in (1) and (2) above need to be recomputed only when S_1 changes.

Next, we show that the time required for updating the edges in (1), (2), and (3) above whenever a large blossom becomes free or a large free blossom gets labelled or a vertex is inserted into S_2 is $O(\sqrt{n}(\log n)^2)$.

(a) Suppose a t-blossom expands and one of its subblossoms, say B, becomes free. Also, suppose B is a large blossom. Then the edge $shortest[B, S_1]$, which is one of the edges in (2) above, is inserted into the priority queue in (1) above. For each vertex $v_j \in S_2$, the edge $shortest[v_j, B]$ is computed in $O((\log n)^2)$ time using the data structure described at the start of the section and is inserted into the priority queue in (3) above. This leads to a cost of $O\sqrt{n}(\log n)^2)$ operations when a large blossom becomes free.

(b) Suppose a large free blossom gets a t-label/s-label. Then the $O(\sqrt{n})$ edges that are incident to vertices in B are deleted from the priority queues in (1) and (3) above.

(c) Suppose a vertex v_j is inserted into S_2. Then for each large free blossom B, we find $shortest[v_j, B]$ in $O((\log n)^2)$ time, and insert it into the priority queue in (3) above. Since at most $\sqrt{n}$ large free blossoms can be present at any given time, an insertion into S_2 costs $O(\sqrt{n}(\log n)^2)$ operations.

From Lemma 4 (§ 4.2), it follows that the number of times blossoms become free or get labelled is $O(n)$, and the number of insertions into S_2 is also $O(n)$. So from (a)-(c) and the bound on the time for recomputing the edges in (1) and (2) above, we may conclude that the time for maintaining δ_{11} is $O(n^{1.5}(\log n)^2)$ per phase.

4.4.2. Maintaining δ_{12}. In this case too S is partitioned into S_1 and S_2 such that $|S_2| \leqq \sqrt{n}$. The free vertices in the small free blossoms are partitioned into $F_1, F_2, \cdots, F_{\lceil n^{0.5} \rceil}$ (some of the F_i's could be empty) such that for each small free blossom B

$$F_i \cap B \neq \phi \Rightarrow B \subseteq F_i, \qquad i = 1, \cdots, \lceil n^{0.5} \rceil,$$

and

$$|F_i| \leqq 2(\sqrt{n}+1), \qquad i = 1, \cdots, \lceil n^{0.5} \rceil.$$

We maintain the following edges and data structures:

(1) For each vertex v_k that is in a small free blossom, the edge $shortest[v_k, S_1]$. All these edges are in a priority queue with the priority of an edge being given by its slack.

(2) For each vertex $v_j \in S_2$, the edges $shortest[v_j, F_i]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. All these edges are also in a priority queue with the priority of an edge being given by its slack.

(3) The weighted Voronoi diagram/range tree (WVD/RT) for S_1 and for each F_i, $1 \leqq i \leqq \sqrt{n}$. For each vertex v_k in a small free blossom, $\alpha_k = w(v_k)$, and for each vertex $v_j \in S$, $\alpha_j = w(v_j) + \Delta$. Thus if F' is a subset of the set of vertices in small free blossoms and $S' \subseteq S$, then

$$slack[shortest[F', S']] = \min_{v_k \in F', v_j \in S'} \{slack[(v_k, v_j)]\}.$$

Hence δ_{12} is achieved either for the edge in (1) above with minimum priority or for the edge in (2) above with minimum priority, and can be computed in $O(\log n)$ time by examining these edges.

A new s-vertex is always inserted into S_2. Whenever the size of S_2 reaches the threshold of $\sqrt{n}$, all the vertices in S_2 are added to S_1 and S_2 is reset to the null set, and the edges in (1) above are recomputed. By Lemma 1 (§ 2.1) the recomputation may be done in $O(n \log n)$ time using the WVD/RT for S_1. So the time for recomputation is $O(n^{1.5} \log n)$ per phase.

Next, we show that the time for updating the edges in (1) and (2) above when a small blossom becomes free or a small free blossom gets labelled or a vertex is inserted in S_2 is $O(\sqrt{n} \log n)$.

(a) Suppose a small blossom B becomes free. Let F_i be such that $|F_i| \leqq \sqrt{n}$. (By the pigeonhole principle, there always exists such an F_i.) All the vertices in B are added to F_i, the WVD/RT for F_i is recomputed, and then for each $v_j \in S_2$, $shortest[v_j, F_i]$ is recomputed in $O(\log n)$ time by querying the WVD/RT for F_i. For each vertex $v_k \in B$, the edge $shortest[v_k, S_1]$ is found in $O(\log n)$ time by querying the WVD/RT for S_1. So when a small blossom B becomes free it costs $O((\sqrt{n} + |B|) \log n) = O(\sqrt{n} \log n)$ operations.

(b) Suppose a small free blossom B gets a t-label/s-label. First, the $O(\sqrt{n})$ edges in the above priority queues that are incident to vertices in B are deleted. Let $B \subseteq F_i$. All the vertices in B get deleted from F_i, the WVD/RT for F_i is recomputed, and then for each $v_j \in S_2$, $shortest[v_j, F_i]$ is recomputed by querying the WVD/RT for F_i. So when a small free blossom gets labelled it costs $O(\sqrt{n} \log n)$ operations.

(c) Suppose a vertex v_j is inserted into S_2. Then we have to compute $shortest[v_j, F_i]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. So the cost in this case is also $O(\sqrt{n} \log n)$ operations.

From Lemma 4 (§ 4.2) it follows that the number of times blossoms become free or get labelled is $O(n)$, and the number of insertions into S_2 is also $O(n)$. From (a)-(c), and the time bound for recomputing the edges in (1) above, we may then conclude that the time for maintaining δ_{12} is $O(n^{1.5} \log n)$.

4.5. Maintaining δ_2 during a phase. In this section we will describe how to maintain δ_2, and an associated edge (v_i, v_j) for which δ_2 is achieved, where v_i, v_j, are s-vertices in distinct s-blossoms. As before, the slack associated with an edge (v_i, v_j), is given by the quantity $d(v_i, v_j) - \alpha_i - \alpha_j$, and denoted by $slack[(v_i, v_j)]$. Note that for each $v_i \in S$, $\alpha_i = w(v_i) + \Delta$. Thus if $S' \subseteq S$ and $S'' \subseteq S$, then

$$slack[shortest[S', S'']] = \min_{v_i \in S', v_j \in S''} \{slack[(v_i, v_j)]\}.$$

So we can use the weights rather the dual variables in maintaining δ_2. Let

$$\delta_{21} = \min_{v_i, v_j \text{ are in distinct large } s\text{-blossoms}} \{slack[(v_i, v_j)]\},$$

$$\delta_{22} = \min_{\substack{v_i \text{ is in a small } s\text{-blossom} \\ v_j \text{ is in a large } s\text{-blossom}}} \{slack[(v_i, v_j)]\},$$

and

$$\delta_{23} = \min_{v_i, v_j \text{ are in distinct small } s\text{-blossoms}} \{slack[(v_i, v_j)]\}.$$

Clearly,

$$\delta_2 = \tfrac{1}{2} \min \{\delta_{21}, \delta_{22}, \delta_{23}\}.$$

In §§ 4.5.1–4.5.3, we describe how to maintain δ_{21}, δ_{22}, and δ_{23}, in $O(n^{1.5}(\log n)^2)$, $O(n^{1.5} \log n)$, and $O(n^{1.5}(\log n)^4)$, time per phase, respectively.

Consider a specific phase. Let B be a blossom that has an s-label sometime during the phase. An s-subblossom (t-subblossom) of B is a subblossom of B that has an s-label (t-label), sometime during the phase, prior to B getting its s-label. We require the following easily proven lemma.

LEMMA 5. *Consider the s-blossoms that are present at the end of a particular phase. With each such s-blossom $\hat{B}$ is associated a tree whose root is $\hat{B}$ itself, and the leaves are all the vertices in $\hat{B}$. Each nonleaf node in this tree is a blossom that had an s-label sometime during the phase. The sons of a nonleaf node (blossom) B are as follows. Each son of B that is not a leaf is an s-subblossom of B. Each son of B that is a leaf is a vertex in B that has switched status from free vertex/t-vertex to s-vertex as a result of B getting an s-label. Let τ denote the forest of such trees associated with the s-blossoms at the end of the phase. Note that for a blossom (node) B in forest τ, the number of vertices in B (i.e., $|B|$) equals the number of leaves in the subtree rooted at B.*

For each node B in the forest τ, define $\sigma(B)$ as follows:

(1) *For a leaf B, let $\sigma(B) = 1$.*

(2) *Let B be a nonleaf node. Consider vertices to be trivial blossoms of size 1. Let B' be a son of B such that B' contains the largest number of vertices among all the sons of B. We let $\sigma(B) = |B| - |B'|$.*

Then

$$\sum_{B \text{ a node in forest } \tau} \sigma(B) = O(n \log n). \qquad \square$$

4.5.1. Maintaining δ_{21}. For each pair $B, \hat{B}$, where B and $\hat{B}$ are distinct large s-blossoms, we maintain the edge $shortest[B, \hat{B}]$. The edges are in a priority queue, with the priority of an edge (v_i, v_j) being $slack[(v_i, v_j)]$. δ_{21} is achieved for an edge with the minimum priority in this priority queue. We also maintain a semidynamic weighted Voronoi diagram/range tree (WVD/RT) described in § 2.3 for each large s-blossom. Note that once the semidynamic WVD/RT for a blossom B is available, $shortest[v_i, B]$ is computable in $O(\log n)^2)$ time for any vertex v_i. The shortest edges between large s-blossoms are updated as follows:

(1) Suppose a large blossom B gets an s-label in Case 1 or Case 3 in § 4.2. Then each vertex in B switches status to s-vertex. For each large s-blossom $\hat{B}$ (other than B), we compute the edge $shortest[v_i, \hat{B}]$ for all vertices v_i in B by querying the WVD/RT for $\hat{B}$, and use these edges to find $shortest[B, \hat{B}]$. We also build the semidynamic WVD/RT for B. As the number of large s-blossoms present at any time is $O(\sqrt{n})$, we spend $O(|B|\sqrt{n}(\log n)^2)$ time when B gets an s-label.

(2) Suppose a new large s-blossom B is created in Case 2 in § 4.2. Let C be the set of all vertices v in B such that v is either in a t-subblossom of B or in a

small s-subblossom of B. Note that each subblossom of B is either a t-subblossom or an s-subblossom, and that each vertex in each t-subblossom of B switches status to s-vertex when B receives an s-label.

(2.1) For each large s-subblossom B' of B, we delete from the above priority queue the $O(\sqrt{n})$ edges which are incident to vertices in B'.

(2.2) Let $\hat{B}$ be a large s-blossom other than B. For each vertex v_i in C, the edge *shortest*$[v_i, \hat{B}]$ is found by querying the semidynamic WVD/RT for $\hat{B}$. Using these edges together with the edges *shortest*$[B', \hat{B}]$, where B' are the large s-subblossoms of B, *shortest*$[B, \hat{B}]$ is obtained, and inserted into the above priority queue. Since at most $O(\sqrt{n})$ large s-blossoms may be simultaneously present and since the number of subblossoms of B is $O(|C|)$, the time to compute *shortest*$[B, \hat{B}]$ for all large s-blossoms $\hat{B}$ is $O(|C|\sqrt{n}(\log n)^2)$.

(2.3) Let B' be an s-subblossom of B such that B' has the greatest size among all the s-subblossoms of B.

(2.3.1) If B' is a small blossom, then $B = C$ and the semidynamic WVD/RT for B is constructed in $O(|C|(\log n)^2)$ time.

(2.3.2) If B' is a large blossom, then the semidynamic WVD/RT for B is obtained by inserting all the vertices in $B - B'$ into the semidynamic WVD/RT for B'. So there are $\sigma(B)$ insertions into a semidynamic WVD/RT where $\sigma(B)$ is as defined in Lemma 5.

A upper bound of $O(n^{1.5}(\log n)^2)$ on the time per phase for the computations in (1), (2.1), and (2.2) follows from the following observations. First, the total number of status switches of vertices to s-vertices is $O(n)$ per phase. Second, the number of blossoms labelled during a phase is $O(n)$, and so the number of s-subblossoms generated during a phase is $O(n)$. Third, for each vertex v, the condition that v is in a small s-subblossom of a large s-blossom can occur at most once during a phase. From these observations it also follows that the time spent in (2.3.1) is $O(n(\log n)^2)$ per phase. By Lemma 5, the total number of insertions into semidynamic WVD/RT's in (2.3.2) is $O(n \log n)$ per phase, and so the total time per phase spent in (2.3.2) is $O(n(\log n)^3)$ at the average rate of at most $O((\log n)^2)$ per insertion. Thus the time for maintaining δ_{21} is $O(n^{1.5}(\log n)^2)$ per phase.

4.5.2. Maintaining δ_{22}. The procedure for maintaining δ_{22} is similar to the one for maintaining the minimum slack δ in the bipartite case discussed in § 3. Let S_L denote the set of those vertices in S that are in large s-blossoms, and let S_M denote the set of those vertices in S that are in small s-blossoms. S_L is partitioned into S_{L1}, S_{L2}, such that $|S_{L2}| \leqq \sqrt{n}$. S_M is partitioned into $S_{M1}, S_{M2}, \cdots, S_{Mr}$, $r = \lceil n^{0.5} \rceil$, such that $|S_{Mi}| \leqq 2(\lceil n^{0.5} \rceil + 1)$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. (Some of the S_{M_i}'s could possibly be empty.) The following information is maintained:

(i) For each $v_i \in S_M$, the edge *shortest*$[v_i, S_{L1}]$. These edges are in a priority queue, with the priority of an edge being its slack.

(ii) For each $v_j \in S_{L2}$, the edges *shortest*$[v_j, S_{Mi}]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$. These edges too are in a priority queue with the priority of an edge being given by its slack.

(iii) The WVD/RT for S_{L1} and for each S_{Mi}, $1 \leqq i \leqq \lceil n^{0.5} \rceil$.

Clearly, δ_{22} is achieved for either the edge in (i) with minimum priority or the edge in (ii) with minimum priority.

Initially, S_{L1} contains all the vertices in the large exposed blossoms, and $S_{L2} = \phi$. When a new vertex is added to S_L, it is always inserted into S_{L2}. When the size of S_{L2} reaches the threshold of $\lceil n^{0.5} \rceil$, all the vertices in S_{L2} are moved over to S_{L1}, and the

edges in (i) are recomputed. By Lemma 1 (§ 2.1), using the WVD/RT for S_{L1} the recomputation may be performed in $O(n \log n)$ time, leading to a time of $O(n^{1.5} \log n)$ per phase for the recomputation.

Suppose there is an insertion into S_M, say v_i is inserted into S_{M_i}. Then the WVD/RT for S_{Mi} and the edges *shortest*$[v_j, S_{Mi}]$ for the vertices v_j in S_{L2} must be recomputed, and *shortest*$[v_i, S_{L1}]$ must be computed and inserted into the priority queue in (i). So an insertion into S_M costs $O(n^{0.5} \log n)$ time. Similarly, a deletion from S_M costs $O(n^{0.5} \log n)$ time. Insertion of a vertex v_j into S_{L2} also costs $O(n^{0.5} \log n)$ time, as the edges *shortest*$[v_j, S_{Mi}]$, $1 \leqq i \leqq \lceil n^{0.5} \rceil$ have to be computed. (Note that there are no deletions from S_L.) The number of insertions into S_L and S_M is $O(n)$ per phase. So the time for all the insertions into S_{L2}, and all the insertions into and deletions from S_M, is $O(n^{1.5} \log n)$ per phase.

Thus the time for maintaining δ_{22} is $O(n^{1.5} \log n)$ per phase.

4.5.3. Maintaining δ_{23}. Next, we show how to maintain δ_{23}. Let the vertices in S be ordered by the following rule. For $v_i, v_j \in S$, $v_i < v_j$ if and only if v_i was added to S before v_j. Note that for an s-blossom B, the ordering on the vertices in S induces a partition of $S - B$ into at most $|B|+1$ intervals. Let $u_1 < u_2 < \cdots < u_{|S|}$ denote the ordered sequence of the vertices in S. By Lemma 3 (§ 2.3), we can maintain a semidynamic data structure for S (that is a dynamized version of the data structure for Problem 2 in § 2) such that:

(I) The total cost of inserting all the points in S, and thus the total time for maintaining the data structure for an entire phase is $O(n(\log n)^3)$.

(II) Given an interval $[u_i, u_j)$, $1 \leqq i < j \leqq |S|+1$, and a vertex v_k, *shortest*$[v_k, [u_i, u_j)]$ may be computed in $O((\log n)^3)$ time.

For each small s-blossom B the following edges incident on the vertices in B are maintained. Let $|B| = r$, and let $u_{i_1} < u_{i_2} < \cdots < u_{i_r}$ be the ordered sequence of the vertices in the small s-blossom B. Then $S - B$ can be expressed as the disjoint union of the intervals $[u_1, u_{i_1})$, $[u_{i_1+1}, u_{i_2}), \cdots, [u_{i_{r-1}+1}, u_{i_r})$, $[u_{i_r+1}, u_{|S|+1})$. (Some of the intervals could possibly be empty.) We shall maintain the edge *shortest*$[B, [u_1, u_{i_1})]$, and the edges *shortest*$[B, [u_{i_k+1}, u_{i_{k+1}})]$, $1 \leqq k < |B|$. (Note that *shortest*$[B, [u_{i_r+1}, u_{|S|+1})]$ is not included.)

All these edges corresponding to the small s-blossoms are placed in a priority queue, with the priority of an edge given by its slack. It is easily seen that if (u_i, u_j) is an edge in this priority queue with the minimum priority then *slack*$[(u_i, u_j)] \leqq \delta_{23}$.

The computations required to maintain the above edges associated with small s-blossoms are broken down into two cases:

(a) An existing small blossom B gets an s-label. This happens in Case 1 and Case 3 in § 4.2, and the vertices in the newly labelled blossom B switch status from free vertices/t-vertices to s-vertices. All the vertices in B get added to S and so B corresponds to a single interval in the ordering on S. Moreover, $S - B$ is also an interval. For each vertex $u_{i_k} \in B$, *shortest*$[u_{i_k}, S - B]$ is found in $O((\log n)^3)$ time by querying the data structure for S, and using these edges *shortest*$[B, S - B]$ is obtained in $O(|B|)$ extra time. So the time spent is $O(|B|(\log n)^3)$.

(b) A new small blossom B is discovered and gets an s-label. This happens in Case 2 in § 4.2. Let B' be an s-subblossom of B such that B' has the largest size among all the s-subblossoms of B. Let $|B| = r$, and let $|B'| = m$. Let $u_{i_1} < u_{i_2} < \cdots < u_{i_r}$ be the vertices in B, and let $u_{j_1} < u_{j_2} < \cdots < u_{j_m}$ be the vertices in B'. Let J be the set of intervals defined by

$$J = \{[u_1, u_{i_1}), [u_{i_1+1}, u_{i_2}), \cdots, [u_{i_{r-1}+1}, u_{i_r})\},$$

and J' be the set of intervals defined by

$$J' = \{[u_1, u_{j_1}), [u_{j_1+1}, u_{j_2}), \cdots, [u_{j_{m-1}+1}, u_{j_m})\}.$$

Note that $(V-B) \cap [u_1, u_{i_r})$ can be expressed as the disjoint union of all the intervals in J, and that $(V-B') \cap [u_1, u_{j_m})$ can be expressed as the disjoint union of all the intervals in J'. Furthermore, $|J \cap J'| \geqq |B'| - (|B| - |B'|)$ and so $|J - J'| \leqq 2(|B| - |B'|)$. The edges *shortest*$[B, I]$ for all the intervals I in J may be obtained in $O(|B|(|B| - |B'|) \times (\log n)^3)$ time as follows. For each interval I' in $J - J'$, we compute *shortest*$[u', I']$ for all vertices u' in B'. For each interval I in J, we compute *shortest*$[u, I]$ for all vertices u in $B - B'$. Each edge is obtained by querying the data structure for S (described in § 2.3) in $O((\log n)^3)$ time, and all these edges may be obtained in $O(|B|(|B| - |B'|) \times (\log n)^3)$ time. Then as the edges *shortest*$[B', I']$, $I' \in J \cap J'$, are already available, the edges *shortest*$[B, I]$, $I \in J$, can be found in $O(|B|(|B| - |B'|) \log n)$ additional time. The time per phase for the computations in (a) above is $O(n(\log n)^3)$, since all the blossoms labelled by an s-label in Case 1 and Case 3 in § 4.2 are disjoint. As the size of a small s-blossom is at most $\sqrt{n}$, $O((|B| - |B'|)\sqrt{n}(\log n)^3)$ time is spent in (b) per each new blossom B, where B' is an s-subblossom of B that contains the largest number of vertices among all the s-subblossoms of B. In other words, $O(\sigma(B)\sqrt{n}(\log n)^3)$ time is spent in (b) above per each new s-blossom B, where $\sigma(B)$ is as defined in Lemma 5. Then we may apply Lemma 5 and conclude that the time per phase spent in performing the computations in (b) above is $O(n^{1.5}(\log n)^4)$. Thus the time per phase for maintaining δ_{23} is $O(n^{1.5}(\log n)^4)$.

5. Conclusion. We have shown that the underlying geometry can be exploited to speed up algorithms for weighted matching when the vertices of the graph are points on the plane, and the weight of an edge between two points is the distance between the points under some metric. The techniques described in the paper can be used to speed up algorithms for related problems such as bottleneck matching [11] for points on the plane, and the transportation problem [11], [14] where the sources and the sinks are located on the plane and the cost of transporting from a source to a sink is proportional to the distance between the source and the sink. The techniques in the paper can also be utilized to speed up scaling algorithms [7], [18] for matching and related problems by a factor of about $\sqrt{n}$ for points on the plane. Finally, we note that for the L_1 and the L_∞ metrics the algorithms in the paper easily extend to the case where the vertices of the graph are points in d-dimensional space (d fixed) rather than points on the plane. For points in d-dimensional space we use d-dimensional range trees instead of two-dimensional range trees [15], and this increases the running time of the matching algorithms by at most $O(\log n)^d$.

Acknowledgments. The author thanks F. P. Preparata for suggesting the problem of matching on the plane, P. Shor, S. Fortune, and H. Edelsbrunner for helpful discussions, and M. R. Garey and D. S. Johnson for useful comments and suggestions concerning the presentation of this paper.

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] D. AVIS, *A survey of heuristics for the weighted matching problem*, Networks, 13 (1983), pp. 475-493.

[3] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone subdivision*, Tech. Report, DEC Systems Research Center, Palo Alto, CA, 1984.

[4] J. EDMONDS, *Maximum matching and a polyhedron with* 0, 1*-vertices*, J. Res. Nat. Bur. Standards, 69B (1965), pp. 125–130.

[5] S. FORTUNE, *A sweepline algorithm for Voronoi diagrams*, in Proc. ACM Annual Symposium Computational Geometry, Association for Computing Machinery, New York, 1986, pp. 313–322.

[6] H. N. GABOW, *An efficient implementation of Edmond's algorithm for maximum matching on graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 221–234.

[7] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for network problems*, Tech. Report, Department of Computer Science, Princeton University, Princeton, NJ, 1987.

[8] Z. GALIL, S. MICALI, AND H. N. GABOW, *Priority queues with variable priority and an* $O(EV \log V)$ *algorithm for finding a maximal weighted matching in general graphs*, in Proc. 22nd Annual IEEE Symposium Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1982, pp. 255–261.

[9] M. IRI, M. MUROTA, AND S. MATSUI, *Linear time heuristics for minimum-weight perfect matching on a plane with application to the plotter problem*, unpublished manuscript.

[10] H. W. KUHN, *The Hungarian method for the assignment problem*, Naval Res. Logist. Quart., 2 (1955), pp. 83–97.

[11] E. LAWLER, *Combinatorial Optimization*: *Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[12] 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.

[13] K. MEHLHORN, *Data Structures and Algorithms* 3: *Multidimensional Searching and Computational Geometry*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, New York, pp. 2–9, 1984.

[14] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization*: *Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[15] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*: *An Introduction*, Springer-Verlag, Berlin, New York, 1985.

[16] M. SHARIR, *Intersection and closest-pair problems for a set of planar discs*, SIAM J. Comput., 14 (1985), pp. 448–468.

[17] K. J. SUPOWIT AND E. M. REINGOLD, *Divide-and-conquer heuristics for minimum weighted Euclidean matching*, SIAM J. Comput., 12 (1983), pp. 118–144.

[18] H. N. GABOW AND R. E. TARJAN, *Faster scaling algorithms for graph matching*, Tech. Report, Department of Computer Science, Princeton University, Princeton, NJ, 1987.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1226–1244, December 1989

010

ON RELAXED SQUASHED EMBEDDING OF GRAPHS INTO A HYPERCUBE*

MING-SYAN CHEN† AND KANG G. SHIN‡

Abstract. Task allocation in an n-dimensional hypercube (or an n-cube) multicomputer consists of two sequential steps: (i) determination of the size of the cube required to accommodate an incoming task composed of a set of interacting modules, and (ii) allocation of the task to a cube of the dimension determined from (i). Step (i) is usually done manually by the users, which is often difficult and leads to the underutilization of processors in an n-cube system. The main objective here is to automate step (i). Step (ii) has already been addressed in [*IEEE Trans. Comput.*, 36 (1987), pp. 1396–1407].

Each incoming task is represented by a graph in which each node denotes a module of the task and each link represents the need of intermodule communication. Each module must be assigned to a subcube in such a way that node adjacencies in the associated task graph are preserved. This assignment problem is called the *relaxed squashed* (RS) *embedding* of a graph, and the minimal dimension of a cube required for a given graph is termed the *weak cubical dimension* of the graph. Some mathematical properties of the RS embedding are derived first. In light of these mathematical properties, fast algorithms are developed to RS embed task graphs. A heuristic function for the A^* search algorithm is also derived to determine the weak cubical dimension of a graph.

Key words. n-cube, loop switching addressing scheme, squashed embedding, weak cubical dimension, heuristic search

AMS(MOS) subject classifications. 05C10, 06E15, 14E25

1. Introduction.

Recently, hypercube multicomputers are beginning to spread widely in the research and development community as well as in commercial markets [Cor85], [Sei85], [Val82], [Wil87]. To execute a task in an n-dimensional hypercube (or n-cube) multicomputer, the task is usually decomposed into a set of interacting modules that are then assigned to a subcube. Thus, task allocation in an n-cube multicomputer system consists of two sequential steps: (i) determination of the dimension of the subcube required to accommodate all the modules of each incoming task, and (ii) allocation of each task to a subcube of the dimension determined from (i) in the hypercube multicomputer. As an efficient solution to (ii), we propose a first-fit linear search for required subcubes whose addresses are represented by the binary reflected Gray code [ChS87]. Conventionally, (i) is determined manually by the users, which is often very difficult and results in the underutilization of processors and degradation of system performance. The automation of step (i) is thus very important and will be the focus of this paper.

Each incoming task is described by a graph (called *task graph*), in which each node denotes a module of the task and each link represents the need of intermodule communication. We want to determine a subcube in the n-cube system that can accommodate the incoming task subject to some constraints. Note that different computing systems and user environments may require different criteria to be used for

* Received by the editors June 13, 1988; accepted for publication (in revised form) January 13, 1989. This work was partially supported by the Office of Naval Research under contract N00014-85-K-0122. This research was performed when M.-S. Chen was at the Real-Time Computing Laboratory, University of Michigan, Ann Arbor, Michigan.

† IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.

‡ Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109-2122.

the determination of the subcube size required for each task. Several important results in various types of embedding have been reported [Fir65], [GaG75], [GrP72], [Har86], [Har80], [HaM72]. Basically, *isomorphic embedding* is a node-to-node adjacency-preserving mapping [GaG75], [HaM72], whereas *isometric embedding* is a node-to-node distance-preserving mapping [Fir65]. In *homeomorphic embedding*, additional nodes are allowed to be inserted into edges so as to make the graph isomorphically embeddable into a cube [Har86]. *Squashed embedding* is a node-to-subcube distance-preserving mapping [GrP72].

In this paper, we propose and investigate a new type of embedding, called *relaxed squashed* (RS) *embedding*, a node-to-subcube adjacency-preserving mapping. In other words, adjacent modules in the task graph are assigned to adjacent subcubes. The dimension of the minimal cube required for the RS embedding of a given graph will henceforth be called the *weak cubical dimension* of the graph. Clearly, the problem of determining the weak cubical dimension of a task graph is similar to the squashed embedding problem [BGK72], [GrP71], [GrP72], [Yao78], [Win83] in the sense that each node in the source graph is mapped into a subcube. But, it differs from the squashed embedding problem in that only adjacency, rather than internode distance, must be preserved under the mapping. For example, the embedding from a path P_4 into a Q_2 in Fig. 1 preserves adjacency, but not distance.

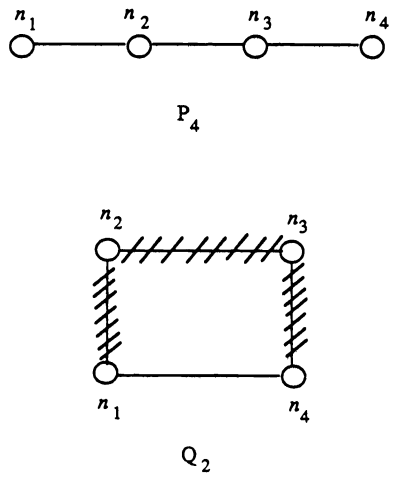


FIG. 1. *A mapping that preserves adjacency but not distance.*

From the result of the squashed embedding problem [Win83], we know that every graph has its weak cubical dimension, although the cubical dimension is defined only for cubical graphs [Har69]. Similarly to the determination of the cubical dimension of cubical graphs [KVC85] [CKV87], we shall prove that the problem of determining the existence of an RS embedding from a graph to a cube of a given dimension is NP-complete. This proof justifies the need of our heuristic approaches to the RS embedding problem. Some mathematical properties for the RS embedding problem will be derived first. Then, using these results, we shall develop (a) fast algorithms for the RS embedding of a given task graph and (b) a heuristic function for the A^* search algorithm to determine if a graph can be RS embedded into a cube of a given dimension.

By applying this search algorithm for different dimensions repeatedly, we can determine the weak cubical dimension of a graph.

This paper is organized as follows. The definitions and notation necessary for our discussion are given in § 2 where related topics and results are also reviewed. Section 3 deals with the mathematical properties of the RS embedding. Using these properties, fast algorithms for the RS embedding are then developed in § 4. A heuristic search algorithm to determine if a graph can be RS embedded into a cube of a given dimension is proposed. Illustrative examples are presented in § 5, and the paper concludes with § 6.

2. Preliminaries.

2.1. Notation and definitions. Denote an undirected graph by $G_A = (V_A, E_A)$, where V_A and E_A are the set of nodes and the set of links in G_A, respectively, and use $\bar{G}$ to denote the *complement* of a graph G [Har69]. For two graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$, G_B is a *subgraph* of G_A if $V_B \subseteq V_A$ and $E_B \subseteq E_A$. An *induced subgraph* of G_A with a node set $V_S \subseteq V_A$ is the maximal subgraph of G_A with the node set V_S.

An edge in a connected graph is called a *bridge* if its removal disconnects the graph. Clearly, the removal of an edge from a tree will result in two trees, called the *attached trees* of the edge. The number of nodes in the larger[1] of the two attached trees of an edge is called the *weight* of the edge. The *centroid edge* of a tree is defined as the edge with the minimal weight. Besides, the graph operations, $\times$ (product), $\cup$ (union) and $+$ (join) [Har69] will be used to facilitate our presentation. Note that while the union operation may be applied on two graphs that are not disjoint, the join operation is applied only on two disjoint graphs. An illustrative example of the above operations is given in Fig. 2. An n-cube can now be defined as $Q_n = K_2 \times Q_{n-1}$, for all $n \geqq 1$, where K_2 is the complete graph with two nodes and Q_0 is a trivial graph with one node.

Let Σ be the ternary symbol set $\{0, 1, *\}$, where $*$ is the *don't care* symbol. Then, every subcube of an n-cube can be uniquely represented by a sequence of ternary symbols, called the *address* of the subcube. Also, let $|q|$ denote the dimension of the subcube q. The distance between two subcubes is then defined as follows.

DEFINITION 1. The Hamming distance, $H: \sum^n \times \sum^n \to I^+$, between two subcubes with addresses $\alpha = a_n a_{n-1} \cdots a_1$ and $\beta = b_n b_{n-1} \cdots b_1$ in a Q_n is defined as $H(\alpha, \beta) = \sum_{i=1}^{n} h(a_i, b_i)$, where

$$h(a, b) = \begin{cases} 1 & \text{if } [a = 0 \text{ and } b = 1] \text{ or } [a = 1 \text{ and } b = 0], \\ 0 & \text{otherwise.} \end{cases}$$

A subcube $\alpha = a_n a_{n-1} \cdots a_1$ is said to *contain* another subcube $\beta = b_n b_{n-1} \cdots b_1$, denoted by $\beta \subseteq \alpha$, if and only if all the nodes in β belong to α. The notation $\beta \subset \alpha$ is used to denote the case when $\beta \subseteq \alpha$ and $\beta \neq \alpha$. The *minimal upper subcube* of two subcubes α and β, denoted by $\text{lcm}(\alpha, \beta)$, is then defined as the smallest subcube among all those subcubes which contain both α and β. Similarly, the *maximal lower subcube* of two subcubes α and β, denoted by $\gcd(\alpha, \beta)$, is the largest subcube among all those subcubes contained in both α and β. For notational convenience, we let $\gcd(\alpha, \beta) = \varnothing$ if $H(\alpha, \beta) \geqq 1$. For example, $H(00*1*, 1000*) = 2$, $\text{lcm}(*100, 0110) = *1*0$, and $\gcd(01**, *10*) = 010*$. Also, let $D(n_i)$ denote the address of the subcube assigned to module n_i and $B(n_i)$ denote the set of nodes adjacent to n_i in the graph. For the graph of Fig. 3, we get $B(n_1) = \{n_2, n_3, n_4, n_6\}$.

[1] One tree is said to be *larger* than the other if it contains more nodes.

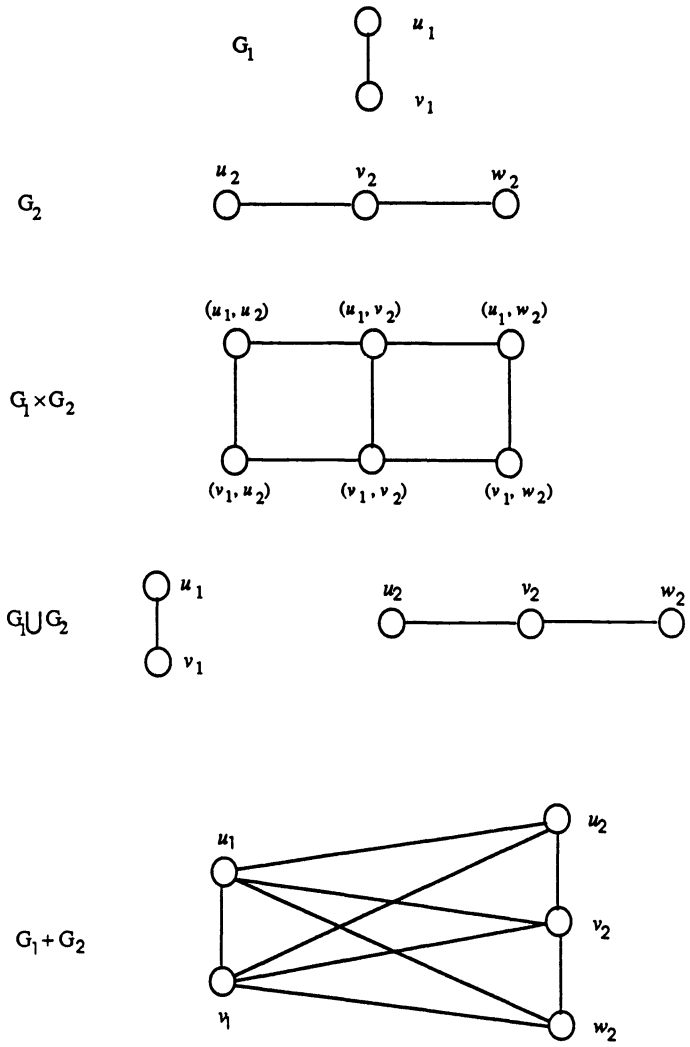


FIG. 2. *The product, union, and join operations on graphs.*

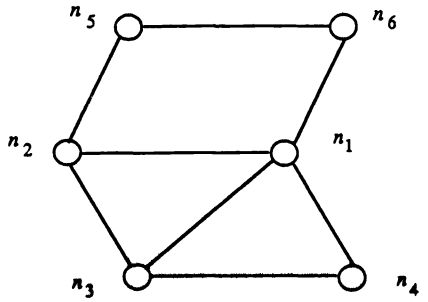


FIG. 3. *An example task graph.*

2.2. Previous related results. In [GrP71], an interesting addressing scheme for loop switching networks [Pie72] has been proposed. In this scheme, a loop switching network is represented by a graph in which nodes and links represent the loops and the contact points between loops, respectively. The problem is to find an addressing scheme in which each node is assigned a sequence of ternary symbols to correctly represent distances between nodes in the graph. More formally, this problem can be stated as follows. Given a connected graph G with n nodes, find the least integer $N(G)$ with which it is possible to assign each node v in G an address $D(v) \in \Sigma^{N(G)}$ such that $d_G(v_1, v_2) = H(D(v_1), D(v_2))$, for all $v_1, v_2 \in V_G$, where $d_G(v_1, v_2)$ is the distance between v_1 and v_2 in G, and V_G is the set of nodes in G. Naturally, the following two questions arise. (1) Does there always exist such an addressing scheme for an arbitrary network G with n nodes? (2) If the answer to (1) is yes, what is the least number $N(G)$ of ternary symbols that suffices to implement the addressing scheme for G? This problem was studied for more than a decade [BGK72], [GrP71], [GrP72], [Yao78] until an important conjecture $N(G) \leqq n-1$ was proved in [Win83]. Thus, questions (1) and (2) have been answered.

As pointed out in [GrP72], this problem is equivalent to the *squashed embedding* problem. Embed a task graph into a cube in such a way that each node of the graph is assigned to a subcube while preserving internode distances. Fig. 4 shows an example of the squashed embedding, where $D(v_1) = *11$, $D(v_2) = 110$, $D(v_3) = 010$, and $D(v_4) = 000$.

When task allocation in a hypercube multicomputer is considered, it is more important to preserve node adjacencies than internode distances, since node adjacencies are directly related to intermodule communication delays. Based on this observation, we shall consider the problem of embedding a given task graph into a hypercube in such a way that each task module must be assigned to a subcube while preserving task module adjacencies. This problem can be viewed as a relaxed version of the squashed

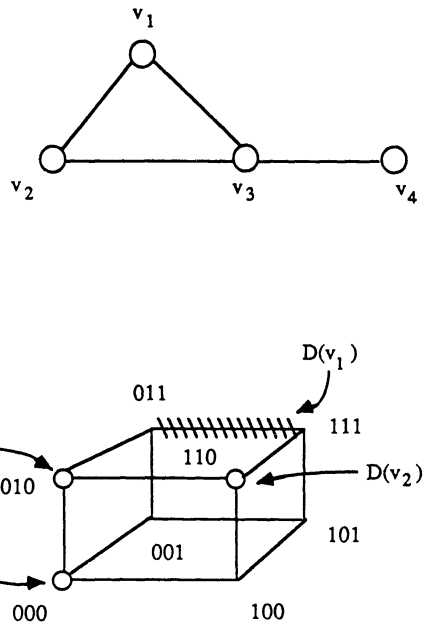


FIG. 4. *An example of squashed embedding.*

embedding problem since it preserves node adjacencies instead of internode distances. (Henceforth it will be called the *relaxed squashed* (RS) *embedding.*) Obviously, a graph can be RS embedded in any cube of size greater than or equal to its weak cubical dimension. Insofar as system utilization is concerned, however, we want to find a minimal cube for the RS embedding of each task graph. Several important properties of the RS embedding problem will be derived in the following section. We shall prove first that the problem of determining the existence of an RS-embedding for a given task graph is NP-complete, and then derive some mathematical properties of the RS embedding. These properties will be applied to develop our heuristic solutions.

3. Mathematical properties of RS-embedding.

THEOREM 1. *The problem of determining if a graph can be* RS *embedded into a cube of a given dimension is* NP*-complete.*

Proof. Suppose k is the dimension of the cube into which a source graph is to be RS embedded. Consider the instance that the source graph contains 2^k nodes. Clearly, the source graph can be RS embedded into a Q_k if and only if it can be isomorphically embedded into a Q_k. However, the problem of determining whether a graph of 2^k nodes can be isomorphically embedded into a Q_k has already been proved to be NP-complete in [CKV87], meaning that the problem of determining whether a graph of 2^k nodes can be RS embedded into a Q_k is also NP-complete. This theorem is thus proved by restriction [GaJ79]. □

Theorem 1 justifies the need of heuristic solution approaches to the RS embedding problem. It is necessary to develop some mathematical properties of the RS embedding problem, on which these heuristic approaches will be based. The following theorem about the squashed embedding has been proved in [GrP72].

THEOREM 2 [GrP72]. $N(K_n) = n-1$, *where* K_n *is a complete graph with n nodes.*

Note that when the graph to be embedded is a complete graph, the requirement of preserving distance is the same as the adjacency requirement. This fact is described by the following corollary.

COROLLARY 2.1. *Let wd* (G) *be the weak cubical dimension of G. Then, wd* $(K_n) = n-1$.

Consider the case when G_1 is a subgraph of G_2. Clearly, we have less restirction in the RS embedding of G_1 than that of G_2. This leads to the following proposition.

PROPOSITION 1. *If* G_1 *is a subgraph of* G_2, *then* $\mathrm{wd}(G_1) \leqq \mathrm{wd}(G_2)$.

Since the number of nodes in the n-cube must be greater than or equal to that of the task graph to be embedded, we have the following corollary.

COROLLARY 2.2. *Let G be a graph with n nodes. Then,* $\lceil \log_2 n \rceil \leqq \mathrm{wd}(G) \leqq n-1$.

Note that Corollary 2.2 provides loose bounds for the weak cubical dimension of a graph with n nodes.

THEOREM 3. *Let* $G=(V, E)$ *be a connected graph and let* $G_S=(V_S, E_S)$ *be a subgraph of G. Suppose the induced subgraph of G with the node set* V_S, *denoted by* $\mathrm{ind}_G(V_S)$, *can be* RS *embedded into a* Q_m, *and the removal of all edges in* E_S *from G results in* $|V_S|$ *disjoint graphs,* $G_i=(V_i, E_i)$, $1 \leqq i \leqq |V_S|$. *Then,* $\mathrm{wd}(G) \leqq \max_{1 \leqq i \leqq |V_S|} \{\mathrm{wd}(G_i)\} + m$.

Proof. Let u_i, $1 \leqq i \leqq |V_S| = k$, be the nodes in $G_S \cap G_i$. Since $\mathrm{ind}_G(V_S)$ can be RS embedded into a Q_m and $G_S \subseteq \mathrm{ind}_G(V_S)$, there exists an addressing scheme for the RS embedding of G_S into the Q_m. Let $D_{G_S}(u_i)$ denote the address of $u_i \in V_S \cap V_i$ in the addressing scheme.

Partition V into k disjoint node sets V_i, $1 \leqq i \leqq k$. For $1 \leqq i \leqq k$, let $D_{G_i}(v)$ denote the address of $v \in V_i$ for the RS embedding of G_i into a $Q_{\mathrm{wd}(G_i)}$ and let $D^k_{G_i}(v)$ denote

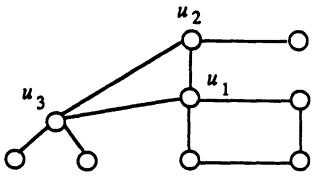
the kth bit of $D_{G_i}(v)$. Without loss of generality, we can assume $r_1 = \max_{1 \leq i \leq k} \{r_i = \text{wd}(G_i)\}$. Address each node w in G with $D_G(w)$ according to the following rules:

(1) For all $w \in V_i$, encode the first m bits of $D_G(w)$ with $D_{G_S}(u_i)$.

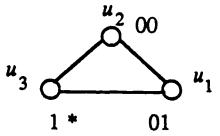
(2) For all $w \in V_i$, $m+1 \leq k \leq r_i + m$, **if** $D^k_{G_i}(u_i) = 1$ **then** $D^k_G(w) = D^{k-m}_{G_i}(w)$ **else** $D^k_G(w) = D^{k-m}_{G_i}(w)$. And, let $D^k_G(w) = *$ for $r_i + m + 1 \leq k \leq r_1 + m$.

This theorem follows from the existence of the above addressing scheme for G, whose length is $\max_{1 \leq i \leq |V_S|} \{\text{wd}(G_i)\} + m$. □

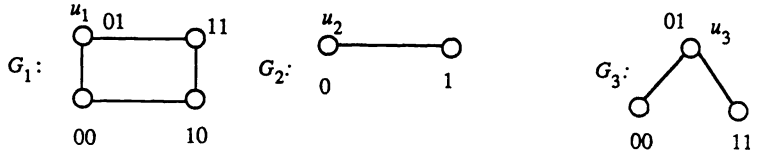
For an illustrative purpose, consider the example graph in Fig. 5(a). The induced subgraph of the node set $\{u_1, u_2, u_3\}$ is G_S in Fig. 5(b) and can be RS embedded into a Q_2. G_1, G_2, and G_3 in Fig. 5(c) are the resulting graphs after removing the edges of G_S from G. We have $\max_{1 \leq i \leq 3} \{\text{wd}(G_i)\} = 2$. By encoding the last two bits of $D_G(u_i)$, $1 \leq i \leq 3$, with 0's and *'s only, inverting some corresponding bits in the address $D_{G_i}(w)$ to preserve the adjacency in G_i, and using $D_{G_S}(u_i)$ as the leading portion of the address of $w \in V_i$, we get the address of each node in G as shown in Fig. 5(d).



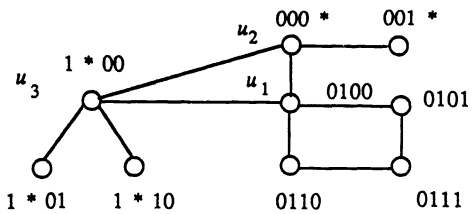
(a) An example graph G.



(b) The encoding of G_S.



(c) The encoding of G_1, G_2 and G_3.



(d) The encoding of G.

FIG. 5. *An illustrative example for Theorem* 3.

It is important to note that there does not exist any bound tighter than the upper bound provided in Theorem 3. This can be proved by showing the existence of some graphs for which the equality relation in Theorem 3 holds. For example, if $G_i = Q_p$ for $1 \leqq i \leqq |V_S|$ and some nonnegative integer p and $G_S = Q_m$, then $\text{wd}(G) = \text{wd}(G_i) + m$. Furthermore, the above theorem and Proposition 1 lead to the following corollary.

COROLLARY 3.1. *Let G_i, $1 \leqq i \leqq k$, be disjoint graphs and $G = \cup_{i=1}^{k} G_i$. Then,* $\text{wd}(G) \leqq \max_{1 \leqq i \leqq k} \{\text{wd}(G_i)\} + \lceil \log_2 k \rceil$.

Since every edge in a tree is a bridge, the following corollary, an immediate result of Theorem 3, can be used to determine a tighter upper bound for the weak cubical dimension of a tree.

COROLLARY 3.2. *Let $c_1(T)$ and $c_2(T)$ denote the two attached trees of the centroid edge of a tree T. Then,* $\text{wd}(T) \leqq \max\{\text{wd}(c_1(T)), \text{wd}(c_2(T))\} + 1$.

As it will be shown in § 4, Corollary 3.2 can be applied to implement a fast algorithm for the RS embedding of a given tree. In addition, the effects of join and product operations on the weak cubical dimension of graphs can be described below by Theorems 4 and 5.

THEOREM 4. $\text{wd}(G_1 + G_2) \leqq \text{wd}(G_1) + \text{wd}(G_2) + 1$.

Proof. Let $D_{G_j}(w)$ denote the address of $w \in G_j$, $j = 1, 2$, before a join operation. Address each node w in $G = G_1 + G_2$ according to the following rules:

(1) If $w \in V_1$, then $D_G(w) = 0* \cdots * D_{G_1}(w)$, which contains $\text{wd}(G_2)$ consecutive $*$'s before $D_{G_1}(w)$.

(2) If $w \in V_2$, then $D_G(w) = 1 D_{G_2}(w) * \cdots *$, which contains $\text{wd}(G_1)$ consecutive $*$'s after $D_{G_2}(w)$.

Clearly, the above addressing scheme, having length $\text{wd}(G_1) + \text{wd}(G_2) + 1$, not only preserves the original adjacency in G_1 and G_2, but also joins every pair of nodes (u_1, u_2), $u_1 \in V_1$, $u_2 \in V_2$. □

Note that Theorem 4 provides the best upper bound, since there exist some graphs for which the equality relation in Theorem 4 holds, e.g., $\text{wd}(Q_1 + Q_2) = \text{wd}(Q_1) + \text{wd}(Q_2) + 1$.

COROLLARY 4.1. *Let $\{V_1, V_2\}$ be a partition of the node set of a graph G_A, i.e., $V_1 \cap V_2 = \varnothing$ and $V_1 \cup V_2 = V_A$. Let the induced subgraphs of G_A with the node sets V_1 and V_2 be G_{I_1} and G_{I_2}, respectively. Then,* $\text{wd}(G_A) \leqq \text{wd}(G_{I_1}) + \text{wd}(G_{I_2}) + 1$.

Proof. Since $G_A \subseteq G_{I_1} + G_{I_2}$, the inequality $\text{wd}(G_A) \leqq \text{wd}(G_{I_1} + G_{I_2}) \leqq \text{wd}(G_{I_1}) + \text{wd}(G_{I_2}) + 1$ follows from Proposition 1 and Theorem 4. □

Let $G_{I_{A-S}}$ denote the induced subgraph of G_A with the node set $V_A - V_S$ where $V_S \subseteq V_A$. Then, we have the following corollary.

COROLLARY 4.2. $\text{wd}(G_A) \leqq \text{wd}(G_{I_{A-S}}) + |V_S|$.

Proof. Let G_{I_S} be the induced subgraph of G_A with V_S. From Corollary 4.1, $\text{wd}(G_A) \leqq \text{wd}(G_{I_S}) + \text{wd}(G_{I_{A-S}}) + 1$. In addition, we get $\text{wd}(G_{I_S}) \leqq |V_S| - 1$ from Corollary 2.2, and thus, this corollary follows. □

Using Corollaries 4.1 and 4.2, in § 4 we shall propose two fast algorithms for the RS embedding of a given graph. The relationship between the weak cubical dimensions of several graphs and that of their union can be described by the following corollary.

COROLLARY 4.3. *Let $G = \cup_{i=1}^{m} G_i$. Then,* $\text{wd}(G) \leqq \sum_{i=1}^{m} \text{wd}(G_i) + m - 1$.

Proof. First, prove the inequality $\text{wd}(G_1 \cup G_2) \leqq \text{wd}(G_1) + \text{wd}(G_2) + 1$. Let G_{I^*} be the induced subgraph of G_2 with the node set $V_2 - V_1$. Clearly, $G_1 \cup G_2 \subseteq G_1 + G_{I^*}$. Then, the inequality, $\text{wd}(G_1 \cup G_2) \leqq \text{wd}(G_1) + \text{wd}(G_{I^*}) + 1 \leqq \text{wd}(G_1) + \text{wd}(G_2) + 1$, follows from Theorem 4 and Proposition 1. The corollary follows by applying this inequality repeatedly. □

It is interesting to compare Corollary 4.3 with Corollary 3.1 that is applicable to disjoint graphs only. This result agrees with our intuition, since there are fewer restrictions in the RS embedding of disjoint graphs. Moreover, we have the following corollary for the complement of a graph.

COROLLARY 4.4. *Let G be a graph with n nodes. Then,* $\text{wd}(G)+\text{wd}(\bar{G}) \geqq n-2$.

Proof. Since $K_n = G \cup \bar{G}$, we get $n-1 = \text{wd}(K_n) = \text{wd}(G \cup \bar{G}) \leqq \text{wd}(G) + \text{wd}(\bar{G})+1$. □

Corollary 4.4 can sometimes be used to determine tighter bounds of the weak cubical dimension of a graph. For example, Corollary 2.2 offers a loose lower bound (6) of wd $(\overline{Q_6})$, whereas Corollary 4.4 gives a much tighter lower bound (56) of wd $(\overline{Q_6})$.

COROLLARY 4.5. *Let T be a tree with n nodes. Then,* $\text{wd}(T) \leqq 2\lceil \log_2 n \rceil$.

Proof. Since every tree is a bigraph, there exists an integer m such that T is a subgraph of $K_{n-m,m}$, where $K_{p,q}$ is a complete bigraph [Har69]. Without loss of generality, we can let $m \leqq n/2$. Clearly, $\lceil \log_2 m \rceil \leqq \lceil \log_2 n/2 \rceil = \lceil \log_2 n \rceil - 1$, and $\lceil \log_2 (n-m) \rceil \leqq \lceil \log_2 n \rceil$. Then, we have $\text{wd}(T) \leqq \text{wd}(K_{n-m,m}) \leqq \lceil \log_2 (n-m) \rceil + \lceil \log_2 m \rceil + 1 \leqq \lceil \log_2 n \rceil + \lceil \log_2 n \rceil - 1 + 1 = 2\lceil \log_2 n \rceil$. □

This corollary offers a tighter upper bound of the weak cubical dimension of a tree. Using Corollary 4.5, a fast RS embedding algorithm for a given tree will be developed in § 4.

THEOREM 5. $\text{wd}(G_1 \times G_2) \leqq \text{wd}(G_1) + \text{wd}(G_2)$.

Proof. For all $u_1 \in V_1$, $u_2 \in V_2$, let $D_{G_1}(u_1)$ and $D_{G_2}(u_2)$ be the addresses of u_1 and u_2 before the product operation. Encode the address of a node (u_1, u_2) in $G_1 \times G_2$ with the concatenation of their original addresses, $D_{G_1}(u_1)D_{G_2}(u_2)$, whose length is $\text{wd}(G_1) + \text{wd}(G_2)$. Obviously, the adjacency requirement in $G_1 \times G_2$ is preserved under the above addressing scheme, and thus the theorem follows. □

Note that Theorem 5 also provides the best upper bound. For example, $\text{wd}(Q_r \times Q_s) = \text{wd}(Q_r) + \text{wd}(Q_s)$ for positive integers r and s. It can also be verified that the above addressing schemes are valid for the squashed embedding problem, i.e., $N(G_1 \times G_2) \leqq N(G_1) + N(G_2)$. In addition, from the topology of a hypercube, we have the theorem below.

THEOREM 6. *Let q be an m-dimensional subcube of a Q_n, where $n \geqq m$. Then, q is adjacent to at most $(n-m)2^m$ subcubes within the Q_n.*

Proof. Without loss of generality, we can let the address of q be $00\cdots0*\cdots*$, in which there are $n-m$ consecutive 0's followed by m consecutive *'s. Note that the address of every Q_0 adjacent to q must have one 1 and $(n-m-1)$ 0's in its left $n-m$ bits. Among all Q_0's adjacent to q, there are 2^m different Q_0's with the kth bit equal to 1 for $m+1 \leqq k \leqq n$. Thus, q is adjacent to exactly $(n-m)2^m Q_0$'s. □

In what follows, the number $(n-m)2^m$ will be referred to as the *adjacency number* of q, where q is an m-dimensional subcube of a Q_n.

COROLLARY 6.1. *Let $\{d_i\}$ be the degree sequence of a graph G_A. If wd $(G_A) \leqq m$, then $\sum_{i=1}^{|V_A|} 2^{b_i} \leqq 2^m$, where for each $1 \leqq i \leqq |V_A|$, b_i is the least nonnegative integer such that the adjacency number of $Q_{b_i} \geqq d_i$.*

Proof. From Theorem 6, the dimension of the subcube assigned to a task node n_i with degree d_i cannot be less than b_i. This corollary follows from the fact that the total number of nodes in a Q_m assigned to G_A must be less than or equal to the total number of its nodes, 2^m. □

Since $b_i \geqq 0$, $1 \leqq i \leqq |V_A|$, we have $\sum_{i=1}^{|V_A|} 2^{b_i} \geqq |V_A|$, meaning that using the knowledge of degree sequence provides a tighter lower bound than Corollary 2.2. Moreover, the relationship between the number of edges in a graph and its weak cubical dimension can be described by the following theorem.

THEOREM 7. *Let* $m = \mathrm{wd}(G_A)$ *and* $k = \lfloor \log_2 (2m/|V_A|) \rfloor$. *Then,*

$$|E_A| \leqq \tfrac{1}{2}[(2^{m-k} - |V_A|) \min\{(m-k-1)2^{k+1}, |V_A|-1\} + (2|V_A| - 2^{m-k}) \min\{(m-k)2^k, |V_A|-1\}].$$

Note that Theorem 7 provides a necessary condition for a task graph with a given number of edges to be RS embedded into a cube. This condition provides information useful for the decomposition of a task into (interacting) modules in such a way that the resulting task graph can be embedded into a cube of a given dimension. In order to simplify the proof of Theorem 7, first it is necessary to introduce the following two lemmas.

LEMMA 1. *The maximal adjacency number of a subcube in a* Q_n *is* 2^{n-1}, *which is attained by a subcube of dimension* $n-1$ *or* $n-2$.

Proof. Let $F(k) = (n-k)2^k$. Since $dF(k)/dk = 2^k[(n-k)\log_e 2 - 1] > 0$ for $0 \leqq k \leqq n-2$, and $F(n-1) = F(n-2) = 2^{n-1} > F(n) = 0$, this lemma follows. □

LEMMA 2. *Let* a_i, $1 \leqq i \leqq r$, *be nonnegative integers and* $\sum_{i=1}^{r} 2^{a_i} \leqq 2^m$. *Then,* $f(a_1, a_2, \cdots, a_r) = \sum_{i=1}^{r} (m-a_i)2^{a_i} \leqq 2^{m-k} - r)(m-k-1)2^{k+1} + (2r - 2^{m-k})(m-k)2^k$, *where* $k = \lfloor \log_2 (2^m/r) \rfloor$.

Proof. Let $(a_1^*, a_2^*, \cdots, a_r^*)$ be the vector that maximizes $f(a_1, a_2, \cdots, a_r)$, i.e., $f^* = f(a_1^*, \cdots, a_r^*)$. From the proof of Lemma 1, we know that $g(a_i) = (m-a_i)2^{a_i}$ is a monotonically increasing function in the integer variables a_i, where $1 \leqq a_i \leqq m-1$. Let $a_p^* = \min_{1 \leqq i \leqq r}\{a_i^*\}$ and suppose $2^m - \sum_{i=1}^{r} 2^{a_i^*} > 0$. Then, $2^m - \sum_{i=1}^{r} 2^{a_i^*}$ must be an integral multiple of $2^{a_p^*}$. This is impossible, since the a_p^* in the function f can be replaced with $a_p^* + 1$, resulting in a larger f-value than f^*. Thus, $\sum_{i=1}^{r} 2^{a_i^*} = 2^m$, implying that there exist a_p^* and a_x^*, $p \neq x$, such that $a_p^* = a_x^* = \min_{1 \leqq i \leqq r}\{a_i^*\}$. We claim that $\max_{1 \leqq i \leqq r}\{a_i^*\} - \min_{1 \leqq i \leqq r}\{a_i^*\} \leqq 1$, and then this lemma follows from the fact that $2^{m-k} - r$ variables among a_i^*'s are $k+1$ and $2r - 2^{m-k}$ variables among a_i^*'s are k.

Let $a_y^* = \max_{1 \leqq i \leqq r}\{a_i^*\}$, $a_p^* = a_x^* = \min_{1 \leqq i \leqq r}\{a_i^*\}$ and suppose $a_y^* - a_p^* \geqq 2$. Then, we have $2^{a_y^*} + 2^{a_p^*} + 2^{a_x^*} = 2^{a_y^*-1} + 2^{a_y^*-1} + 2^{a_p^*+1}$ and $(m-a_y^*)2^{a_y^*} + (m-a_p^*)2^{a_p^*} + (m-a_x^*)2^{a_x^*} \leqq (m-a_y^*+1)2^{a_y^*-1} + (m-a_y^*+1)2^{a_y^*-1} + (m-a_p^*-1)2^{a_p^*+1}$. This leads to a contradiction, because a_y^*, a_p^*, and a_x^* in the function f can be replaced by $a_y^* - 1$, $a_y^* - 1$, and $a_p^* + 1$, respectively, yielding a larger f-value than f^*. Therefore, the claim $\max_{1 \leqq i \leqq r}\{a_i^*\} - \min_{1 \leqq i \leqq r}\{a_i^*\} \leqq 1$ is proved and, thus, this lemma follows. □

Proof of Theorem 7. Let a_i be the dimension of the subcube assigned to a task node n_i in G_A, $1 \leqq i \leqq |V_A| = r$. Then, $\sum_{i=1}^{r} 2^{a_i} \leqq 2^m$ follows from the capacity constraint of a Q_m. Note that the adjacency number of the subcube assigned to n_i is $(m-a_i)2^{a_i}$ and the degree of any node in $G_A \leqq |V_A| - 1$. This theorem follows from Lemma 2 and the fact that $\sum_{i=1}^{r} d_i = 2|E_A|$. □

When a graph belongs to some regular families, its weak cubical dimension can be determined by the theorem below.

THEOREM 8. *The weak cubical dimensions of a cycle* C_m, *a path* P_m, *and a star* S_m *can be determined by the following formulas*:

(i) $\mathrm{wd}(C_m) = \lceil \log_2 m \rceil$;

(ii) $\mathrm{wd}(P_m) = \lceil \log_2 m \rceil$;

(iii) $\mathrm{wd}(S_m) = \lceil \log_2 (m-1) \rceil + 1$.

Proof. Consider (i) first. Clearly, $\mathrm{wd}(C_m) \geqq \lceil \log_2 m \rceil \equiv k$. From the existence of Hamiltonian cycles in a Q_k, we know that a C_m can be RS embedded into a Q_k by embedding $2^k - m$ nodes of the C_m into Q_1's and $2m - 2^k$ nodes of the C_m into Q_0's, and thus (i) is proved. Part (ii) follows from (i) immediately.

Consider (iii). Let δ_k be a trivial graph with k nodes and no edges. Note that $S_m = \delta_{m-1} + \delta_1$ and $\mathrm{wd}(\delta_k) = \lceil \log_2 k \rceil$. Then, we have $\mathrm{wd}(S_m) \leqq \lceil \log_2 (m-1) \rceil + 1$ by

Theorem 4. From Lemma 1, we know that the maximal adjacency number of a subcube in a $Q_{\lceil \log_2(m-1) \rceil}$ is $2^{\lceil \log_2(m-1) \rceil - 1} < m-1$. Thus, $\text{wd}(S_m) > \lceil \log_2(m-1) \rceil$ and (iii) follows. □

COROLLARY 8.1. *Let $m_{c\times d}$ denote a $(c\times d)$-dimensional mesh. Then,* $\text{wd}(m_{c\times d}) \leqq \lceil \log_2 c \rceil + \lceil \log_2 d \rceil$.

Proof. Since a $(c\times d)$-dimensional mesh is $P_c \times P_d$, this corollary follows from Theorem 5 and (ii) of Theorem 8. □

Due to its nature of NP-completeness, the weak cubical dimension of a graph is in general very difficult to characterize. However, as we shall show in the following section, the mathematical properties derived in this section can play a significant role in designing efficient algorithms for the RS embedding of a given graph.

4. Algorithms for relaxed squashed embedding. The mathematical properties derived in § 3 are applied to the design of algorithms for the RS embedding. Fast algorithms of polynomial time complexity that are efficient but may not provide the minimal cube required for a given task graph are presented first. Then, a heuristic search algorithm is developed to determine the weak cubical dimension of a graph.

4.1. Fast algorithms for RS-embedding. Since every tree is a bigraph, we have an efficient addressing scheme for a tree with n nodes as described below.

ALGORITHM $A_1(T)$/*. This algorithm uses the property that every tree T is a bigraph and determines an efficient addressing scheme for T.*/

Step (1). Choose an arbitrary node in T. Label it with a symbol +.

Step (2). Label with −'s all the nodes adjacent to each node labeled with +. **If** every node in T has been labeled with + or − **then** goto Step (4).

Step (3). Label with +'s all the nodes adjacent to each node labeled with −. **If** every node in T has been labeled with + or − **then** goto Step (4) **else** goto Step (2).

Step (4). Suppose there are j nodes with + and k nodes labeled with −. Then, encode all the nodes labeled with + with $0*\cdots*B^{(+)}(i)$, $0 \leqq i \leqq j-1$, where $B^{(+)}(i)$ is a binary representation of the number i with $\lceil \log_2 j \rceil$ bits, which follows $\lceil \log_2 k \rceil$ *'s. Also, encode all the nodes labeled with − with $1B^{(-)}(i)*\cdots*$, $0 \leqq i \leqq k-1$, where $B^{(-)}$(i) is a binary representation of the number i with $\lceil \log_2 k \rceil$ bits, followed by $\lceil \log_2 j \rceil$ *'s.

By Corollary 4.5, the length of the above addressing scheme must be less than or equal to $2\lceil \log_2 n \rceil$. (This, in general, is significantly less than $n-1$ for a large n.) Although the required length of the addressing scheme used in A_1 may be larger than the weak cubical dimension of the tree, A_1 is favorable in some cases due to its *linear* complexity.

Corollary 3.2 suggests the following algorithm that also determines a cube required to accommodate a tree.

ALGORITHM $A_2(T)$/*. This algorithm determines the dimension of a cube to accommodate a task tree T.*/

Step (1). **If** T is a star or a path **then** determine $\text{wd}(T)$ by Theorem 8 and return $\text{wd}(T)$ **else** compute the weight of each edge and determine the centroid edge of the tree.

Step (2). Let T_1 and T_2 be the two attached trees of the centroid edge of T. Return $\max\{A_2(T_1), A_2(T_2)\}+1$.

A_2 is recursive and uses the divide-and-conquer technique. We decompose a tree by removing its centroid edge first, and then continue to decompose the remaining

trees in the same way until only stars or paths are left, whose weak cubical dimension can be determined by Theorem 8. An illustrative example for this algorithm can be found in § 5. Note that the complexity of A_2 depends on the degree of sophistication in the way of determining the centroid edge. Nevertheless, it is easy to verify that A_2 requires only polynomial time.

Consider the case when the task graph is an *arbitrary* graph. Clearly, using Corollary 2.1, we can derive a straightforward algorithm for the RS embedding of each graph: address any first two nodes with $0\cdots0$ and $0\cdots01$, respectively, each of which consists of $n-1$ bits, and then the kth node, $3 \leqq k \leqq n$, with $0\cdots01*\cdots*$ that consists of $n-k$ consecutive 0's and $k-2$ consecutive $*$'s. However, despite its linear complexity, this naive algorithm is not used for a better system utilization. Instead, we present Algorithm A_3 below that is derived from Corollary 4.2.

ALGORITHM $A_3(G)$/*. Using the technique of node-removing, this is a fast algorithm to determine the size of the cube required for a given task graph.*/

Step (1). Let n^* be the node with the largest degree among all the nodes in G. **If** $d(n^*) \leqq 2$ **then** goto Step (2) **else** goto Step (3).

Step (2). Determine all the cycles in G, denoted by $C^1, C^2, \cdots, C^m$, and the least integer p such that $2^p \geqq \sum_{i=1}^{m} 2^{\lceil \log_2 |C^i| \rceil} + 2^{\lceil \log_2 (|V| - \sum_{i=1}^{m} |C^i|) \rceil}$, where $|C^i|$ is the number of nodes in the cycle C^i. Return p.

Step (3). Let $G_1 := G - n^*$ and return $A_3(G_1)+1$.

Using A_3, a graph is reduced by removing the node with the largest degree from the graph. The reduction steps are performed repeatedly until the graph is reduced to the extent that it contains only disjoint cycles and paths. By Corollary 4.2, the size determined in Step (2) plus the total number of nodes removed will be the dimension of a cube required to accommodate the original task graph. Note that in A_2 and A_3, it is required to determine if a graph belongs to some families of graph such as paths, stars, and cycles. For this purpose, an adjacency matrix [Har69] can be used to represent each task graph, since these families of graph can be easily identified if they are represented with adjacency matrices. Moreover, by Corollary 4.1 we can modify Step (3) of A_3 as follows and get a generalized version of A_3, called Algorithm A_4.

Step (3′). Partition V into V_1 and V_2. Let G_{I_1} and G_{I_2} be, respectively, the induced subgraphs of G with the node sets V_1 and V_2. Return $A_4(G_{I_1}) + A_4(G_{I_2}) + 1$.

Several heuristic approaches can be employed in determining how to partition the node set V into V_1 and V_2 in Step (3′) of A_4. Clearly, a more sophisticated method will lead to an addressing scheme with a shorter length at the cost of higher computational costs of A_4.

Although the above proposed algorithms are efficient in determining the required cube for a given graph, the resulting cube may not be minimal. As far as the system utilization is concerned, we want to find the minimal subcube required for a given task graph. This is explored in § 4.2.

4.2. An algorithm for determining the weak cubical dimension. To determine the weak cubical dimension of a task graph, first we present an algorithm that determines whether or not there is an RS embedding from a given task graph into a cube. Then, the algorithm is applied to determine the weak cubical dimension of the graph. To facilitate our discussion, we label the task graph as follows. Label the node with the largest degree with n_1 and let $X := \{n_1\}$ and $i := 2$. Then, among all the nodes that are adjacent to any node in X and are not in X, choose a node with the largest degree

and label this node with n_i. Then, let $X := X \cup \{n_i\}$ and $i := i+1$. Repeat the same procedure until all nodes are labeled.

Now, we want to assign a subcube within the n-cube to each node in a task graph, node by node, subject to the adjacency requirement in the RS embedding. Clearly, this problem is a graph matching problem and can be solved by a state-space search similar to the one in [ShT85]. In what follows, we shall formulate a heuristic function, and the A^* search algorithm [Nil80] will then be used to determine the existence of an RS embedding from a given graph into a cube. The following definitions are necessary to facilitate our presentation.

DEFINITION 2. The *merge* operation, denoted by $\odot$, of two sets of subcubes, U_1 and U_2, is defined as

$$U_1 \odot U_2 = \{\tau \mid \tau = \text{lcm}(\alpha, \beta) \text{ for } \alpha \in U_1 \text{ and } \beta \in U_2\}.$$

The merge operation among $k \geqq 2$ sets of subcubes is written as $\odot_{1 \leqq i \leqq k} U_i$.

DEFINITION 3. The *exclusion* operation of two sets of subcubes, U_1 and U_2, is defined as

$$U_1 - U_2 = \{r \mid r \in U_1 \text{ and } \gcd(t, r) = \varnothing, \forall t \in U_2\}.$$

DEFINITION 4. The *reduced set* of a set of subcubes U is defined as

$$\text{Rd}(U) = U - \{r \mid r \in U \text{ and } t \subset r \text{ for some } t \in U\}.$$

For example, let $U_1 = \{0{**}, 0{*}0, 01{*}, 001\}$, $U_2 = \{00{*}, 10{*}\}$, and $U_3 = \{001\}$. Then, $\text{Rd}(U_1) = \{0{*}0, 01{*}, 001\}$, $U_1 - U_2 = \{01{*}\}$, and $U_2 \odot U_3 = \{00{*}, {*}0{*}\}$. Recall that $B(n_i)$ is the set of all nodes adjacent to n_i in the task graph G_T. Let M^i denote the partial mapping for the task node n_j, $1 \leqq j \leqq i$. Let $A_{n_j}^{(i)}$ be the set of unoccupied Q_0's that are adjacent to $D(n_j)$ under the partial mapping M^i. Also, define the set of *essential subcubes* of n_j under the partial mapping M^i, denoted by $E_{n_j}^{(i)}$, as the reduced set of unoccupied subcubes that are adjacent to the subcubes assigned to all $n_k \in B(n_j)$, $1 \leqq k \leqq i$. For example, suppose that in the graph of Fig. 3, we have $D(n_1) = 00{*}$, $D(n_2) = 010$. Then, $A_{n_1}^{(2)} = \{011, 100, 101\}$, $A_{n_2}^{(2)} = \{110, 011\}$, and $E_{n_3}^{(2)} = \{011, 1{*}0\}$. That is, the subcube to be assigned to n_3 should contain either 011 or $1{*}0$ to satisfy the adjacency requirement. Then, the $E_{n_k}^{(i)}$ generated under M^i can be expressed as follows:

$$E_{n_k}^{(i)} = Rd\left(\bigodot_{\substack{n_j \in B(n_k) \\ 1 \leqq j \leqq i}} A_{n_j}^{(i)} \right) - \bigcup_{j=1}^{i} D(n_j) \quad \forall k > i. \tag{1}$$

From this formula, we can determine the sets of all essential subcubes of unassigned task nodes. Note that $E_{n_k}^{(i)}$ is determined by $A_{n_j}^{(i)}$, for all $n_j \in B(n_k)$, and the adjacency requirement, and the term $\cup_{j=1}^{i} D(n_j)$ in equation (1) is necessary to exclude the possibility of allocating the already occupied subcubes. Given a partial mapping M^i, the set of all possible subcubes that can be assigned to the task node n_{i+1} is represented by

$$\text{Sp}(E_{n_{i+1}}^{(i)}) = \left\{ q \;\middle|\; \text{there exists } t \in E_{n_{i+1}}^{(i)},\ t \subseteq q \text{ and } \sum_{j=1}^{i} |D(n_j)| + |q| < 2^n - (|V_T| - i - 1) \right\} - \bigcup_{j=1}^{i} D(n_j). \tag{2}$$

The inequality in equation (2) is to ensure that after the allocation of q to n_{i+1}, there is a sufficient number of nodes in the Q_n to be assigned to the remaining task nodes. From equation (2), it is easy to see that *both* (i) more subcubes in $E_{n_{i+1}}^{(i)}$ *and* (ii) subcubes of smaller dimensions in $E_{n_{i+1}}^{(i)}$ will allow for more freedom in allocating

a required subcube to n_{i+1}. This in turn implies that the sets of essential subcubes of unassigned nodes can be used in determining the heuristic value of the node in the search tree associated with the partial mapping made thus far.

Suppose that a node p in the search tree corresponds to the allocation of a subcube q to the task node n_i. Then, $A_{n_i}^{(i)}$ can be determined by the method introduced in the proof of Theorem 6, and the sets $A_{n_k}^{(i)}$, $1 \leqq k < i$, can be updated from their predecessors by equation (3) below. They are in turn used to determine $E_{n_k}^{(i)}$, $k > i$, by using equation (1).

$$A_{n_k}^{(i)} = A_{n_k}^{(i-1)} - \{q\}, \qquad 1 \leqq k < i. \tag{3}$$

Combining all the results and findings discussed thus far, a heuristic function for each node p in the search tree can be contructed as follows:

$$\begin{aligned} &f(p) = g(p) + h(p) \text{ where } g(p) = i2^n, \\ &h(p) = \sum_{i+1}^{|V_s|} V(E_{n_k}^{(i)}), \quad \text{and} \quad V(E_{n_k}^{(i)}) = \sum_{t \in E_{n_k}^{(i)}} \frac{1}{2^{|t|}}. \end{aligned} \tag{4}$$

Note that the heuristic value (or h-value) of a node p is defined so that *both* more subcubes in $E_{n_k}^{(i)}$ *and* subcubes of smaller dimensions in $E_{n_k}^{(i)}$, $k > i$, will result in a larger h-value of p. Applying the above heuristic function to the A^* search algorithm, we propose the following RS-embedding algorithm.

ALGORITHM RS-embedding $(G_T, k)/*$. This algorithm determines the existence of an RS embedding from a task graph G_T into a Q_k.*/

Step (1). Without loss of generality, let the list OPEN be $\{00\cdots0, 0\cdots0*, \cdots, 0*\cdots*\}$ consisting of k strings of length $k-1$ each. Check the validity of these nodes by using Theorem 6. Compute equations (1), (3), and (4) for nodes in the list OPEN.

Step (2). If OPEN $= \varnothing$, report *false* and exit. Determine the node p with the maximal f-value from the list OPEN. Remove it from OPEN and put it into the list CLOSE. If node p is associated with the allocation of the last node, report *true* and exit.

Step (3). Determine the successors of p by equation (2). Check the validity of successors by using Theorem 6, evaluate equations (1), (3), and (4) for valid nodes, and put these nodes in the list OPEN.

Step (4). Go to Step 2.

According to the formulation of the heuristic function, the h-value of any node in the search tree must be less than 2^n. This means that our heuristic function satisfies the monotone restriction [Nil80]. In other words, the goal nodes whose distances from the root node are $|V_T|$ have the maximal f-value. Thus, if there exist goal nodes in the search tree, then one of them should be reached in a finite number of steps. Note that there may be more than one goal node in the search tree. However, we are concerned only with the existence of such nodes, rather than the number of such nodes in the search tree.

Using the RS embedding algorithm, we can determine the existence of an RS embedding of a given graph into a cube. For some graphs whose weak cubical dimensions are in a narrow range, a linear search algorithm is suggested as follows. Since an unsuccessful search usually involves more computational costs than a successful one, the linear search algorithm is designed to perform a top-down search for the weak cubical dimension of a graph. Let *ub* and *lb* be, respectively, the upper and lower bounds of the weak cubical dimension of the graph determined by the mathematical properties in § 3. Using a linear search, the expected number of times to execute

the RS embedding algorithm is $(ub - lb + 1)/2$, containing $(ub - lb - 1)/2$ successful searches and one unsuccessful search.

On the other hand, the bounds for the weak cubical dimension of some other graphs may be quite loose, making any linear search algorithm inefficient. For those graphs, a binary search algorithm is suggested. Then, the expected number of times used to execute the RS embedding algorithm becomes $\lceil \log_2 (ub - lb + 1) \rceil$, in which successful and unsuccessful searches have the same likelihood of occurrence.

5. Examples. In this section, examples are presented to illustrate the application of the results developed in § 3 and the execution of the algorithms proposed in § 4.

Example 1. Consider the example graph G shown in Fig. 6. From Corollary 2.2, we have $\lceil \log_2 6 \rceil = 3 \leqq \text{wd}(G) \leqq 6 - 1 = 5$. Moreover, from Corollary 6.1 we get $\text{wd}(G) \neq 3$. Let $V_1 = \{n_1, n_2, n_5, n_6\}$ and $V_2 = \{n_3, n_4\}$. Denote the induced subgraph with the node in V_j by $G_{I_j}, j = 1, 2$. Clearly, $G_{I_1} = C_4$ and $G_{I_2} = P_2$. Thus, from Corollary 4.1 we obtain $\text{wd}(G) \leqq \text{wd}(C_4) + \text{wd}(P_2) + 1 = 2 + 1 + 1$. From the above results, we get $\text{wd}(G) = 4$.

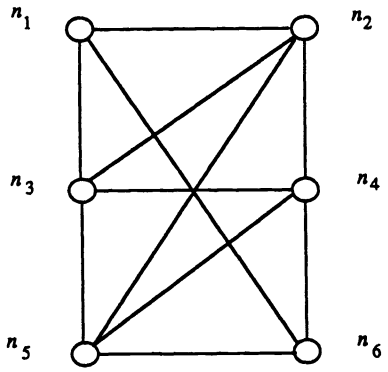
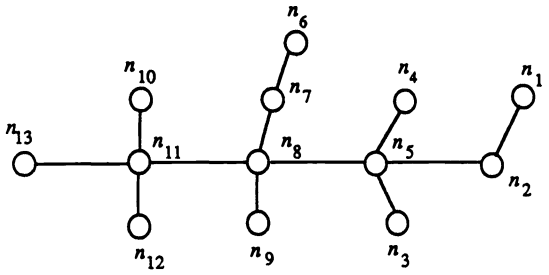


FIG. 6. *An example graph G.*

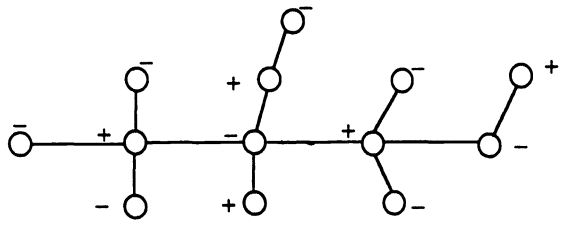
Example 2. Consider the task tree shown in Fig. 7(a). Using A_1, we obtained a labeled tree as shown in Fig. 7(b), and then derived an addressing scheme with the length $\lceil \log_2 5 \rceil + \lceil \log_2 8 \rceil + 1 = 7$. For example, under this addressing scheme the assigned address of n_5, the second node labeled with +, is 0010*** and that of n_8, the fourth node labeled with −, is 1***100.

The application of A_2 to the tree in Fig. 7(a) can be described by Fig. 8. The tree with the weight of each edge specified is given in Fig. 8(a), and the operations of A_2 are illustrated by the binary tree in Fig. 8(b). Each internal node in Fig. 8(b) has two children that are the disjoint trees resulting from the removal of its centroid edge. For example, T_1 and T_2 are the two attached trees of the edge (n_5, n_8), while n_8 is in T_1 and n_5 in T_2. Using A_2, we get $A_2(T_3) = 3$, $A_2(T_4) = 2$, $A_2(T_5) = 2$, $A_2(T_6) = 1$, $A_2(T_1) = 4$, $A_2(T_2) = 3$, and $A_2(T) = 5$.

Example 3. Consider the example graph $G = (V, E)$ of Fig. 3. Again, we have $\text{wd}(G) \geqq \lceil \log_2 |V| \rceil = 3$ from Corollary 2.2. In addition, the induced subgraph of G with the node set $\{n_2, n_3, n_4, n_5, n_6\}$ is P_5 whose weak cubical dimension is 3. From Corollary 4.2, we get $3 \leqq \text{wd}(G) \leqq 4$. To determine $\text{wd}(G)$, we must apply the RS embedding algorithm.



(a) An example tree T.



(b) The labeling of the tree T.

FIG. 7. *An example of labeling a tree.*

Figure 9 shows the state-space search tree for a Q_3 to accommodate the task graph. Let L denote the list of occupied subcubes. By using the heuristic search algorithm in § 4.2, initially we get OPEN = {000, 00*, 0**}. Note that the allocation $(n_1 \leftarrow 000)$ and $(n_1 \leftarrow 0**)$ will be eliminated by Theorem 6 and equation (2), respectively. Thus, node A is the only node to be expanded. According to equations (1), (3), and (4), we have the following:

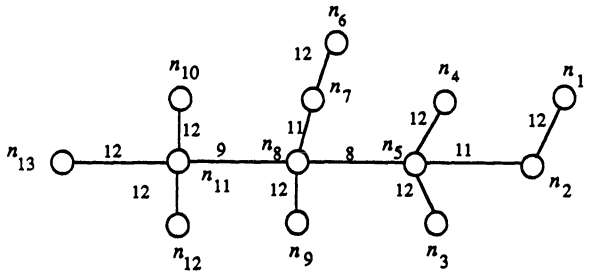
(1) Node $A(n_1 \leftarrow 00*)$: $A^{(1)}_{n_1} = E^{(1)}_{n_2} = E^{(1)}_{n_3} = E^{(1)}_{n_4} = E^{(1)}_{n_6} = \{010, 011, 100, 101\}$, $E^{(1)}_{n_5} = \{010, 011, 100, 101, 110, 111\}$, and $L = \{00*\}$. $g(A) = 2^3$, $h(A) = 4+4+4+4+6$, and $f(A) = 30$.

Under the allocation $M^1 = (n_1 \leftarrow 00*)$, we get $\mathrm{Sp}(E^{(1)}_{n_2}) = \{010, 011, 101, 100, 01*, 10*, *10, *11, 1*0, 1*1\}$ from equation (2). Due to the symmetry, only the computation for the nodes B, C, and D is shown below.

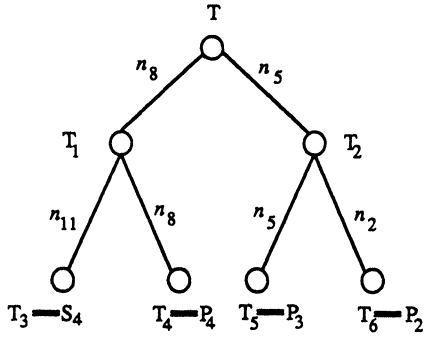
(2) Node B $(n_2 \leftarrow 010)$: $A^{(2)}_{n_1} = \{011, 100, 101\}$, $A^{(2)}_{n_2} = \{011, 110\}$, $E^{(2)}_{n_3} = \mathrm{Rd}(A^{(2)}_{n_1} \odot A^{(2)}_{n_2}) = \{011, 1*0\}$, $E^{(2)}_{n_4} = E^{(2)}_{n_6} = \{011, 100, 101\}$, $E^{(2)}_{n_5} = \{011, 110\}$, and $L = \{00*, 010\}$. $g(B) = 2^3 2 = 16$, $h(B) = 1\frac{1}{2}+3+2+3$, and $f(B) = 25\frac{1}{2}$.

(3) Node C $(n_2 \leftarrow 01*)$: $A^{(2)}_{n_1} = \{100, 101\}$, $A^{(2)}_{n_2} = \{110, 111\}$, $E^{(2)}_{n_3} = \mathrm{Rd}(A^{(2)}_{n_1} \odot A^{(2)}_{n_2}) = \{1*0, 1*1\}$, $E^{(2)}_{n_4} = E^{(2)}_{n_6} = \{100, 101\}$, $E^{(2)}_{n_5} = \{110, 111\}$, and $L = \{00*, 01*\}$. $g(C) = 2^3 2 = 16$, $h(C) = \frac{1}{2}+\frac{1}{2}+2+2+2$, and $f(C) = 23$.

(4) Node D $(n_2 \leftarrow *10)$: $A^{(2)}_{n_1} = \{011, 100, 101\}$, $A^{(2)}_{n_2} = \{100, 011, 111\}$, $E^{(2)}_{n_3} = \{011, 100, 1*1\}$, $E^{(2)}_{n_4} = E^{(2)}_{n_6} = \{101, 100, 011\}$, $E^{(2)}_{n_5} = \{100, 011, 111\}$, and $L = \{00*, *10\}$. $g(D) = 2^3 2 = 16$, $h(D) = 2\frac{1}{2}+3+3+3$, and $f(D) = 27\frac{1}{2}$.



(a) A tree with the weight of each edge specified.



(b) The determination of the subcube required for a tree using divide and conquer.

FIG. 8. *An application example for Corollary* 3.2.

Since node D has the maximal f-value among the three nodes, D is now the next node to be expanded. Note that under the partial mapping $M^2 = (n_1 \leftarrow 00*, n_2 \leftarrow *10)$, $\mathrm{Sp}(E^{(2)}_{n_3}) = \{011, 100\}$. Then, using the same procedure, the remaining computation for the heuristic search algorithm is given below.

(5) Node E $(n_3 \leftarrow 011)$: $g(E) = 2^3 3 = 24$, $h(E) = \frac{1}{2} + 2 + 2$ and $f(E) = 28\frac{1}{2}$.

(6) Node F $(n_3 \leftarrow 100)$: $g(F) = 2^3 3 = 24$, $h(F) = 1 + 2 + 2$ and $f(F) = 29$.

Node F is now the next node to be expanded. Using the same procedure, it is easy to verify that all the children of nodes F and E can be pruned, and node B becomes the next node to be expanded, since $f(B) > f(C)$. Thus, $M^2 = (n_1 \leftarrow 00*, n_2 \leftarrow 010)$, leading to the following results: $f(G) = 27\frac{1}{2}$, $f(H) = 28\frac{1}{2}$, and $f(I) = 28\frac{1}{2}$.

Now, node H is to be expanded. Continuing the same procedure, we obtain the following results: $f(J) = 34$, $f(K) = 41$, and $f(L) = 48$.

Since the node L is associated with the allocation of the last node, *true* will be reported, meaning that an RS embedding of G into Q_3 has been found and $\mathrm{wd}(G) = 3$. It is easy to see that the proposed heuristic function plays an important role in guiding and, thus, speeding up the state-space search. Use of the f-value of a node as an indication of the likelihood for the node to lead to a successful mapping results in a significant improvement over a blind search. However, as the size of the task graph increases, large amounts of computation will be required for the node expansion of the heuristic search, and the necessity of applying this state search algorithm to every graph calls for an optimization in some sense. For example, depending on the system's objective function, one can strike a compromise between the system utilization and the computational cost.

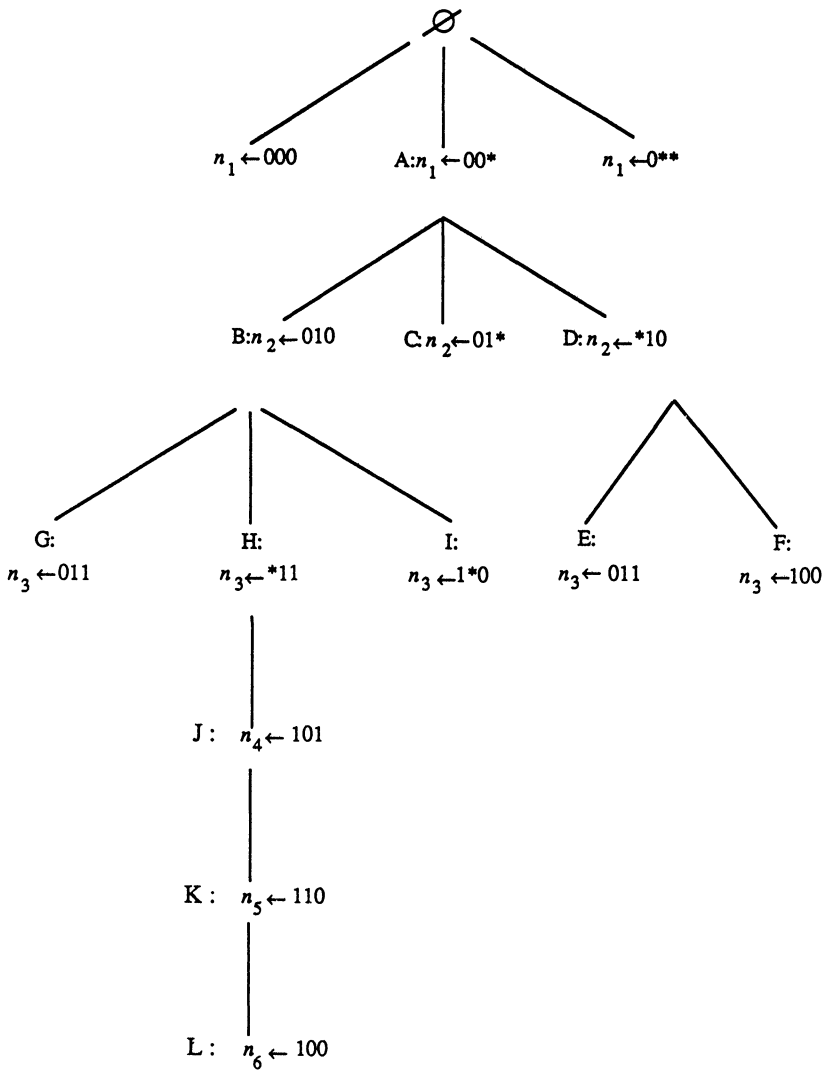


FIG. 9. *Part of the search tree.*

6. Discussion and conclusion. We have proposed and investigated a new type of embedding, called the RS embedding, that was motivated by the problem of allocating tasks in a hypercube multicomputer. Several mathematical properties for the weak cubical dimension have been derived that are not only applied to develop fast algorithms for the RS embedding, but also used to guide the heuristic search for an RS embedding.

The problem studied in this paper can be generalized by considering both the computation load of each module and the communication load between modules in a task graph. The task graph can then be represented by a labeled graph. The number assigned to a node of the graph denotes the dimension of a subcube required for the corresponding module to perform the computation load of the module. The number assigned to an edge of the task graph represents the required number of communication links between the two subcubes assigned to the two task nodes incident to this edge to provide enough communication capacity between them. Note that two adjacent

subcubes could have different numbers of connecting links. For example, *10* and 00*0 are connected by a link (0100, 0000), and 010* and 00** are connected by two links, (0100, 0000) and (0101, 0001). Thus, the constraint treated in this paper is a special case of the generalized version, since one is assigned to every node and every edge of the task graph.

Clearly, the inclusion of computation and communication loads of modules increases the number of constraints to meet, and thus, makes the RS embedding more realistic but complicated.

REFERENCES

[BGK72] L. H. BRANDENBURG, B. GOPINATH, AND R. P. KURSHAN, *On the addressing problem of loop switching*, Bell System Tech. J., 51 (1972), pp. 1445-1469.

[ChS87] M.-S. CHEN AND K. G. SHIN, *Processor allocation in an N-cube multiprocessor using Gray codes*, IEEE Trans. Comput., 36 (1987), pp. 1396-1407.

[Cor85] N. CORP., NCUBE/ten: an overview, November 1985.

[CKV87] G. CYBENKO, D. W. KRUMME, AND K. N. VENKATARAMAN, *Fixed hypercube embedding*, Inform. Process. Lett., 25 (1987), pp. 35-39.

[Fir65] V. V. FIRSOV, *On isometric embedding of a graph into a Boolean cube*, Cybernetics, 1 (1965), pp. 112-113.

[GaG75] M. R. GAREY AND R. L. GRAHAM, *On cubical graphs*, J. Combin. Theory Ser. B, 18 (1975), pp. 84-95.

[GaJ79] M. R. GAREY AND D. S. JOHNSON, *Computer and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979.

[GrP71] R. L. GRAHAM AND H. O. POLLAK, *On the addressing problem for loop switching*, Bell System Tech. J., 50 (1971), pp. 2495-2519.

[GrP72] ———, *On embedding graph in squashed cubes*, Springer Lecture Notes Math., 303 (1972), pp. 99-110.

[Har69] F. HARARY, *Graph Theory*, Addison-Wesley, MA, 1969.

[Har86] ———, *The Topological Cubical Dimension of a Graph*, Lecture Notes on the first Japan Conference on Graph Theory and Applications, Hakone, Japan, June 3, 1986.

[Har80] J. HARTMAN, *On homeomorphic embeddings of $K_{m,n}$ in the cube*, Canad. J. Math., 32 (1980), pp. 644-652.

[HaM72] I. HAVEL AND J. MORAVEK, *B-valuations of graphs*, Czech. Math. J., 22 (1972), pp. 338-351.

[KVC85] D. W. KRUMME, K. N. VENKATARAMAN, AND G. CYBENKO, *Hypercube embedding is NP-complete*, Proc. First Hypercube Conference, Knoxville, TN, August 1985, pp. 148-157.

[Nil80] N. J. NILSSON, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[Pie72] J. R. PIERCE, *Network for block switching of data*, Bell System Tech. J., 51 (1972), pp. 1133-1145.

[Sei85] C. L. SEITZ, *The cosmic cube*, Commun. Assoc. Comput. Mach., 28 (1985), pp. 22-33.

[ShT85] C. C. SHEN AND W. H. TSAI, *A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion*, IEEE Trans. Comput., 34 (1985), pp. 197-203.

[Val82] L. G. VALIANT, *A scheme for fast parallel communication*, SIAM J. Comput., 11 (1982), pp. 350-361.

[Wil87] P. WILEY, *A parallel architecture comes of age at last*, IEEE Spectrum, 24 (1987), pp. 46-50.

[Win83] P. M. WINKLER, *Proof of the squashed cube conjecture*, Combinatorica, 3 (1983), pp. 135-139.

[Yao78] A. C. YAO, *On the loop switching addressing problem*, SIAM J. Comput., 7 (1978), pp. 515-523.

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1245-1262, December 1989

011

SIMPLE FAST ALGORITHMS FOR THE EDITING DISTANCE BETWEEN TREES AND RELATED PROBLEMS*

KAIZHONG ZHANG† AND DENNIS SHASHA‡

Abstract. Ordered labeled trees are trees in which the left-to-right order among siblings is significant. The distance between two ordered trees is considered to be the weighted number of edit operations (insert, delete, and modify) to transform one tree to another. The problem of approximate tree matching is also considered. Specifically, algorithms are designed to answer the following kinds of questions:

1. What is the distance between two trees?

2. What is the minimum distance between T_1 and T_2 when zero or more subtrees can be removed from T_2?

3. Let the pruning of a tree at node n mean removing all the descendants of node n. The analogous question for prunings as for subtrees is answered.

A dynamic programming algorithm is presented to solve the three questions in sequential time $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$ and space $O(|T_1| \times |T_2|)$ compared with $O(|T_1| \times |T_2| \times (depth(T_1))^2 \times (depth(T_2))^2)$ for the best previous published algorithm due to Tai [*J. Assoc. Comput. Mach.*, 26 (1979), pp. 422-433]. Further, the algorithm presented here can be parallelized to give time $O(|T_1| + |T_2|)$.

Key words. trees, editing distance, parallel algorithm, dynamic programming, pattern recognition

AMS(MOS) subject classifications. 68P05, 68Q25, 68Q20, 68R10

1. Motivation.

1.1. Applications. Ordered labeled trees are trees whose nodes are labeled and in which the left-to-right order among siblings is significant. As such they can represent grammar parses, image descriptions, and many other phenomena. Comparing such trees is a way to compare scenes, parses, and so on.

As an example, consider the secondary structure comparison problem for RNA. Because RNA is a single strand of nucleotides, it folds back onto itself into a shape that is topologically a tree (called its secondary structure). Each node of this tree contains several nucleotides. Nodes have colorful labels such as "bulge" and "hairpin." Various researchers [ALKBO], [BSSBWD], [DD] have observed that the secondary structure influences translation rates (from RNA to proteins). Because different sequences can produce similar secondary structures [DA], [SK], comparisons among secondary structures are necessary to understanding the comparative functionality of different RNAs. Previous methods for comparing multiple secondary structures of RNA molecules represent the tree structures as parenthesized strings [S88]. These have been recently converted to using our tree distance algorithms.

Currently we are implementing a package containing algorithms described in this paper and some other related algorithms. A preliminary version of the package is being used at the National Cancer Institute for the RNA comparison problem.

1.2. Algorithmic approach. The tree distance problem is harder than the string distance problem. Intuitively, here is why. In the string case, if $S_1[i] = S_2[j]$, then the

* Received by the editors August 5, 1987; accepted for publication (in revised form) February 12, 1989. This work was partially supported by the National Science Foundation under grant number DCR8501611 and by the Office of Naval Research under grant number N00014-85-K-0046.

† Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012 (zhang@csd2.nyu.edu). Present address, Department of Computer Science, Middlesex College, The University of Western Ontario, London, Ontario, Canada N6A 5B7.

‡ Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York, 10012 (shasha@nyu.edu).

distance between $S_1[1..i-1]$ and $S_2[1..j-1]$ is the same as between $S_1[1..i]$ and $S_2[1..j]$. The main difficulty in the tree case is that preserving ancestor relationships in the mapping between trees prevents the analogous implication from holding.

By introducing the distance between ordered forests and careful elimination of certain subtree-to-subtree distance calculations we are able to improve the time and space of best previous published algorithm [T]. Note that the improvement of space for this problem is extremely important in practical applications.

Besides improving on the time and space of the best previous algorithm [T], our algorithm is far simpler to understand and to implement. In style, it resembles algorithms for computing the distance between strings. In fact, the string distance algorithm is a special case of our algorithm when the input is a string.

2. Definitions.

2.1. Edit operations and editing distance between trees. Let us consider three kinds of operations. Changing node n means changing the label on n. Deleting a node n means making the children of n become the children of the parent of n and then removing n. Inserting is the complement of delete. This means that inserting n as the child of n' will make n the parent of a consecutive subsequence of the current children of n'. Figs. 1-3 illustrate these editing operations.

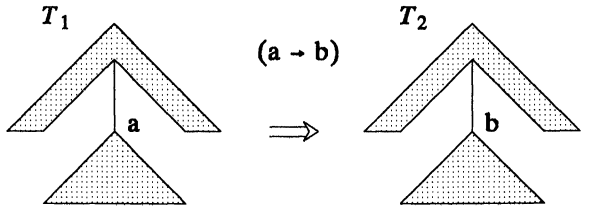


FIG. 1

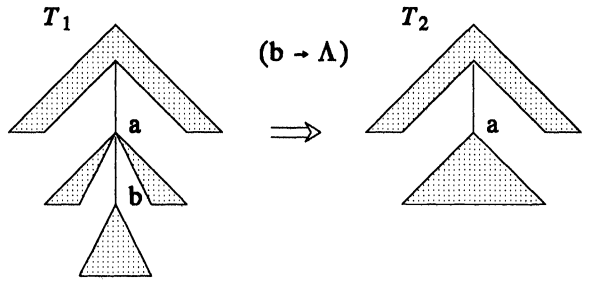


FIG. 2

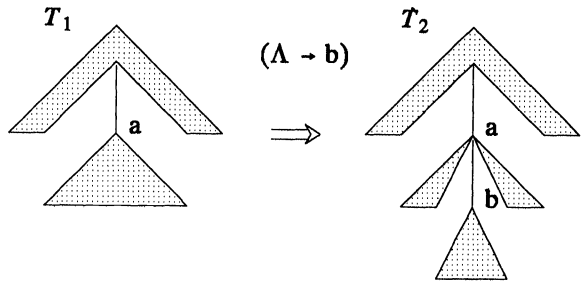


FIG. 3

(1) Change. To change one node label to another.

(2) Delete. To delete a node. (All children of the deleted node b become children of the parent a.)

(3) Insert. To insert a node. (A consecutive sequence of siblings among the children of a become the children of b.)

Following [WF] and [T], we represent an edit operation as a pair $(a, b) \neq (\Lambda, \Lambda)$, sometimes written as $a \to b$, where a is either Λ or a label of a node in tree T_1 and b is either Λ or a label of a node in tree T_2. We call $a \to b$ a change operation if $a \neq \Lambda$ and $b \neq \Lambda$; a delete operation if $b = \Lambda$; and an insert operation if $a = \Lambda$. Since many nodes may have the same label, this notation is potentially ambiguous. It could be made precise by identifying the nodes as well as their labels. However, in this paper, which node is meant will always be clear from the context.

Let S be a sequence $s_1, \cdots, s_k$ of edit operations. An S-derivation from A to B is a sequence of trees $A_0, \cdots, A_k$ such that $A = A_0$, $B = A_k$, and $A_{i-1} \to A_i$ via s_i for $1 \leqq i \leqq k$.

Let γ be a cost function that assigns to each edit operation $a \to b$ a nonnegative real number $\gamma(a \to b)$. This cost can be different for different nodes, so it can be used to give greater weights to, for example, the higher nodes in a tree than to lower nodes.

We constrain γ to be a distance metric. That is,

(i) $\gamma(a \to b) \geqq 0$; $\gamma(a \to a) = 0$

(ii) $\gamma(a \to b) = \gamma(b \to a)$; and

(iii) $\gamma(a \to c) \leqq \gamma(a \to b) + \gamma(b \to c)$.

We extend γ to the sequence S by letting $\gamma(S) = \sum_{i=1}^{i=|S|} \gamma(s_i)$. Formally the distance between T_1 and T_2 is defined as follows:

$$\delta(T_1, T_2) = \min \{\gamma(S) \mid S \text{ is an edit operation sequence taking } T_1 \text{ to } T_2\}.$$

The definition of γ makes δ a distance metric also.

2.2. Mapping. Let T_1 and T_2 be two trees with N_1 and N_2 nodes, respectively. Suppose that we have an ordering for each tree, then $T[i]$ means the ith node of tree T in the given ordering.

The edit operations give rise to a mapping that is a graphical specification of what edit operations apply to each node in the two trees (or two ordered forests). The mapping in Fig. 4 shows a way to transform T_1 to T_2. It corresponds to the sequence (delete (node with label c), insert (node with label c)).

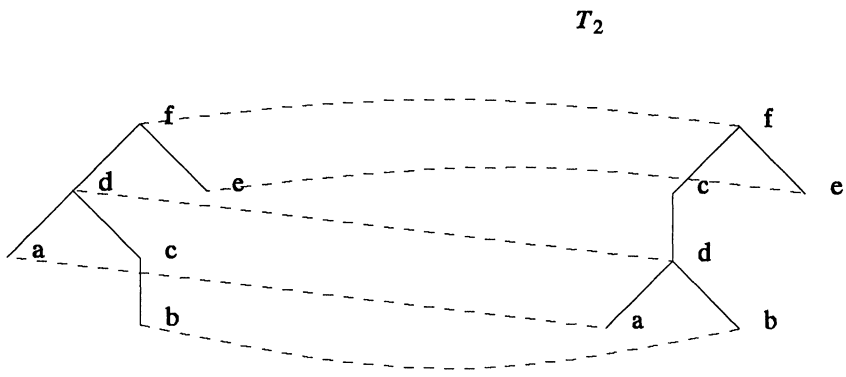


FIG. 4

Consider the diagram of a mapping in Fig. 4. A dotted line from $T_1[i]$ to $T_2[j]$ indicates that $T_1[i]$ should be changed to $T_2[j]$ if $T_1[i] \neq T_2[j]$, or that $T_1[i]$ remains unchanged if $T_1[i] = T_2[j]$. The nodes of T_1 not touched by a dotted line are to be deleted and the nodes of T_2 not touched are to be inserted. The mapping shows a way to transform T_1 to T_2.

Formally we define a triple (M, T_1, T_2) to be a mapping from T_1 to T_2, where M is any set of pair of integers (i, j) satisfying:[1]

(1) $1 \leqq i \leqq N_1, 1 \leqq j \leqq N_2$;

(2) For any pair of (i_1, j_1) and (i_2, j_2) in M,

(a) $i_1 = i_2$ if and only if $j_1 = j_2$ (one-to-one),

(b) $T_1[i_1]$ is to the left of $T_1[i_2]$ if and only if $T_2[j_1]$ is to the left of $T_2[j_2]$ (sibling order preserved),

(c) $T_1[i_1]$ is an ancestor of $T_1[i_2]$ if and only if $T_2[j_1]$ is an ancestor of $T_2[j_2]$ (ancestor order preserved).

We will use M instead of (M, T_1, T_2) if there is no confusion. Let M be a mapping from T_1 to T_2. Let I and J be the sets of nodes in T_1 and T_2, respectively, not touched by any line in M. Then we can define the cost of M:

$$\gamma(M) = \sum_{(i,j) \in M} \gamma(T_1[i] \to T_2[j]) + \sum_{i \in I} \gamma(T_1[i] \to \Lambda) + \sum_{j \in J} \gamma(\Lambda \to T_2[j]).$$

Mappings can be composed. Let M_1 be a mapping from T_1 to T_2 and let M_2 be a mapping from T_2 to T_3. Define

$$M_1 \circ M_2 = \{(i, j) \mid \exists k \text{ s.t. } (i, k) \in M_1 \text{ and } (k, j) \in M_2\}.$$

LEMMA 1. (1) *$M_1 \circ M_2$ is a mapping.*

(2) $\gamma(M_1 \circ M_2) \leqq \gamma(M_1) + \gamma(M_2)$.

Proof. Case (1) follows from the definition of mapping.

(2) Let M_1 be the mapping from T_1 to T_2. Let M_2 be the mapping from T_2 to T_3. Let $M_1 \circ M_2$ be the composed mapping from T_1 to T_3 and let I and J be the corresponding deletion and insertion sets. Three general situations occur. $(i, j) \in M_1 \circ M_2$, $i \in I$, or $j \in J$. In each case this corresponds to an editing operation $\gamma(x \to y)$ where x and y may be nodes or may be Λ. In all such cases, the triangle inequality on the distance metric γ ensures that $\gamma(x \to y) \leqq \gamma(x \to z) + \gamma(z \to y)$. □

The relation between a mapping and a sequence of edit operation is as follows.

LEMMA 2. *Given S, a sequence $s_1, \cdots, s_k$ of edit operations from T_1 to T_2, there exists a mapping M from T_1 to T_2 such that $\gamma(M) \leqq \gamma(S)$. Conversely, for any mapping M, there exists a sequence of editing operations such that $\gamma(S) = \gamma(M)$.*

Proof. The first part can be proved by induction on k. The base case is $k = 1$. This case holds, because any single editing operation preserves the ancestor and sibling relationships in the mapping. In the general case, let S_1 be the sequence $s_1, \cdots, s_{k-1}$ of edit operations. There exist a mapping M_1 such that $\gamma(M_1) \leqq \gamma(S_1)$. Let M_2 be the mapping for s_k. From Lemma 1, we have that

$$\gamma(M_1 \circ M_2) \leqq \gamma(M_1) + \gamma(M_2) \leqq \gamma(S).$$

To construct the sequence of editing operations, simply perform all the deletes indicated by the mapping (i.e., all nodes in T_1 having no lines attached to them are deleted), then all relabellings, then all inserts. □

[1] Note that our definition of mapping is different from the definition in [T]. We believe that our definition is more natural because it does not depend on any traversal ordering of the tree.

Hence, $\delta(T_1, T_2) = \min\{\gamma(M) | M$ is a mapping from T_1 to $T_2\}$.

There has been previous work on this problem. Tai [T] gave the best published algorithm for the problem. [Z83] is an improvement of [T], giving better sequential time and space than [T]. Our new algorithm is much simpler than [T] and [Z83], gives better time and space than both of them, and extends to related problems. The algorithm of Lu [L] does not solve this problem for trees of more than two levels.

3. A simple new algorithm. This algorithm, unlike [T], [L], and [Z83], will, in its intermediate steps, consider the distance between two ordered forests. At first sight one may think that this will complicate the work, but it will in fact make matters easier.

We use a postorder numbering of the nodes in the trees. In the postordering, $T_1[1..i]$ and $T_2[1..j]$ will generally be forests as in Fig. 5. (The edges are those in the subgraph of the tree induced by the vertices.) Fortunately, the definition of mapping for ordered forests is the same as for trees.

3.1. Notation. Let $T[i]$ be the ith node in the tree according to the left-to-right postorder numbering. $l(i)$ is the number of the leftmost leaf descendant of the subtree rooted at $T[i]$. When $T[i]$ is a leaf, $l(i) = i$. The parent of $T[i]$ is denoted $p(i)$. We define $p^0(i) = i, p^1(i) = p(i), p^2(i) = p(p^1(i))$, and so on. Let $anc(i) = \{p^k(i) | 0 \leqq k \leqq depth(i)\}$.

$T[i..j]$ is the ordered subforest of T induced by the nodes numbered i to j inclusive (Fig. 5). If $i > j$, then $T[i..j] = \varnothing$. $T[1..i]$ will be referred to as $forest(i)$, when the tree T referred to is clear. $T[l(i)..i]$ will be referred to as $tree(i)$. $Size(i)$ is the number of nodes in $tree(i)$.

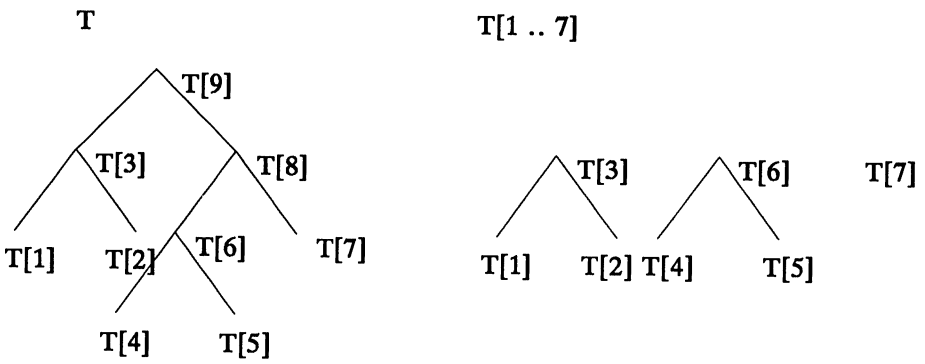


FIG. 5

The distance between $T_1[i'..i]$ and $T_2[j'..j]$ is denoted $forestdist(T_1[i'..i], T_2[j'..j])$ or $forestdist(i'..i, j'..j)$ if the context is clear. We use a more abbreviated notation for certain special cases. The distance between $T_1[1..i]$ and $T_2[1..j]$ is sometimes denoted $forestdist(i, j)$. The distance between the subtree rooted at i and the subtree rooted at j is sometimes denoted $treedist(i, j)$.

3.2. New algorithm. We first present three lemmas and then give our new algorithm. Recall that $anc(i) = \{p^k(i) | 0 \leqq k \leqq depth(i)\}$.

LEMMA 3. (i) $forestdist(\varnothing, \varnothing) = 0$.

(ii) $forestdist(T_1[l(i_1)..i], \varnothing) = forestdist(T_1[l(i_1)..i-1], \varnothing) + \gamma(T_1[i] \to \Lambda)$.

(iii) $forestdist(\varnothing, T_2[l(j_1)..j]) = forestdist(\varnothing, T_2[l(j_1)..j-1]) + \gamma(\Lambda \to T_2[j])$

where $i_1 \in anc(i)$ *and* $j_1 \in anc(j)$.

Proof. Case (i) requires no edit operation. In (ii) and (iii), the distances correspond to the cost of deleting or inserting the nodes in $T_1[l(i_1)..i]$ and $T_2[l(j_1)..j]$), respectively. □

LEMMA 4. *Let* $i_1 \in anc(i)$ *and* $j_1 \in anc(j)$. *Then*

$$forestdist(l(i_1)..i, l(j_1)..j) = \min\begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda), \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]), \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ \quad + forestdist(l(i)..i-1, l(j)..j-1) \\ \quad + \gamma(T_1[i] \to T_2[j]). \end{cases}$$

Proof. We compute $forestdist(l(i_1)..i, l(j_1)..j)$ for $l(i_1) \leqq i \leqq i_1$ and $l(j_1) \leqq j \leqq j_1$. We are trying to find a minimum-cost map M between $forest(l(i_1)..i)$ and $forest(l(j_1)..j)$. The map can be extended to $T_1[i]$ and $T_2[j]$ in three ways.

(1) $T_1[i]$ is not touched by a line in M. Then $(i, \Lambda) \in M$. So, $forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda)$.

(2) $T_2[j]$ is not touched by a line in M. Then $(\Lambda, j) \in M$. So, $forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j])$.

(3) $T_1[i]$ and $T_2[j]$ are both touched by lines in M. Then $(i, j) \in M$. Here is why. Suppose (i, k) and (h, j) are in M. If $l(i_1) \leqq h \leqq l(i)-1$, then i is to the right of h so k must be to the right of j by the sibling condition on mappings. This is impossible in $forest(l(j_1)..j)$. Similarly, if i is a proper ancestor of h, then k must be a proper ancestor of j by the ancestor condition on mappings. This too is impossible. So, $h = i$. By symmetry, $k = j$ and $(i, j) \in M$.

Now, by the ancestor condition on mapping, any node in the subtree rooted at $T_1[i]$ can only be touched by a node in the subtree rooted at $T_2[j]$. Hence,

$$forestdist(l(i_1)..i, l(j_1)..j) = forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + forestdist(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \to T_2[j]).$$

Figure 6 shows the situation.

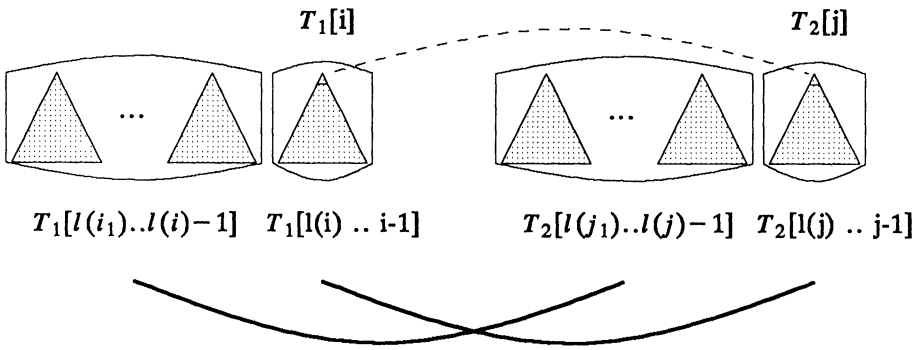


FIG. 6. *Case* (3) *of Lemma* 4.

Since these three cases express all the possible mappings yielding *forestdist*$(l(i_1)..i, l(j_1)..j)$, we take the minimum of these three costs. Thus,

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda) \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]) \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ \quad + forestdist(l(i)..i-1, l(j)..j-1) \\ \quad + \gamma(T_1[i] \to T_2[j]). \end{cases}$$

□

LEMMA 5. *Let* $i_1 \in anc(i)$ *and* $j_1 \in anc(j)$. *Then*

(1) *If* $l(i) = l(i_1)$ *and* $l(j) = l(j_1)$

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda), \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]), \\ forestdist(l(i_1)..i-1, l(j_1)..j-1) + \gamma(T_1[i] \to T_2[j]). \end{cases}$$

(2) *If* $l(i) \neq l(i_1)$ *or* $l(j) \neq l(j_1)$ (i.e., *otherwise*)

$$forestdist(l(i_1)..i, l(j_1)..j) = \min \begin{cases} forestdist(l(i_1)..i-1, l(j_1)..j) + \gamma(T_1[i] \to \Lambda), \\ forestdist(l(i_1)..i, l(j_1)..j-1) + \gamma(\Lambda \to T_2[j]), \\ forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) \\ \quad + treedist(i, j). \end{cases}$$

Proof. By Lemma 4, if $l(i) = (i_1)$ and $l(j) = l(j_1)$ then, since *forestdist*$(l(i_1)..l(i) - 1, l(j_1)..l(j)-1) = forestdist(\emptyset, \emptyset) = 0$, (1) follows immediately.

Because the distance is the cost of a minimal cost mapping, we know *forestdist*$(l(i_1)..i, l(j_1)..j) \leqq forestdist(l(i_1)..l(i)-1, l(j_1)..l(j)-1) + treedist(i, j)$ since the latter formula represents a particular (and therefore possibly suboptimal) mapping of *forest*$(l(i_1)..i)$ to *forest*$(l(j_1)..j)$. For the same reason, *treedist*$(i, j) \leqq$ *forestdist*$(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \to T_2[j])$. Lemma 4 and these two inequalities imply that the substituting of *treedist*(i, j) for *forestdist*$(l(i)..i-1, l(j)..j-1) + \gamma(T_1[i] \to T_2[j])$ in (2) is correct. (See Fig. 7.) □

Lemma 5 has three important implications:

First, the formulas it yields suggest that we can use a dynamic programming style algorithm to solve the tree distance problem.

Second, from (2) of Lemma 5 we observe that to compute *treedist*(i_1, j_1) we need in advance almost all values of *treedist*(i, j) where i_1 is the root of a subtree containing i and j_1 is the root of a subtree containing j. This suggests a bottom-up procedure for computing all subtree pairs.

Third, from (1) in Lemma 5 we can observe that when i is in the path from $l(i_1)$ to i_1 and j is in the path from $l(j_1)$ to j_1, we do not need to compute *treedist*(i, j) separately. These subtree distances can be obtained as a byproduct of computing *treedist*(i_1, j_1).

These implications lead to the following definition and then our new algorithm. Let us define the set *LR_keyroots* of tree T as follows:

$$LR\_keyroots(T) = \{k \mid \text{there exists no } k' > k \text{ such that } l(k) = l(k')\}.$$

That is, if k is in *LR_keyroots*(T) then either k is the root of T or $l(k) \neq l(p(k))$, i.e., k has a left sibling. Intuitively, this set will be the roots of all the subtrees of tree T that need separate computations.

Consider trees T_1 and T_2 in Fig. 4. From the above definition we can see that *LR_keyroots*$(T_1) = \{3, 5, 6\}$ and *LR_keyroots*$(T_2) = \{2, 5, 6\}$.

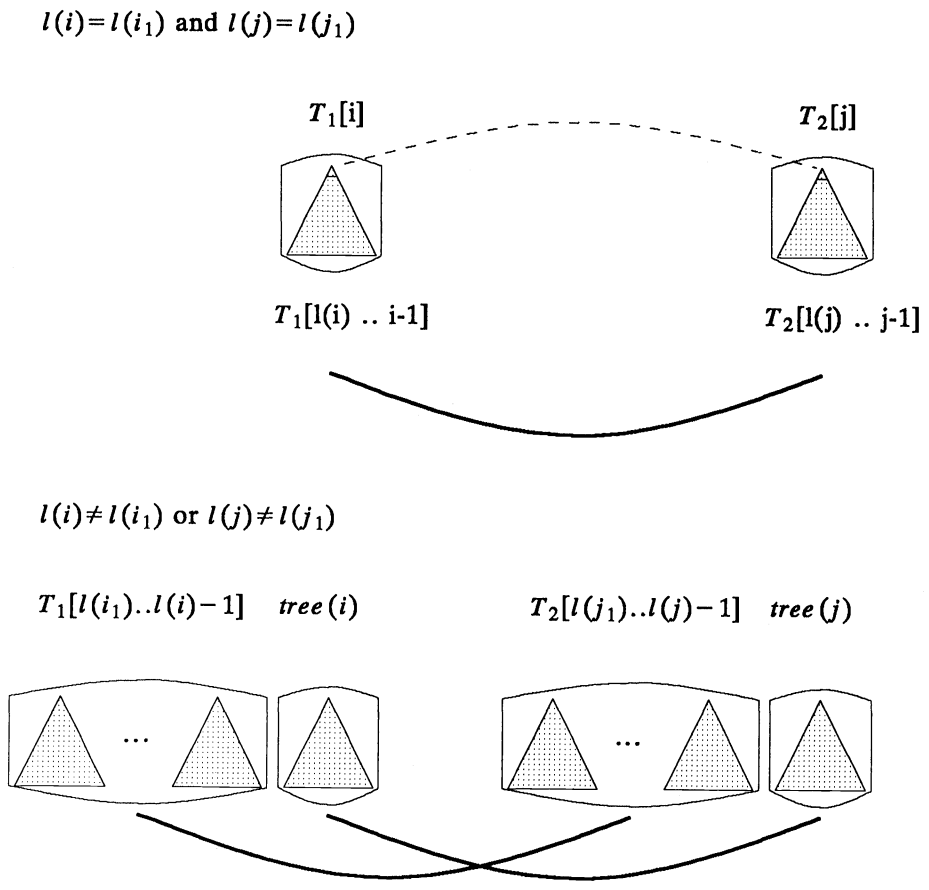


FIG. 7. *The two situations of Lemma* 5.

It is easy to see that there is a linear time algorithm to compute the function $l(\ )$ and the set *LR_keyroots*. We can also assume that the result is in array l and *LR_keyroots*. Furthermore, in array *LR_keyroots* the order of the elements is in increasing order.

We are now ready to give our new simple algorithm.

Input: Tree T_1 and T_2.
Output: *Tree_dist*(i, j), where $1 \leqq i \leqq |T_1|$ and $1 \leqq j \leqq |T_2|$.
Preprocessing
(To compute $l(\ )$, *LR_keyroots*1 and *LR_keyroots*2)
Main loop
  for $i' := 1$ to $|LR\_keyroots(T_1)|$
    for $j' := 1$ to $|LR\_keyroots(T_2)|$
      $i = LR\_keyroots1[i']$;
      $j = LR\_keyroots2[j']$;
      Compute *treedist*(i, j);

We use dynamic programming to compute *treedist*(i, j). The forestdist values computed and used here are put in a temporary array that is freed once the corresponding treedist is computed. The treedist values are put in the permanent treedist array.

The computation of *treedist*(i, j).

```
forestdist(∅, ∅) = 0;
for i1 := l(i) to i
  forestdist(T1[l(i)..i1], ∅) = forestdist(T1[l(i)..i1 − 1], ∅) + γ(T1[i1] → Λ)
for j1 := l(j) to j
  forestdist(∅, T2[l(j)..j1]) = forestdist(∅, T2[l(j)..j1 − 1]) + γ(Λ → T2[j1])
for i1 := l(i) to i
  for j1 := l(j) to j
    if l(i1) = l(i) and l(j1) = l(j) then
      forestdist(T1[l(i)..i1], T2[l(j)..j1)] = min {
        forestdist(T1[l(i)..i1 − 1], T2[l(j)..j1]) + γ(T1[i1] → Λ),
        forestdist(T1[l(i)..i1], T2[l(j)..j1 − 1]) + γ(Λ → T2[j1]),
        forestdist(T1[l(i)..i1 − 1], T2[l(j)..j1 − 1]) + γ(T1[i1] → T2[j1])}
      treedist(i1, j1) = forestdist(T1[l(i)..i1], T2[l(j)..j1])/* put in permanent
        array */
    else
      forestdist(T1[l(i)..i1], T2[l(j)..j1]) = min {
        forestdist(T1[l(i)..i1 − 1], T2[l(j)..j1]) + γ(T1[i1] → Λ),
        forestdist(T1[l(i)..i1], T2[l(j)..j1 − 1]) + γ(Λ → T2[j1]),
        forestdist(T1[l(i)..l(i1) − 1], T2[l(j)..l(j1) − 1]) + treedist(i1, j1)}
```

THEOREM 1. *The basic algorithm is correct.*

Proof. We will prove that for any pair (i, j) such that $i \in LR\_keyroots(T_1)$ and $j \in LR\_keyroots(T_2)$, the following invariants holds.

tree_dist(3, 2)

0	1
1	0
2	1

tree_dist(3, 5)

0	1
1	1
2	2

tree_dist(3, 6)

0	1	2	3	4	5	6
1	1	1	2	3	4	5
2	2	2	2	2	3	4

tree_dist(5, 2)

0	1
1	1

tree_dist(5, 5)

0	1
1	0

tree_dist(5, 6)

0	1	2	3	4	5	6
1	1	2	3	4	4	5

tree_dist(6, 2)

0	1
1	1
2	1
3	2
4	3
5	4
6	5

tree_dist(6, 5)

0	1
1	1
2	2
3	3
4	4
5	4
6	5

tree_dist(6, 6)

0	1	2	3	4	5	6
1	0	1	2	3	4	5
2	1	0	1	2	3	4
3	2	1	2	3	4	5
4	3	2	1	2	3	4
5	4	3	2	3	2	3
6	5	4	3	3	3	2

tree_dist

0	1	2	3	1	5
1	0	2	3	1	5
2	1	2	2	2	4
3	3	1	2	4	4
1	1	3	4	0	5
5	5	3	3	5	2

FIG. 8. *The result of computation for T_1 and T_2 in Fig. 4.*

(1) Immediately before the computation of $treedist(i, j)$, all distances $treedist(i_1, j_1)$, where $l(i) \leqq i_1 \leqq i$ and $l(j) \leqq j_1 \leqq j$ and either $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$, are available. In other words, $treedist(i_1, j_1)$ is available if i_1 is in the subtree of $tree(i)$ but not in the path from $l(i)$ to i and j_1 is in the subtree of $tree(j)$ but not in the path from $l(j)$ to j.

(2) Immediately after the computation of $treedist(i, j)$, all distances $treedist(i_1, j_1)$, where $l(i) \leqq i_1 \leqq i$ and $l(j) \leqq j_1 \leqq j$ are available.

We first show that if (1) is true then (2) is true. From Lemma 5 we know that all required subtree-to-subtree distances are available. (We need all $treedist(i_1, j_1)$ such that $l(i) \leqq i_1 \leqq i$ and $l(j) \leqq j_1 \leqq j$ and either $l(i) \neq l(i_1)$ or $l(j) \neq l(j)$, and by (1) all these distances are available.) We compute each $treedist(i_1, j_1)$, where $l(i_1) = l(i)$ and $l(j_1) = l(j)$ in the if part and add it to the permanent treedist array. So, (2) holds.

Let us show that (1) always holds. Suppose $l(i_1) \neq l(i)$. Let i_1' be the lowest ancestor of i_1 such that $i_1' \in LR\_keyroots(T_1)$. Since $l(i_1') = l(i_1) \neq l(i)$, $i_1' \neq i$. Since $i \in LR\_keyroots(T_1)$, $i_1' \leqq i$. So $i_1' < i$. Let j_1' be the lowest ancestor of j_1 such that $j_1' \in LR\_keyroots(T_2)$. Since $j \in LR\_keyroots(T_2)$, $j_1' \leqq j$. Hence $i_1' + j_1' < i + j$. This means that $treedist(i_1', j_1')$ will have already been computed before $treedist(i, j)$ because in the main loop $LR\_keyroots1$ and $LR\_keyroots2$ are in increasing order. Hence $treedist(i_1, j_1)$ is available after the computation of $treedist(i_1', j_1')$. □

As an example, consider tree T_1 and T_2 in Fig. 4. For simplicity, assume that all insert, delete, and change (of labels) operations will cost one. Figure 8 shows the result of applying our new algorithm to T_1 and T_2. The matrix below $tree\_dist(i, j)$ is the result of temporary array produced by the computation of $tree\_dist(i, j)$. (Out of 36 possible *tree_dist* arrays, only nine—those corresponding to pairs of keyroots—are explicitly computed.) The matrix below *tree_dist* is the final result. The value in the lower right corner (2) is the distance between T_1 and T_2.

4. Some aspects of our algorithm.

4.1. Complexity.

LEMMA 6. $|LR\_keyroots(T)| \leqq |leaves(T)|$.

Proof. We will prove that for any $i, j \in LR\_keyroots(T)$, $l(i) \neq l(j)$.

Let $i, j \in LR\_keyroots(T)$ and $i < j$. If $l(i) = l(j)$ from $i < j$ we know that i is in the path from $l(j)$ to j. By the definition of $l(j)$, i has no *left_sibling*. This contradicts the assertion that $i \in LR\_keyroots(T)$. Hence each leaf is the leftmost descendant of at most one member of $LR\_keyroots(T)$. So, $|LR\_keyroots(T)| \leqq |leaves(T)|$. □

Because not all subtree-to-subtree distances need be computed, the number of such calculation a node participates in is less than its depth. Instead, it is the node's *collapsed depth*:

$$LR\_colldepth(i) = |anc(i) \cap LR\_keyroots(T)|.$$

We define the collapsed depth of tree T as follows:

$$LR\_colldepth(T) = \max LR\_colldepth(i).$$

By the definition and Lemma 6 we can see that $LR\_colldepth(i) \leqq \min(depth(T), leaves(T))$ for $1 \leqq i \leqq |T|$. Hence $LR\_colldepth(T) \leqq \min(depth(T), leaves(T))$.

LEMMA 7.

$$\sum_{i=1}^{i=|LR\_keyroots(T)|} Size(i) = \sum_{j=1}^{j=N_1} |LR\_colldepth(j)|.$$

Proof. Consider when node j is counted in the first summation: in the subtrees corresponding to each of its ancestors that is in $LR\_keyroots(T)$. By the definition of $LR\_colldepth(\ )$, j is counted $LR\_colldepth(j)$ times.

THEOREM 2. *The time complexity is* $O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2)))$. *The space complexity is* $O(|T_1| \times |T_2|)$.

Proof. Let us consider the space complexity first. We use a permanent array for *treedist* and a temporary array for *forestdist.* Each of these two arrays requires space $O(|T_1| \times |T_2|)$.

Consider the time complexity of our algorithm. The preprocessing takes linear time. The subtree distance dynamic programming algorithm takes $Size(i) \times Size(j)$ for the subtree rooted at $T_1[i]$ and the subtree rooted at $T_2[j]$. We have a main loop that calls this subroutine several times. So the time is:

$$\sum_{i=1}^{i=|LR\_keyroots(T_1)|} \sum_{j=1}^{j=|LR\_keyroots(T_2)|} Size(i) \times Size(j)$$

$$= \sum_{i=1}^{i=|LR\_keyroots(T_1)|} Size(i) \times \sum_{j=1}^{j=|LR\_keyroots(T_2)|} Size(j).$$

By Lemma 6, the above equals

$$\sum_{i=1}^{i=N_1} LR\_colldepth(i) \times \sum_{j=1}^{j=N_2} LR\_colldepth(j).$$

This is less than

$$|T_1| \times |T_2| \times LR\_colldepth(T_1) \times LR\_colldepth(T_2).$$

By the definition of $LR\_colldepth$, we have that the time complexity is

$$O(|T_1| \times |T_2| \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2))). \quad \square$$

These time and space complexities are an improvement over the $O(|T_1| \times |T_2| \times depth(T_1)^2 \times depth(T_2)^2)$ time and space complexity of [T].

Note. If we use a right-to-left postorder numbering for tree nodes and define similar functions $r(i)$, $RL\_keyroots(T)$ and $RL\_colldepth(i)$, we can have the same result as above. The complexity will be $\sum_{i=1}^{i=N_1} RL\_colldepth(i) \times \sum_{j=1}^{j=N_2} RL\_colldepth(j)$.

Clearly, using the left-to-right or right-to-left postorder numberings give the same worst-case time complexity. However, in practice it may be beneficial to choose the ordering that gives the lower of the following two products: $\sum_{i=1}^{i=N_1} LR\_colldepth(i) \times \sum_{j=1}^{j=N_2} LR\_colldepth(j)$ and $\sum_{i=1}^{i=N_1} RL\_colldepth(i) \times \sum_{j=1}^{j=N_2} RL\_colldepth(j)$.

4.2. Mapping. It is natural to ask for a mapping that yields the distance computed. Also given two trees, we may ask, what is the largest common substructure of these two trees? This is analogous to the longest common substring problem for strings. We can find the mapping in the same time and space complexity as finding the distance, although we do not give the details here. The mapping is produced by our toolkit.

4.3. Parallel implementation. A straightforward transformation of our algorithm to a parallel one yields an algorithm with time complexity $O(N_1 + N_2)$ whereas [T] and [Z83] have time complexity $O((N_1 + N_2) \times (depth(T_1) + depth(T_2)))$. Our algorithm uses $O(\min(|T_1|, |T_2|) \times leaves(T_1) \times leaves(T_2))$ processors.[2]

[2] Actually, by controlling the starting point of each *treedist* computation more carefully, we can reduce the processor bound to $O(\min(|T_1|, |T_2|) \times \min(depth(T_1), leaves(T_1)) \times \min(depth(T_2), leaves(T_2))$. The algorithm is more complicated however.

The algorithm computes in "waves" for all subtree pairs $tree(i)$ and $tree(j)$, where $i \in LR\_keyroots(T_1)$ and $j \in LR\_keyroots(T_2)$, simultaneously. We start at wave 0. At wave k, for each such subtree pair $tree(i)$ and $tree(j)$, compute $forestdist(l(i)..i_1, l(j)..j_1)$, where $(i_1 - l(i)) + (j_1 - l(j)) = k$.

We now present the parallel algorithm in detail. (When the PARBEGIN-PAREND construct surrounds one or more for loops, it means that every setting of the iterators in the enclosed for loops can be executed in parallel. The semantics are those of the sequential program ignoring this construct.)

In the algorithm $dist[i, j]$ is the array for the computation of $treedist(i, j)$. Therefore $dist[i, j][p, q]$ is the distance $forestdist(l(i)..p, l(j)..q)$ and is the p, qth member of the array computing $treedist(i, j)$.

```
ALGORITHM PARALLEL DISTANCE.
begin
PARBEGIN
  for i' := 1 to |LR_keyroots(T1)|
    for j' := 1 to |LR_keyroots(T2)|
      i := LR_keyroots1[i']
      j := LR_keyroots2[j']
      dist[i,j][l(i)-1, l(j)-1] := 0/* initializes temporary array for each tree
dist */
PAREND
for k := 0 to N-1
  PARBEGIN
    for i' := 1 to |LR_keyroots(T1)|
      for j' := 1 to |LR_keyroots(T2)|
        i := LR_keyroots1[i']
        j := LR_keyroots2[i']
        dist[i,j][l(i)+k, l(j)-1]
          := dist[i,j][l(i)+k-1, l(j)-1] + γ(T1[l[i]+k] → Λ)
PAREND
for k := 0 to M-1
  PARBEGIN
    for i' := 1 to |LR_keyroots(T1)|
      for j' := 1 to |LR_keyroots(T2)|
        i := LR_keyroots1[i']
        j := LR_keyroots2[j']
        dist[i,j][l(i)-1, l(j)+k]
          := dist[i,j][l(i)-1, l(j)+k-1] + γ(Λ → T1[l[j]+k])
  PAREND
for k := 0 to N+M-2
  PARBEGIN
    for i' := 1 to |LR_keyroots(T1)|
      for j' := 1 to |LR_keyroots(T2)|
        i := LR_keyroots1[i']
        j := LR_keyroots2[j']
        for i1, j1 satisfying i1 - l(i) + j1 - l(j) = k and l(i) ≦ i1 ≦ i, l(j) ≦ j1 ≦ j
          if l(i) = l(i1) and l(j) = l(j1) then
            dist[i,j][i1, j1] := min {
              dist[i,j][i1-1, j1] + γ(T1[i1] → Λ)
```

```
            dist[i, j][i1, j1 − 1] + γ(Λ → T2[j1])
            dist[i, j][i1 − 1, j1 − 1] + γ(T1[i1] → T2[j1])
          }
          treedist(i1, j1) := dist[i, j][i1, j1]
        else
          dist[i, j][i1, j1] := min {
            dist[i, j][i1 − 1, j1] + γ(T1[i1] → Λ)
            dist[i, j][i1, j1 − 1] + γ(Λ → T2[j1])
            dist[i, j][l(i1) − 1, l(j1) − 1] + treedist[i1, j1]
          }
end
```

It is easy to see that in the above algorithm all the terms, except $treedist[i_1, j_1]$, are available whenever needed. We now show that $treedist[i_1, j_1]$ is available whenever we use it. Our argument is similar to the one we used in the sequential case.

Note that we compute all terms such that $(i_1 - l(i)) + (j_1 - l(j)) = k$ together. During that computation, all terms such that $(i_1 - l(i)) + (j_1 - l(j)) < k$ are available. So, when we need item $treedist[i_1, j_1]$, either $l(i_1) > l(i)$ or $l(j_1) > l(j)$. Let i_2 be the lowest ancestor of i_1 such that $i_2 \in LR\_keyroots(T_1)$. Let j_2 be the lowest ancestor of j_1 such that $j_2 \in LR\_keyroots(T_2)$. Since $l(i_1) = l(i_2)$ and $l(j_1) = l(j_2)$ we know either $l(i_2) > l(i)$ or $l(j_2) > l(j)$. Therefore, $(i_1 - l(i_2)) + (j_1 - l(j_2)) < (i_1 - l(i)) + (j_1 - l(j)) = k$. Hence $treedist[i_1, j_1]$ was already computed in the computation of $dist[i_2, j_2][i_1, j_1]$ and put into the permanent tree distance array. This settles correctness.

THEOREM 3. *The Parallel Distance Algorithm has time complexity* $O(|T_1| + |T_2|)$.

Proof. By simple analysis of the for loop. □

4.4. From trees to strings. Strings are an important special case of trees. This algorithm is a generalization of the natural dynamic programming algorithms on strings in two senses: time complexity and algorithmic style.

First, we consider the time complexity. Since a string has only one leaf, applying our algorithms to strings yields a time complexity of $O(|T_1| \times |T_2|)$. This is the same as that of the best available algorithm for the general problem of string distance.

Second, we consider the algorithm itself. For a string S, $LR\_keyroots(S) = \{root\}$. So the main loop will only have one iteration. In the dynamic programming subroutine, since $l(i) = 1$, we will never come to the case $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$. So if we change i to $|T_1|$, j to $|T_2|$, $l(i)$ to one, $l(j)$ to one, delete the main loop and delete the case where $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$, we will have exactly the string distance algorithm.

5. The general technique applied to approximate tree matching. Many problems in strings can be solved with dynamic programming. Similarly, our algorithm not only applies to tree distance but also provides a way to do dynamic programming for a variety of tree problems with the same time complexity. In this section we show how to apply this general paradigm to approximate tree matching.

5.1. Algorithm template. Here is the general form of the algorithm (assuming a left-to-right postorder traversal):

```
preprocessing
main loop
  for i' := 1 to |LR_keyroots(T1)|
    for j' := 1 to |LR_keyroots(T2)|
      i = LR_keyroots1[i'];
      j = LR_keyroots2[j'];
```

```
        compute Tree_D(i, j);
subroutine for Tree_D(i, j)
  empty_initialization
  for i1 := l(i) to i
    left_initialization
  for j1 := l(j) to j
    right_initialization
  for i1 := l(i) to i
    for j1 := l(j) to j
      if l(i1) = l(i) and l(j1) = l(j) then
        general_if_computation
        Tree_D(i1, j1) = Forest_D(T1[l(i)..i1], T2[l(j)..j1]);
      else
        general_else_computation
```

5.2. Approximate tree matching. We first consider approximate string matching [S80], [U83], [U85], [LV]. We will then give two natural generalizations of approximate string matching to approximate tree matching. This will also be a generalization of the exact tree matching algorithm as found in Hoffmann and O'Donnell [HO].

The approximate string matching problem is the following. Given two strings *STEXT* and *SPAT*, the problem is to compute, for each i, $SD[i, SPAT] = \min_j \{D(STEXT[j..i], SPAT)\}$, where $1 \leqq j \leqq i+1$ and D is the string distance metric. In other words, the problem is to compute, for each i, the minimum number of editing operations between the "pattern" string $SPAT[1..|PAT|]$ and the "text string" $STEXT[1..i]$ where any prefix can be removed from $STEXT[1..i]$. (Intuitively, the algorithm finds the "occurrence" in *TEXT* that most closely matches *PAT*.)

To extend this problem to trees, we must generalize the notion of removing a prefix. For us, a prefix will mean a collection of subtrees.

We first define two operations at a node.

Removing at node $T[i]$ means removing the subtree rooted at $T[i]$. In other words, delete $T[l(i)..i]$. (See Fig. 9.)

Pruning at node $T[i]$ means removing all the descendants of $T[i]$. In other words, delete $T[l(i)..i-1]$. (Thus, a pruning never eliminates the entire tree.) (See Fig. 10.)

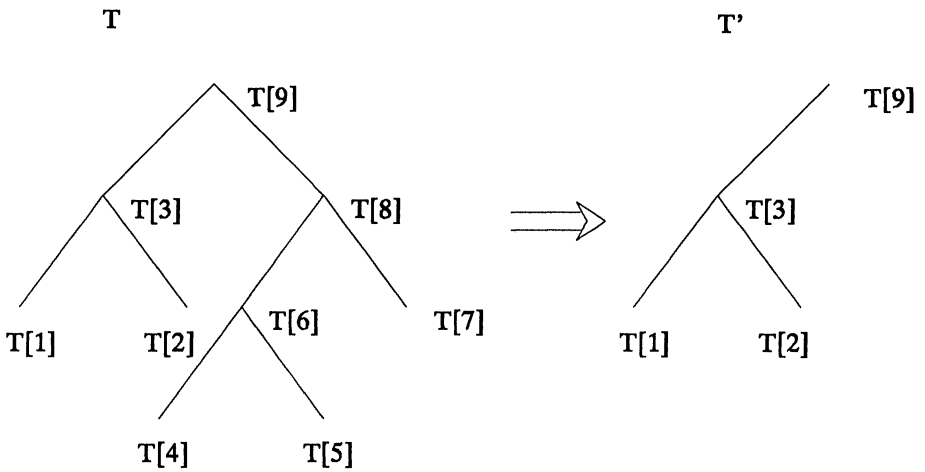


FIG. 9. Remove subtree rooted at T[8].

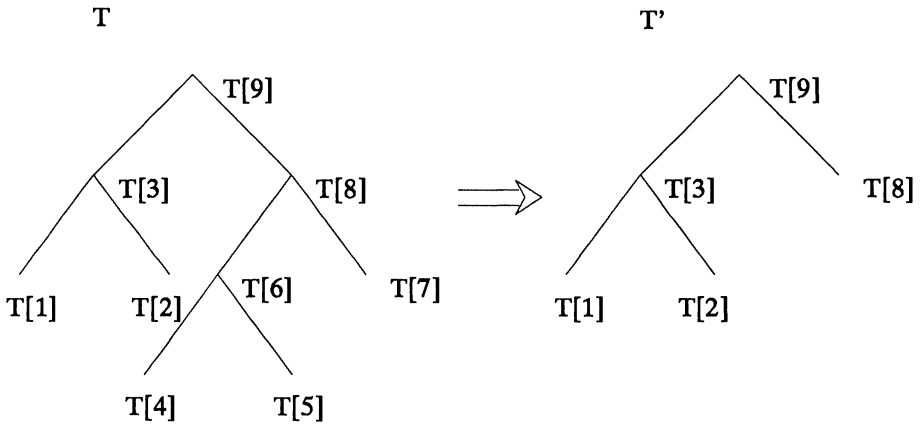


FIG. 10. *Pruning at T[8]—remove all its proper descendants.*

Assume an ordering for tree T. Define a subtree set $S(T)$ as follows: $S(T)$ is a set of numbers satisfying

(1) $i \in S(T)$ implies that $1 \leqq i \leqq |T|$

(2) $i, j \in S(T)$ implies that neither is an ancestor of the other.

Define $R(T, S(T))$ to be the tree T with removing at all nodes in $S(T)$.
Define $P(T, S(T))$ to be the tree T with pruning at all nodes in $S(T)$.

Now we can give the definition of approximate tree matching. Given tree T and PAT, for each i, we want to calculate

$$DR(T[l(i)..i, PAT) = \min_S \{treedist(R(T[l(i)..i], S(T[l(i)..i])), PAT)\}.$$

$$DP(T[l(i)..i, PAT) = \min_S \{treedist(P(T[l(i)..i], S(T[l(i)..i])), PAT)\}.$$

The minimum here is over all possible subtree sets $S(T[l(i)..i])$. We consider each generalization in turn.

5.2.1. Remove any number of subtrees from *TEXT* tree. The problem is as follows. Given trees T_1 and T_2, we want to know what is the minimum distance between $T_1[l(i)..i]$ and T_2 when zero or more subtrees can be removed from $T_1[l(i)..i]$.

Let $F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1])$ denote the minimum distance between forest $T_1[l(i)..i_1]$ and $T_2[l(j)..j_1]$ with zero or more subtrees removed from $T_1[l(i)..i_1]$. Let $T\_DR(i, j)$ denote the minimum distance between tree $T_1[l(i)..i]$ and $T_2[l(j)..j]$ with zero or more subtrees removed from $T_1[l(i)..i]$. We write the algorithm in the form suggested by the algorithm template.

ALGORITHM SUBTREE REMOVAL.
empty_initialization:
$F\_DR(\varnothing, \varnothing) = 0$
left_initialization:
$F\_DR(T_1[l(i)..i_1], \varnothing) = 0$
right_initialization:
$F\_DR(\varnothing, T_2[l(j)..j_1]) = F\_DR(\varnothing, T_2[l(j)..j_1 - 1]) + \gamma(\Lambda \to T_2[j_1])$
general_if_computation
/* applies if $l(i_1) = l(i)$ and $l(j_1) = l(j)$*/
$F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$

$F\_DR(\varnothing, T_2[l(j)..j_1])$,
$F\_DR(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda)$,
$F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1])$,
$F\_DR(T_1[l(i)..i_1-1], T_2[l(j)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])\}$
/* put the derived *treedist* in the permanent array, as specified by template */
general_else_computation
$F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$
$F\_DR(T_1[l(i)..l(i_1)-1], T_2[l(j)..j_1])$,
$F\_DR(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda)$,
$F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1])$,
$F\_DR(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1]) + T\_DR(i_1, j_1)\}$

LEMMA 8. *Algorithm Subtree Removal is correct.*

Proof. First we show that the initialization is correct. The empty_initialization and the right_initialization are the same as in the tree distance algorithm. The left_initialization $F\_DR(T_1[l(i)..i_1], \varnothing) = 0$ is correct, because we can remove all of $T_1[l(i)..i_1]$.

For the general term $F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1])$, we ask first whether or not the subtree $T_1[l(i_1)..i_1]$ is removed. If it is removed, then the distance should be $F\_DR(T_1[l(i)..l(i_1)-1], T_2[l(j)..j_1])$. Otherwise, consider the mapping between $T_1[l(i)..i_1]$ and $T_2[l(j)..j_1]$ after we perform an optimal removal of subtrees of $T_1[l(i)..i_1]$. Now we have the same three cases as in Lemma 4. Hence the general expression should be the minimum of these four terms:

$F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$
$F\_DR(T_1[l(i)..l(i_1)-1], T_2[l(j)..j_1])$,
$F\_DR(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda)$,
$F\_DR(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1])$,
$F\_DR(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1])$
$+ F\_DR(T_1[l(i_1)..i_1-1], T_2[l(j_1)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])\}$

As in Lemma 5, this specializes to the general_if_computation and the general_else_computation given in the algorithm. □

5.2.2. Prune at any number of nodes from the *TEXT* tree. Given trees T_1 and T_2, we want to know what is the minimum distance between $T_1[l(i)..i]$ and T_2 when there have been zero or more prunings at nodes of $T_1[l(i)..i]$.

Let $F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1])$ denote the minimum distance between *forest* $T_1[l(i)..i_1]$ and $T_2[l(j)..j_1]$ with zero or more pruning from $T_1[l(i)..i_1]$. Let $T\_DP(i, j)$ denote the minimum distance between *tree* $T_1[l(i)..i]$ and $T_2[l(j)..j]$ with zero or more prunings from $T_1[l(i)..i]$. The following initialization and general term computation steps will give us an algorithm to solve our problem.

ALGORITHM PRUNINGS.
empty_initialization:
$F\_DP(\varnothing, \varnothing) = 0$
left_initialization:
$F\_DP(T_1[l(i)..i_1], \varnothing = F\_DP(T_1[l(i)..l(i_1)-1], \varnothing) + \gamma(T_1[i_1] \to \Lambda)$
right_initialization:
$F\_DP(\varnothing, T_2[l(j)..j_1]) = F\_DP(\varnothing, T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1])$
general_if_computation
/* applies if $l(i_1) = l(i)$ and $l(j_1) = l(j)$*/
$F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$

$F\_DP(\varnothing, T_2[l(j)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1]),$
$F\_DP(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda),$
$F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1]),$
$F\_DP(T_1[l(i)..i_1-1], T_2[l(j)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])\}$

/* put the derived *treedist* in the permanent array, as specified by template */

general_else_computation

$F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1]) = \min\{$
$F\_DP(T_1[l(i)..l(i_1)-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda),$
$F\_DP(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda),$
$F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1]),$
$F\_DP(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1]) + T\_DP(i_1, j_1)\}$

LEMMA 9. *Algorithm Prunings is correct.*

Proof. First we show that the initialization is correct. The empty_initialization and the right_initialization are the same as in the tree distance algorithm. For left_initialization, the best we can do for tree $T_1[l(i_1)..i_1]$ is to prune at $T_1[i_1]$. Therefore $F\_DP(T_1[l(i)..i_1], \varnothing) = F\_DP(T_1[l(i)..l(i_1)-1], \varnothing) + \gamma(T_1[i_1] \to \Lambda)$. Hence the left_initialization is correct.

For the general term $F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1])$, we have the following similar three cases.

(1) $T_1[i_1]$ is not touched by a line of M.
  (1a) (without pruning) $F\_DP(T_1[l(i)..i_1-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda)$
  (1b) (with pruning) $F\_DP(T_1[l(i)..l(i_1)-1], T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda)$

(2) $T_2[j_1]$ is not touched by a line of M. Since we only prune from T_1, there is only one case here:

$$F\_DP(T_1[l(i)..i_1], T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_1[i_1])$$

(3) both $T_1[i_1]$ and $T_2[j_1]$ are touched by lines of M.
  (3a) (without pruning)

$$F\_DP(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1]) + F\_DP(T_1[l(i_1)..i_1-1], T_2[l(j_1)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])$$

  (3b) (with pruning)

$$F\_DP(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1]) + F\_DP(\varnothing, T_2[l(j_1)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])$$

If $l(i) = l(i_1)$ and $l(j) = l(j_1)$, consider cases (1b) and (3b.) Case (1b) becomes $F\_DP(\varnothing, T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda)$. Case (3b) becomes $F\_DP(\varnothing, T_2[l(j)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1])$. Now from the right_initialization we know that

$$\begin{aligned} &F\_DP(\varnothing, T_2[l(j)..j_1]) + \gamma(T_1[i_1] \to \Lambda) \\ &\quad \geqq F\_DP(\varnothing, T_2[l(j)..j_1-1]) + \gamma(\Lambda \to T_2[j_1]) + \gamma(T_1[i_1] \to \Lambda) \\ &\quad \geqq F\_DP(\varnothing, T_2[l(j)..j_1-1]) + \gamma(T_1[i_1] \to T_2[j_1]). \end{aligned}$$

So the distance given by case (1b) $\geqq$ the distance from (3b). The proposed general_if_computation is therefore correct where the first term handles two cases.

If $l(i) \neq l(i_1)$ or $l(j) \neq l(j_1)$, consider case (3). As in Lemma 6, cases (3a) and (3b) can be replaced by $F\_DP(T_1[l(i)..l(i_1)-1], T_2[l(j)..l(j_1)-1]) + T\_DP(i_1, j_1)$. The proposed general_else_computation is therefore correct. Hence algorithm pruning is correct. □

6. Conclusion. We present a simple dynamic programming algorithm for finding the editing distance between ordered labelled trees.[3] Our algorithm

(1) Has better time and space complexity than any in the literature;

(2) Is efficiently parallelizable; and

(3) Is generalizable with the same time complexity to approximate tree matching problems.

We have implemented these algorithms as a toolkit that has already been used at the National Cancer Institute.

Acknowledgments. We thank Bob Hummel for helpful discussions and the referees for valuable comments.

REFERENCES

[ALKBO] S. ALLUVIA, H. LOCKER-GILADI, S. KOBY, O. BEN-NUN, AND A.B. OPPENHEIM, RNase III *stimulates the translations of the* cIII *gene of bateriophage lambda*, Proc. Nat. Acad. Sci. U.S.A., 85 (1987), pp. 1-5.

[BSSBWD] B. BERKOUT, B.F. SCHMIDT, A. STRIEN, J. BOOM, J. WESTRENEN, AND J. DUIN, "*Lysis gene of bateriophage* MS2 *is activated by translation termination at the overlapping coat gene*," Proc. Nat. Acad. Sci. U.S.A., 195 (1987), pp. 517-524.

[DA] N. DELIHAS AND J. ANDERSON, *Generalized structures of* 5s *ribosomal* RNA's, Nucleic Acid Res. 10 (1982) p. 7323.

[DD] I. C. DECKMAN AND D. E. DRAPER, S4-*alpha* mRNA *translation regulation complex*, Molecular Biol 196 (1987), pp. 323-332.

[HO] C. M. HOFFMANN AND M. J. O'DONNELL, *Pattern matching in trees*, J. Assoc. Comput. Mach., 29 (1982), pp. 68-95.

[L] S. Y. LU, *A tree-to-tree distance and its application to cluster analysis*, IEEE Trans. Pattern Anal. Mach. Intelligence, 1 (1979), pp. 219-224.

[LV] G. M. LANDAU AND U. VISHKIN, *Introducing efficient parallelism into approximate string matching and a new serial algorithm*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 220-230.

[S80] P. H. SELLERS, *The theory and computation of evolutionary distances*, J. Algorithms, 1 (1980), pp. 359-373.

[S88] B. A. SHAPIRO, *An algorithm for comparing multiple* RNA *secondary structures*, Comput. Appl. Biosci. (1988), pp. 387-393.

[SK] J. L. SUSSMAN AND S. H. KIM, *Three dimensional structure of a transfer* RNA *in two crystal forms*, Science, 192 (1976), p. 853.

[T] KUO-CHUNG TAI, *The tree-to-tree correction problem*, J. Assoc. Comput. Mach., 26 (1979), pp. 422-433.

[U83] E. UKKONEN, *On approximate string matching*, in Proc. Internat. Conference on the Foundations of Computing Theory, Lecture Notes in Computer Science 158, Springer-Verlag, Berlin, New York, 1983, pp. 487-495.

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

[WF] R. WAGNER AND M. FISHER. *The string-to-string correction problem*, J. Assoc. Comput. Mach., 21 (1974), pp. 168-178.

[Z83] KAIZHONG ZHANG, *An algorithm for computing similarity of trees*, Tech. Report, Mathematics Department, Peking University, Peking, China, 1983.

[Z89] ———, *The editing distance between trees: algorithms and applications*, Ph.D. thesis, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, 1989.

[3] In a separate result obtained with the help of Rick Statman, we find that the editing distance between unordered labelled trees (i.e., where the sibling order is insignificant) is NP-complete. The reduction is from exact cover by 3-sets [Z89].

SIAM J. COMPUT.
Vol. 18, No. 6, pp. 1263–1282, December 1989

012

RELATING THE TYPE OF AMBIGUITY OF FINITE AUTOMATA TO THE SUCCINCTNESS OF THEIR REPRESENTATION*

BALA RAVIKUMAR† AND OSCAR H. IBARRA‡

Abstract. This paper considers the problem of how the size of a nondeterministic finite automaton (nfa) representing a regular language depends on the type of ambiguity of the nfa. Primarily, the relationship between the ambiguity and the size in five types of nfa's with increasing degrees of nondeterminism is studied: DFA (deterministic), UNA (unambiguous), FNA (finitely ambiguous), PNA (polynomially ambiguous), and ENA (exponentially ambiguous) nfa's. The goal is to show "separation" among these classes, where a class A is said to be "separated" from B (written (A, B)) if for infinitely many n, there are machines of type B with n states whose minimal equivalent type A machine has more than $p(n)$ states for any polynomial p. Two classes are "polynomially equivalent" (written $A = B$) if machines of type A can be converted to machines of type B with only a polynomial increase in the number of states, and vice versa. For a class X, let $X(b)$ denote the restricted class of machines of type X with the restriction that the language accepted is bounded. The first main result compares the bounded restrictions of the five classes mentioned above. Specifically, the following is shown: $(\mathrm{DFA}(b), \mathrm{UNA}(b))$, $(\mathrm{UNA}(b), \mathrm{FNA}(b))$, $\mathrm{FNA}(b) = \mathrm{PNA}(b)$ and $\mathrm{PNA}(b) = \mathrm{ENA}(b)$, providing a complete picture of how the type of ambiguity affects the size complexity for unary and bounded languages. For unbounded languages it is conjectured that each of the five types of nondeterminism is separate from its higher types. But a proof does not exist at this time for two of the separations, the other two carrying over directly from the unary case. Candidates are offered that may be useful in proving the (other two) conjectured separations, and also a weaker form of separation in one case is shown. The notion of "concurrent conciseness" introduced by Kintala and Wotschke is studied. A class C is said to be concurrently concise over two classes A and B if (A, B) and (B, C) can be proved using the same collection of witness languages. One of the main results of this paper shows that, for unrestricted inputs, PNA is concurrently concise over DFA and UNA. This answers an open problem of Stearns and Hunt. The succinctness problem is also studied through (regularity preserving) closure properties, an approach initiated by Sakoda and Sipser, and some interesting contrasts between various classes of nfa's are shown.

Key words. nondeterministic finite automaton, ambiguity, succinctness of representation

AMS(MOS) subject classifications. 68O, 68Q

1. Introduction.

Ever since Rabin and Scott [RABI59] introduced the concept of a nondeterministic finite automaton (nfa), problems concerning the complexity of representation (size complexity) of regular languages have been studied extensively. Since regular languages have a plethora of different characterizations, the study of concise representations of regular languages seems to offer an endless stream of problems. In light of recent developments relating the randomness of strings to the size complexity of machines generating them (this is the idea of Kolmogorov complexity, see e.g., [PAUL79]), there is a renewed interest in the study of succinctness of representations. We begin with a brief summary of some earlier results in this area.

Meyer and Fischer [MEYE71] have considered the blow-up in the number of states when converting an nfa to a deterministic finite automaton (dfa) and have shown that, in the worst case, the number of states in a minimum dfa equivalent to an nfa of n states is 2^n. The same problem has also been considered when the underlying alphabet is unary. In this case the blow-up, although smaller than 2^n, is still exponential. An exact expression for the blow-up has hitherto remained elusive. Moore [MOOR71],

* Received by the editors July 23, 1987; accepted for publication (in revised form) January 3, 1989. This research was supported in part by National Science Foundation grant DCR-8604603.

† Department of Computer Science and Statistics, University of Rhode Island, Kingston, Rhode Island 02881.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

Mandl [MAND73], and recently Denes, Kim, and Rouch [DENE85] have obtained a near-optimal bound of $\Theta(e^{(n \log n)^{1/2}})$ for this problem. (In the above expression, e stands for the base of the natural logarithm, and $\log n$ denotes $\log_2 n$, as do all the logarithms in the rest of the paper.)

Schmidt [SCHM78] has compared the relative succinctness offered by several pairs of devices that include, apart from finite-state devices, pushdown machines. Kintala and Wotschke [KINT80] have defined the "amount of nondeterminism" as the minimum number of nondeterministic moves on any accepted input and established a hierarchy of succinctness based on the amount of nondeterminism. They [KINT86] also have introduced a concept called concurrent conciseness and compared the concurrent conciseness of degree automata [WOTC77], probabilistic automata [RABI63], and nfa's. Ehrenfeucht and Zeiger [EHRE76] have obtained several results comparing the relative succinctness of finite automata and regular expressions. Additional results comparing the expressive power of various control features (such as "goto") in regular expressions have been presented by Abrahamson [ABRA87].

An important classification of nfa's is based on the notion of ambiguity, i.e., the number of accepting derivations of an input string x. (An accepting derivation of a string is defined as a sequence of states visited on input x leading to acceptance.) This notion was first introduced and studied by Mandel and Simon [MAND77], Jacob [JACO77], and Reutenauer [REUT77]. It has been further investigated by Chan and Ibarra [CHAN83], Stearns and Hunt [STEA85], Ibarra and Ravikumar [IBAR86], and Weber and Seidl [WEBE86]. An nfa is said to be k-ambiguous if, for each string accepted, there are at most k accepting computations. An nfa is said to be finitely ambiguous if it is k-ambiguous for some k. An nfa is said to be polynomially ambiguous if there is a polynomial $p(.)$ such that the number of accepting computations for any string of length n is at most $p(n)$. An nfa that is polynomially ambiguous, but not finitely ambiguous is called strictly polynomially ambiguous. The same way, an nfa is said to be strictly exponentially ambiguous if it is not polynomially ambiguous. (Note that the number of derivations of a string in *any* nfa is at most a single exponential in the length of the string, hence the term exponentially ambiguous would hold for any nfa.) It has been shown by Mandel and Simon [MAND77] and Reutenauer [REUT77] (and independently by others) that there is an algorithm to decide if a given nfa is finitely, strictly polynomially or strictly exponentially ambiguous. Weber and Seidl [WEBE86] have shown that there is a polynomial time algorithm for the above classification. The present study is motivated by the following question regarding the above classification. How is the succinctness of representation related to the degree of ambiguity?

In this study, we compare the following families of devices: DFA (the collection of deterministic finite automata), UNA (the collection of unambiguous nfa, i.e., one-ambiguous nfa's), FNA (the collection of finitely ambiguous nfa's), PNA (the collection of polynomially ambiguous nfa's), and ENA (the collection of exponentially ambiguous nfa's). For a collection X of devices (such as $X = \text{UNA}$, etc.), let $X(b)$ denote the subset of X with the property that each device in this subset accepts a bounded language (i.e., a language whose strings are of the form $a_1^{i_1} a_2^{i_2} \cdots a_k^{i_k}$, where a_i's are symbols of the alphabet). Also, for two classes of devices A and B, let (A, B) denote the fact that B *can be* exponentially more succinct than A (i.e., there exists a collection of languages $\{L_n\}$ such that L_n can be accepted by an n state device of type B, but any device of type A accepting L_n must have more than $p(n)$ states for any polynomial p) and let $A = B$ denote the fact that each device in A can be converted to an *equivalent* device in B with a polynomial blow-up and vice versa. We present

results comparing DFA(b), UNA(b), FNA(b), PNA(b), and ENA(b). Specifically, we show that (DFA(b), UNA(b)), (UNA(b), FNA(b)), FNA(b) = PNA(b), and PNA(b) = ENA(b) hold, i.e., for bounded inputs, fna's can be exponentially more succinct than una's and una's can be exponentially more succinct than dfa's; however, ena's and pna's are convertible to fna's at only a polynomial increase in the number of states. Note that the above claims immediately imply, for unrestricted inputs, the following relations: (DFA, UNA) and (UNA, FNA). The other relationships seem hard to prove. We conjecture that (FNA, PNA) and (PNA, ENA), and offer candidates for proving the claim. We also provide an evidence for the later conjecture by proving a special case of it.

Concurrent conciseness of a family A of devices over two families B and C has been defined in [KINT86] as a simultaneous succinctness of A over B and C. By "simultaneous" we mean that the same collection of languages bears witness to the succinctness of A over B, as also of B over C. The question of whether ENA is concurrently concise over UNA and DFA has been raised by Stearns and Hunt [STEA85]. We prove the stronger claim that PNA is concurrently concise over UNA and DFA, resolving the question of Stearns and Hunt in the affirmative.

We also study the succinctness problem through "closure properties," an approach initiated by Sakoda and Sipser [SAKO78]. We define the closure (with respect to an operation $\bullet$) for a class A of finite-state devices as follows. Let $\bullet$ be a regularity preserving operation, and let $M_1, M_2 \in A$. Then $L(M_1) \bullet L(M_2)$ is regular, and assuming that A is universal in the sense that it can accept the class of all regular languages, there is an $M \in A$ such that $L(M) = L(M_1) \bullet L(M_2)$. The question is: "What is the size of the smallest such M, as a function f of the numbers of states of M_1 and M_2?" If f is bounded by a polynomial, then we say that A is polynomially closed under $\bullet$. Closure properties show an interesting contrast between different subclasses of nfa's. For example, the class DFA is not polynomially closed under concatenation or Kleene star but ENA is, under these operations. Apart from the fact that many results regarding succinctness can be sharpened or presented in a different perspective through the closure properties, we feel that this study is of interest in its own right. We prove some closure and nonclosure properties for the various classes of devices mentioned above.

The main results are presented in §§ 3–5 following some definitions and background material in § 2.

2. Preliminaries. We begin with some basic definitions and notation. Define a dfa M to be a 5-tuple $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of accepting states, and δ is defined as $\delta : Q \times \Sigma \to Q$. An nfa is defined as a five-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$, where all the components are exactly as above except δ that is defined as a map $\delta : Q \times \Sigma \to 2^Q$. We do not allow ε-moves in an nfa. We further assume that the fa's are reduced, i.e., they do not have useless states (thus, all the states are reachable from the start state on some input, and from each state an accepting state can be reached on some input). Note that none of the results presented in this paper require these assumptions; rather these are made for stylistic simplicity and avoidance of inessential details. $|M|$ denotes the number of states in M. This will be used as the *size complexity* of a machine. We use $a_M(n)$ to denote the "ambiguity function" of an nfa, defined as the maximum number of *accepting* computations (i.e., sequences of states ending in an accepting state) for any input string w such that $|w| = n$ and $w \in L(M)$, the language accepted by M. An nfa is said to be k-ambiguous (for a fixed k) if $a_M(n) \leqq k$, for all n. One-ambiguous nfa's are called unambiguous. An nfa is polynomially ambiguous if

there exists a polynomial $p(.)$ such that $a_M(n) \leqq p(n)$ for all n. The unrestricted nfa is called exponentially ambiguous since the number of derivations for a string of length n is bounded by c^n (where c is the number of states of the machine).

Let Γ be an arbitrary collection of symbols. We use DFA to denote the collection of dfa's whose input alphabet is a finite subset Σ of Γ. In the same way, the collections of unambiguous, finitely, polynomially, and exponentially ambiguous nfa's are denoted by UNA, FNA, PNA, and ENA, respectively. Clearly DFA $\subseteq$ UNA $\subseteq$ FNA $\subseteq$ PNA $\subseteq$ ENA. To make a more definite reference to the type of ambiguity of an nfa, we use the following definition. An nfa of a given type X (in the above sequence of inclusions) is said to be *strictly* of type X if it is of type X, but not of any preceding type. For example, a polynomially ambiguous nfa is strictly polynomially ambiguous if it is polynomially ambiguous but not finitely ambiguous. We will also be interested in the following subsets of these classes of collections. When the input of any fixed device in any of these collections is restricted to a unary alphabet, we obtain the families DFA(u), UNA(u), etc. When the underlying languages accepted by these collections are bounded (i.e., subsets of $a_1^* a_2^* \cdots a_r^*$, where $a_i \in \Sigma$), we obtain the collections DFA(b), UNA(b), etc. We denote the collections of two-way dfa's (whose head can move on either directions on the input tape) by 2-DFA and those of sweeping dfa's (that are 2-dfa's whose head can reverse only on the endmarkers) by SDFA. Their restrictions to unary and bounded languages will be denoted by 2-DFA(u), 2-DFA(b), etc. To denote a typical or generic device of a specific type, we use the name of the collection in small letters, e.g., "Let M be an fna," etc.

The following definition of f-conciseness was adopted from [KINT86] with a minor modification:

DEFINITION. A class of automata B is f-concise over another class A if and only if there is an infinite sequence of languages $\{L_n\}$ such that

(i) For all n, there exists an n-state automaton M_n in B accepting L_n.

(ii) For infinitely many n, every M in A accepting L_n must have at least $f(n)$ states.

We write $B - f(n) \to A$ to denote the fact that B is f-concise over A. Since we are mainly interested in contrasting polynomial growths and superpolynomial growths, we also need the following coarser definition. We denote by (A, B) the fact that B is f-concise over A for some f that cannot be upper-bounded by a polynomial. If (A, B) is not true, then we write $B \leqq A$. Thus, if $B \leqq A$, then any $M \in B$ can be converted to an equivalent M' in A with at most a polynomial increase in the number of states. We write $A = B$ to denote the fact that $A \leqq B$ and $B \leqq A$. If $B \leqq A$ and if it is not true that $A \leqq B$, then we write $B < A$. Observe that it is possible that (A, B) and (B, A) are both true. In this case, we write $A <\ > B$. (A and B are *incomparable.*)

3. Succinctness problem for unary and bounded languages. In this section, we study the succinctness versus size relation when the input is restricted to bounded languages. First we consider unary languages and prove that DFA(u) $<$ UNA(u), UNA(u) $<$ FNA(u), and FNA(u) = PNA(u) = ENA(u). We then extend these results to bounded languages. We also present results comparing 2-DFA(u) with some of the classes stated above. Obviously the inequalities stated above among DFA, UNA, and FNA carry over to the unbounded languages. At present we can claim much less about the relative succinctness of FNA, PNA, and ENA. We conjecture that PNA $<$ FNA and ENA $<$ PNA, i.e., all the five types of nfa's form a hierarchy. We offer candidates that might enable us to prove the conjectured separations. In the latter case, we also provide an evidence in defense of the conjecture by proving a weaker form of it.

We need the following preliminary results. Our first lemma is from [STEA85].

LEMMA 1. *Let* $M=\langle Q, \Sigma, \delta, q_0, F\rangle$ *be an unambiguous* nfa *with* m *states. The shortest word not accepted by* M *is no longer than* m.

Our next result is from [IBAR86] and [WEBE86]. This result provides a characterization of polynomial and exponential ambiguities of nfa's. Before we state the result, we need the following notation. For an nfa $M=\langle Q, \Sigma, \delta, q_0, F\rangle$, where $p, q \in Q$, and $w \in \Sigma^*$, define $N(p, w, q)$ to be the number of ways in which w can be derived starting from p, ending in q.

LEMMA 2. *Let* $M=\langle Q, \Sigma, \delta, q_0, F\rangle$ *be an* nfa.

(i) [IBAR86] *A necessary and sufficient condition for* M *to be strictly exponentially ambiguous is that there exists a* $q \in Q$ *and a string* $w \neq \varepsilon$ *such that* $N(q, w, q) \geqq 2$.

(ii) [WEBE86] *A necessary and sufficient condition for* M *to be strictly polynomially ambiguous is that* M *is not exponentially ambiguous and there exist different states* p, q *and a word* $w \neq \varepsilon$ *such that* $p \in \delta(p, w)$, $q \in \delta(p, w)$, *and* $q \in \delta(q, w)$.

Figures 1 and 2 show examples of strictly polynomially and exponentially ambiguous nfa's.

The following definition and lemma are from [CHRO86].

DEFINITION. An nfa $M=\langle Q, \Sigma, \delta, q_0, F\rangle$ where $\Sigma=\{a\}$ is in *normal form* if it has the following properties:

(a) $Q=\{q_0, \cdots, q_m\} \cup C_1 \cup \cdots C_k$, where $C_i=\{p_{i,0}, \cdots, p_{i,y_i-1}\}$, $i=1, 2, \cdots, k$,

(b) δ can be described as follows: $q_{i+1} \in \delta(q_i, a)$, $i=1, \cdots, m-1$, $p_{i,j+1} \in \delta(p_{i,j}, a)$, $i=1, \cdots, k$, and $j=0, \cdots, y_i-1$, $p_{i,0} \in \delta(q_m, a)$, $i=1, \cdots, k$.

(The addition $j+1$ above is mod y_i.)

We now state the next lemma.

LEMMA 3 [CHRO86]. *For each* nfa A *(over a unary alphabet) with* n *states there is an equivalent* nfa A' *in normal form with at most a total of* $O(n^2)$ *states, and at most* n *states that are not in a loop.*

Proof. For the proof see Lemma 4.3 of [CHRO86]. □

In fact, [CHRO86] presents a simple algorithm to construct A' from A.

We now prove the result comparing UNA(u), FNA(u), PNA(u), and ENA(u).

THEOREM 1. DFA(u) < UNA(u), UNA(u) < FNA(u), and FNA(u) = PNA(u) = ENA(u).

Proof of DFA(u) < UNA(u). It is sufficient to show (DFA(u), UNA(u)). We construct a collection $\{L_n\}$, $n=1, 2, \cdots$ of regular languages such that for all sufficiently

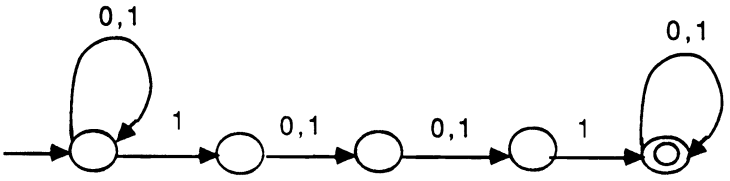


FIG. 1. *Example of a strictly polynomially ambiguous* nfa.

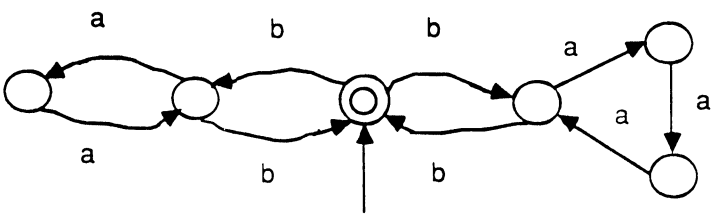


FIG. 2. *Example of a strictly exponentially ambiguous* nfa.

large n, the following holds: (i) L_n can be accepted by an unambiguous nfa with m_n+1 states, and (ii) for any dfa N accepting L_n, the number of states M_n of N must be at least $m^{(\log m_n)^{1/2}/8}$. We first show how to construct $\{L_n\}$, $n=1,2,\cdots$, and prove claims (i) and (ii) about them.

Let p_i denote the ith prime number. In order to define $\{L_n\}$, it is convenient to construct a square matrix A_n of order $n\times n$: the diagonal entries of A_n are * (space-filler), A_n is symmetric so we only need to describe the upper triangular portion of A_n. The upper triangular portion of A_n (consisting of $n(n-1)/2$ entries) is simply the sequence $p_n, p_{n+1}, \cdots, p_{n-1+n(n-1)/2}$ placed in the row-major order. Figure 3 shows A_n. A specific example is shown in Fig. 4.

Now define the sets $\{S_i^n\}$, $i=1,\cdots,n$ as follows: S_i^n is the collection of elements of the ith column in A_n (excluding *). Also let $T_n=\cup_{i=1}^{n} S_i^n$. We can now define the languages $\{L_j^n\}$, $j=1,2,\cdots,n$ as follows: $L_j^n=\{a^i \mid i\equiv j \pmod m\}$ for all $m\in S_j^n\}$. Finally, let $L_n=\cup_{j=1}^{n} L_j^n$. Also let $m_n=\prod_{p\in S_1^n} p+\prod_{p\in S_2^n} p+\cdots+\prod_{p\in S_n^n} p$, and $M_n=\prod_{p\in T_n} p$.

To simplify the notation we use m and M instead of m_n, M_n, respectively. The same way, we will drop n from all the symbols keeping in mind the fact that we are

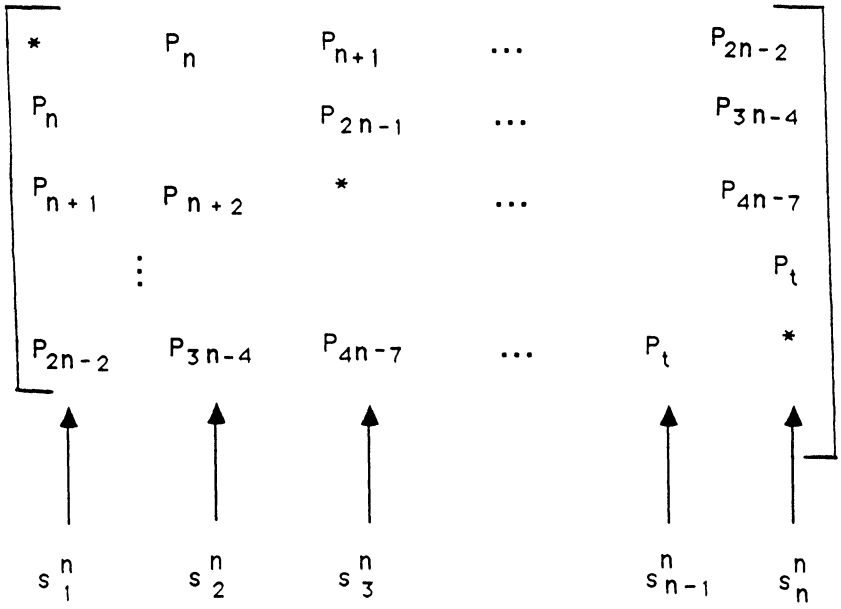


FIG. 3. *The matrix* A_n $(t=n-1+n(n-1)/2)$.

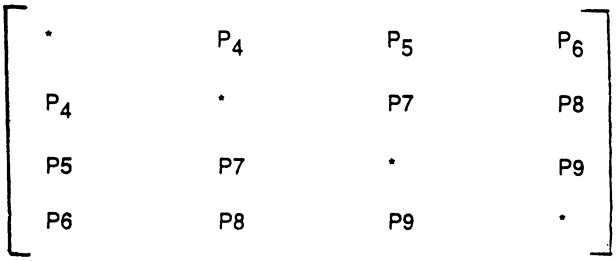


FIG. 4. *Matrix* A_4.

proving the result for a generic n. Thus, for example, we will use S_i, T, L_j, and L in the place of S_i^n, T_n, L_j^n, and L_n, respectively.

CLAIM 1. For all sufficiently large n, $M \geqq m^{(\log m)^{1/2}/8}$.

Proof. Recall from the prime number formula [APOS76] and [ROSS62] that there exist integers c_1, c_2 such that for all r, $c_1 r \log r \leqq p_r \leqq c_2 r \log r$. Now for any $p \in T$, $p \leqq p_{n^2}$. Thus $\prod_{p \in S_i} p \leqq (p_{n^2})^{n-1}$. Therefore

$$\begin{aligned} m &= \prod_{p \in S_1} p + \prod_{p \in S_2} p + \cdots + \prod_{p \in S_n} p \\ &\leqq n \cdot (p_{n^2})^{n-1} \\ &\leqq n \cdot (2c_2 n^2 \log n)^{n-1} \\ &\leqq (c_2 n \log n)^{2n} \end{aligned} \tag{1}$$

for all sufficiently large n. Thus, for all large n,

$$m \leqq (c_2 n \log n)^{2n}.$$

Next,

$$\begin{aligned} M &= \prod_{p \in T} p \\ &\geqq p_n^{|T|} \text{ (since } p_n \text{ is the smallest element in } T) \\ &= p_n^{n(n-1)/2} \\ &\geqq (c_1 n \log n)^{n(n-1)/2} \\ &\geqq (c_2 n \log n)^{n^2/4} \text{ for all sufficiently large } n, \\ &\geqq m^{n/8} \text{ from inequality (1).} \end{aligned}$$

Thus,

$$M \geqq m^{n/8}. \tag{2}$$

Also from the inequality (1), $m \leqq (c_2 n \log n)^{2n}$ so $\log m \leqq 2n \log (c_2 n \log n) < n^2$ for large n. Thus, $n \geqq (\log m)^{1/2}$. Combining this with (2) we have the desired inequality. □

CLAIM 2. There exists an unambiguous nfa with $m+1$ states that accepts L.

Proof. It is easy to see that there exists a dfa M_i with $\prod_{p \in S_i} p$ states accepting L_i. Now construct an nfa M with $1 + \sum_{i=1}^{n} |M_i| = 1 + m$ states as shown in Fig. 5. The machine M has ε-moves, but they can be removed without increasing the number of states.

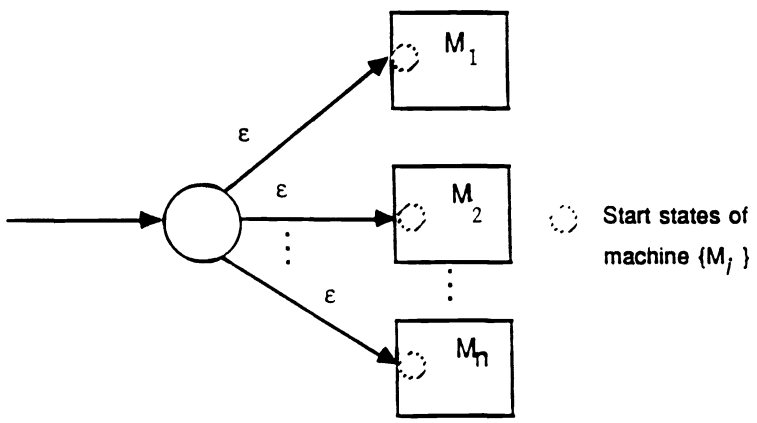


FIG. 5. nfa M.

Clearly M accepts L and has $m+1$ states as claimed. It only remains to show that M is unambiguous. We show this by proving that $L_i \cap L_j = \emptyset$ for $i \neq j$.

SUBCLAIM. For $i \neq j$, $L_i \cap L_j = \emptyset$.

Proof. Assume the contrary. (Without loss of generality we assume that $i > j$.) Then there exists k such that $a^k \in L_i \cap L_j$. We then have $k \equiv i \pmod{p}$ for all $p \in S_i$, and $k \equiv j \pmod{p}$ for all $p \in S_j$.

This means that $k = \alpha \cdot (\prod_{p \in S_i} p) + i$ for some integer α, and $k = \beta \cdot (\prod_{p \in S_j} p) + j$ for some integer β. Thus

$$(**) \qquad -\alpha \cdot \left(\prod_{p \in S_i} p \right) + \beta \cdot \left(\prod_{p \in S_j} p \right) = i - j.$$

Note that there is (exactly) one $t \in S_i \cap S_j$. Clearly, t divides $-\alpha \cdot (\prod_{p \in S_i} p) + \beta \cdot (\prod_{p \in S_j} p)$. Thus, from (**) it follows that t divides $i-j$. Note that $1 \leqq j < i \leqq n$, so $i - j < n$. However $t \geqq p_n \geqq c_1 n \log n > n$. Therefore t cannot divide $i-j$ since t is larger than $i-j$. This is a contradiction. Hence M is unambiguous. □

We now prove our final claim.

CLAIM 3. For any dfa N accepting L, the number of states in N must be at least M.

Proof. Assume the contrary, i.e., there is a dfa $N = \langle Q, \Sigma, \delta, q_0, F \rangle$ with fewer than M states accepting L. Then there exist integers i, j $(0 \leqq j < i < M)$ such that $\delta(q_0, a^i) = \delta(q_0, a_j)$. Let the modular representation of i and j be $\bar{i} = (i_1, i_2, \cdots, i_t)$ and $\bar{j} = (j_1, j_2, \cdots, j_t)$, respectively, where $t = |T|$. (This means that i_m is the remainder when i is divided by the mth element s_m in some fixed ordering $s_1, s_2, \cdots, s_t$ of T.) Clearly, $\bar{i} \neq \bar{j}$, so for some r, $1 \leqq r \leqq t$, $i_r \neq j_r$. We will now describe a procedure to find an integer k such that $a^{i+k} \in L$, but $a^{j+k} \notin L$. The procedure is given below.

First note that there is a unique pair of sets (S_p, S_q) where $p < q$ such that $S_p \cap S_q = \{s_r\}$. The basic idea is to choose k such that

(i) $a^{j+k} \notin L_i$ for any $i \neq q$, and

(ii) $a^{i+k} \in L_q$.

We would then argue that a^{j+k} is in L_q. We carry out the construction using the matrix A_n. First delete the qth row and the qth column of A_n to obtain A'_n. In A'_n choose the elements in the diagonal immediately above the main diagonal. Let these elements be $r_2, r_3, \cdots, r_{q-1}, r_{q+1}, \cdots, r_{n-1}$. (Note that the sets to which these elements belong are $S_2, S_3, \cdots, S_{q-1}, S_{q+1}, \cdots, S_{n-1}$, respectively.) Finally pick r_1 from the set $S_1 - \{r_2, r_3, \cdots, r_{n-1}\}$.

Let the remainders when j is divided by $r_1, r_2, \cdots, r_{n-1}$ be $e_1, \cdots, e_{n-1}$, respectively. (Note that $e_1, e_2, \cdots$, etc. are components of the vector $\bar{j}$.) Also let the elements of S_q be $\{u_1, \cdots, u_{n-1}\}$ and the remainders when i is divided by $u_1, \cdots, u_{n-1}$ be $f_1, f_2, \cdots, f_{n-1}$, respectively. (As above, note that $f_1, f_2, \cdots$, etc. are components of vector $\bar{i}$.) Now k (a positive integer) can be chosen to satisfy the following set of congruences:

$$\left\{ \begin{array}{l} k \equiv -e_1 + 1 + 1 \pmod{r_1} \\ k \equiv -e_2 + 2 + 1 \pmod{r_2} \\ \qquad \cdots \\ k \equiv -e_{n-1} + n - 1 + 1 \pmod{r_{n-1}} \end{array} \right\} \quad \text{(set 1)}$$

$$\left\{ \begin{array}{l} k \equiv -f_1 + q \pmod{u_1} \\ k \equiv -f_2 + q \pmod{u_2} \\ \qquad \cdots \\ k \equiv -f_{n-1} + q \pmod{u_{n-1}} \end{array} \right\} \quad \text{(set 2)}$$

Note that, by the Chinese Remainder Theorem [NIVE60], such a positive integer k exists. It is easy to see that $a^{j+k} \notin L_i$ for any $i \neq q$, and $a^{i+k} \in L_q$. Let c be such that $u_c = s_r$. Clearly $i+k \equiv q \pmod{u_c}$, but $i \not\equiv j \pmod{u_c}$. (Recall that $i \equiv i_r \pmod{u_c}$, $j \equiv j_r \pmod{u_c}$ and $i_r \neq j_r$.) Thus $j+k \not\equiv q \pmod{u_c}$ so $a^{j+k} \notin L_q$. Thus $a^{j+k} \notin L$. This is a contradiction since $\delta(q_0, a^i) = \delta(q_0, a^j)$. This completes the proof of Claim 3. Thus $\mathrm{DFA}(u) < \mathrm{UNA}(u)$.

Proof of $\mathrm{UNA}(u) < \mathrm{FNA}(u)$. Since $\mathrm{UNA}(u) \leqq \mathrm{FNA}(u)$, it is enough to show that $(\mathrm{UNA}(u), \mathrm{FNA}(u))$. Define a collection of languages as follows. Let p_i denote the ith prime number and L_k be defined as

$$L_k = \{a^n \mid n \not\equiv 0 \pmod{p_j} \text{ for some } j \leqq k\}.$$

It is easy to see that L_k can be accepted by an fna with $\sum_{i=1}^{k} p_i$ states. Such an fna for $k=3$ is shown in Fig. 6. By the prime number theorem [APOS76], $p_i = O(i \log i)$, so $\sum_{i=1}^{k} p_i = O(\sum_{i=1}^{k} i \log i) = O(k^2 \log k)$.

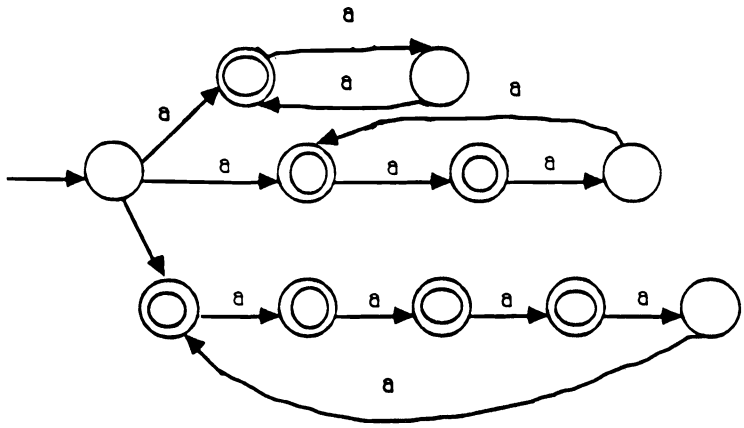


FIG. 6. *An* fna *accepting* L_3.

We next show that any una accepting L_k requires at least $\prod_{i=1}^{k} p_i - 1$ states. To show this, let M be any una accepting L_k. We see that the shortest string not accepted by L_k is $a^{p_1 p_2 \cdots p_k}$. Thus, by Lemma 1, the size of M must be at least $\prod_{i=1}^{k} p_i - 1 \geqq 2^k - 1$. This proves that $(\mathrm{UNA}(u), \mathrm{FNA}(u))$. Using the f-conciseness notation of [KINT86] (see § 2), the above result can be stated in a sharper form as $\mathrm{FNA}(u) - (2^{(2n/\log n)^{1/2}} - 1) \to \mathrm{UNA}(u)$.

Proof of $\mathrm{FNA}(u) = \mathrm{PNA}(u) = \mathrm{ENA}(u)$. This immediately follows from Lemma 3 via the following observations: (i) A' in Lemma 3 is finitely ambiguous and (ii) $|A'| \leqq O(|A|^2)$. □

We next show how to extend Theorem 1 to bounded languages.

THEOREM 2. $\mathrm{DFA}(b) < \mathrm{UNA}(b)$, $\mathrm{UNA}(b) < \mathrm{FNA}(b)$, *and* $\mathrm{FNA}(b) = \mathrm{PNA}(b) = \mathrm{ENA}(b)$.

Proof. The facts that $\mathrm{DFA}(b) < \mathrm{UNA}(b)$ and $\mathrm{UNA}(b) < \mathrm{FNA}(b)$ immediately follow from Theorem 1. In the rest of the proof we show that $\mathrm{FNA}(b) = \mathrm{ENA}(b)$. For simplicitly we consider bounded languages L such that $L \subseteq a^*b^*$. It is easy to generalize the result. Let $L \subseteq a^*b^*$ be accepted by an ena and let $|M| = n$. We further simplify the discussion by assuming that $L \cap a^* = \varnothing$. (Otherwise, we can let $L = L_1 \cup L_2$ where $L_1 \cap a^* = \varnothing$ and $L_2 \subseteq a^*$ and apply the following procedure for the language L_1. It is

straightforward to construct a machine for L_2 from M. Finally one can construct a machine for $L_1 \cup L_2$.) We show that L can be accepted by a fna with at most $p(n)$ states for some polynomial $p(.)$. The intuitive idea behind the proof is simple, and we illustrate it with an example. Intuitively the states of an nfa accepting $L \subseteq a^*b^*$ can be partitioned into two sets A and B as follows: for some state q, if all the arcs incident into q are labeled a, then q is in A else it is in B. For the nfa in Fig. 7, the partition is as shown in Fig. 8.

Clearly if p, q are in A (in B) and if $q \in \delta(p, t)$, then $t = a$ ($t = b$). As a next step, we introduce additional states in A and B so that in the resulting machine there are only ε-arcs from A to B. This requires the introduction of the following states: for each q in B such that for some $p \in A$, $q \in \delta(p, a)$, introduce a state q' in A and replace the arc $\langle p, q\rangle$ labeled a by two arcs: $\langle p, q'\rangle$ labeled a and $\langle q', q\rangle$ labeled ε. For each q in B such that for some $p \in A$, $q \in \delta(p, b)$, introduce a state q'' in B and replace the arc $\langle p, q\rangle$ labeled a by two arcs: $\langle p, q''\rangle$ labeled ε and $\langle q', q\rangle$ labeled b. This construction applied on Fig. 8 results in Fig. 9.

Let M_1 and M_2 be the machines in Fig. 9, whose transition graphs are induced by the set of states A and B, respectively. For any pair (p, q), where p and q are states in M_i ($i = 1$ or 2), let $M_i(p, q)$ denote the machine obtained by keeping the transitions of M_i, but renaming the start state as p, and the accepting state as q. Convert each $M_i(p, q)$ to a finitely ambiguous machine using Lemma 3. (Call the resulting machine again as $M_i(p, q)$.) Also assume that the states of $M_i(p, q)$ have been distinctly named. Finally, construct a new machine M' as follows. M' uses the following machines: (1) For each q with an ε-arc leaving q, the machine $M_1(p_0, q)$ where p_0 is the start state of M, (2) for each p with an ε-arc entering into p, and for each accepting state q of

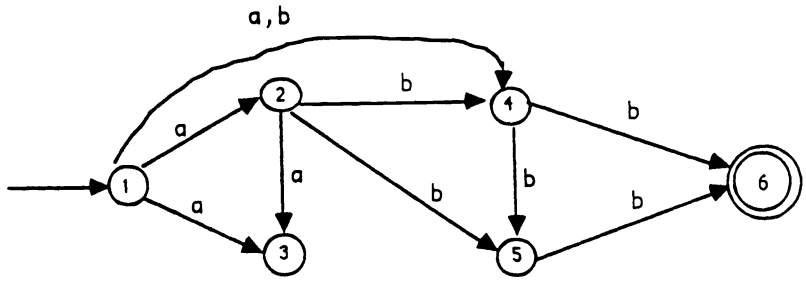


FIG. 7

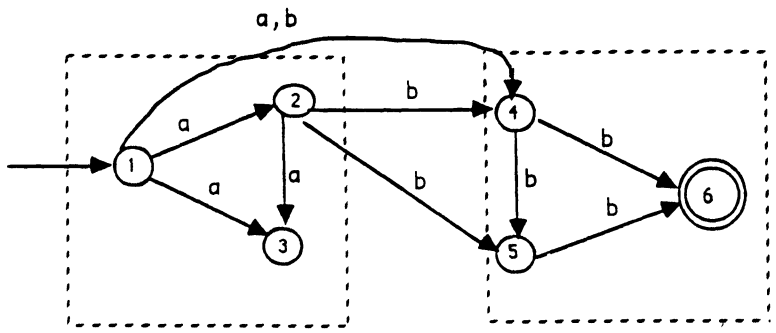


FIG. 8

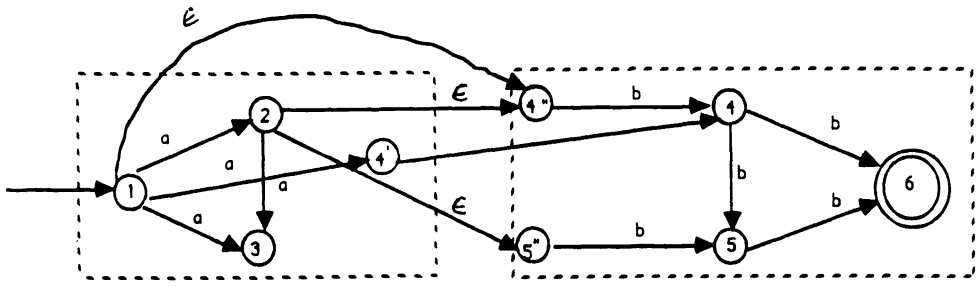


FIG. 9

M, the machine $M_2(p, q)$. Also introduce a new start state p_0' and introduce an ε-arc from p_0' to the start state p_0 in each machine $M_1(p_0, r)$ for each r. Also for each ε-arc $\langle p, q\rangle$ in Fig. 9, connect an ε-arc from the state p of the machine $M_1(p_0, p)$ to the state q of the machine $M_2(q, t)$ for each t. Finally, the set of accepting states of M' is $\{t | M_2(q, t)$ is used in $M'\}$. The machine M' derived from Fig. 9 is shown in Fig. 10.

It is easy to prove the correctness of the above construction. □

We conclude this section with a brief discussion of the relative conciseness of the collections 2-DFA(u) and NFA(u). The problem of 2-DFA versus NFA was considered in [SAKO78] and [SIPS79]. In [SAKO78] it has been shown that (ENA, 2-DFA) and conjectured that ENA $<>$ 2-DFA, but this claim is (to the best of our knowledge) still open. In [SIPS79] it has been proved that (SDFA, ENA), a weaker claim. Berman [BERM79] and Sipser [SIPS79] have shown that (SDFA, 2-DFA). In contrast to the above claims, we prove the following result.

THEOREM 3. (i) ENA(u) $<$ 2-DFA(u), (ii) 2-DFA(u) $=$ SDFA(u).

Proof of (i). To prove the desired claim, we must show two results: ENA(u) $\leqq$ 2-DFA(u) and (ENA(u), 2-DFA(u)). We first outline a proof of the former result. This result was also obtained independently by Chrobak [CHRO86]. We must show that any given nfa M over a unary alphabet can be converted to a 2-dfa with a polynomial blowup in the number of states. Using Lemma 3, convert M to a fna M' in the normal form. M' can be viewed as a finite union of dfa's. Now construct a 2-dfa

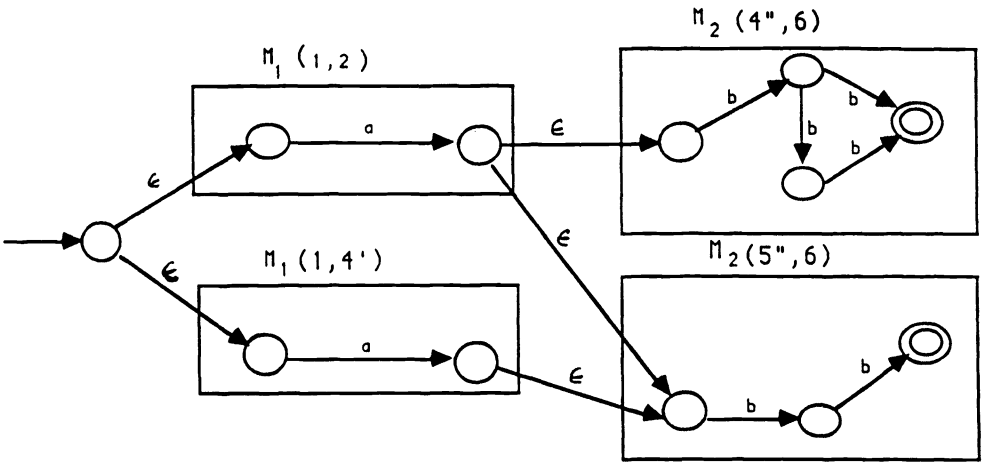


FIG. 10

that simulates this fna by simulating a dfa during each sweep. This result was independently obtained by Chrobak [CHRO86]. We next prove that (ENA(u), 2-DFA(u)). Consider the collection F_k of languages defined as follows: $F_k = \{a^n | n$ is divisible by p_j for *all* $j \leqq k\}$. It is easy to show that F_k can be accepted by a $\sum_{j=1}^{k} p_j$ state 2-dfa, and any nfa accepting F_k requires at least $\prod_{j=1}^{k} p_j$ states. This completes the proof of (i).

Proof of (ii). The proof is analogous to Theorem 2 presented in [IBAR88]. We omit the details. □

We conclude this section with some remarks regarding the succinctness problem for unbounded languages. The result (DFA, UNA) has been proved by Meyer and Fischer [MEYE71] using the following candidate: $L_m = L(r_m)$ where $r_m = (0+1+\varepsilon)^m 1(0+1)^{m-1}$. L_m can be accepted by a ufa with $2m+1$ states and any dfa that accepts L_m has at least 2^m states. Note that Theorem 1 shows that the claim holds even over a one-letter alphabet. The same way, as shown in Theorem 1, (UNA, FNA) also holds even for unary alphabet languages. We have not been able to prove either of the following claims: (1) PNA $<$ ENA (2) FNA $<$ PNA. We conjecture that both these results are true and offer the following candidate languages.

For (FNA, PNA): Let $L_k = L(r_k)$ where $r_k = (0+1)^*1(0+1)^k1(0+1)^*$. Figure 11 shows a pna with $k+3$ states accepting L_k. We conjecture that there is no polynomial $p(.)$ such that for a collection $\{M_k\}$ of fna's accepting L_k, $|M_k| \leqq p(k)$ for all k.

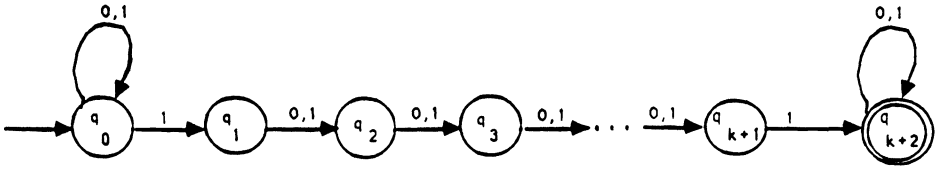


FIG. 11. *A* pna *for* L_k.

For (PNA, ENA): Let $L_k = \{a^n | n \not\equiv 0 \pmod{p_j}$ for some $j \leqq k$ and $n \geqq 1\}$, where p_j is the jth prime number. (Recall that this collection of languages was used in Theorem 1.) Now let $A_k = (\# L_k \#)^*$. (A_k is the "marked Kleene closure" of L_k in the automata theoretic parlance.) It is easy to see that there exists an ena of size $O(\sum_{i=1}^{k} p_i) = O(k^2 \log k)$ states accepting A_k.

We conjecture that the number of states in any pna accepting A_k is not bounded by a polynomial in k. Although we do not have a proof of this result, we provide an evidence by showing that among a restricted class of pna's none with fewer than $2^{k/2}$ states can accept A_k. This restricted class of machines is defined as follows. For any machine M in this class, there is only one state q in Q, the set of states of M, such that for any q', q'' in Q, if $q'' \in \delta(q', \#)$, then $q'' = q$. Informally it means that there is only one state in M with arcs labeled "#" entering into it. We now prove the following claim regarding restricted pna's.

CLAIM. Let M be a restricted pna accepting A_k. Then $|M| \geqq 2^{k/2}$.

Proof. Let $M = \langle Q, \Sigma, \delta, q_0, \{f\}\rangle$. Clearly, f is the only state such that for some $q \in Q$, $f \in \delta(q, \#)$. Define $Q_1 = \{q | f \in \delta(q, \#)\}$ and $Q_2 = \{q | q \in \delta(f, \#)\} = \{f\}$. For any $q \in Q_1$, define M'_q as $M'_q = \langle Q, \Sigma, \delta, f, \{q\}\rangle$. Next define M_q from M'_q as follows: (i) Remove all the edges labeled # from M'_q; and (ii) reduce the resulting nfa, i.e., remove the useless states.

First observe that $Q_1 \cap Q_2 = \varnothing$, for if $q \in Q_1 \cap Q_2$, then for some string $w \in A_k$, $f \in \delta(q_0, w)$, and hence $f \in \delta(q_0, w\#\#)$ implying that $w\#\# \in A_k$, a contradiction, since

ε is not in A_k. We claim that M_q is a ufa for all $q \in Q_2$, as seen from the following argument. Suppose M_q is ambiguous. Then there is a word $w \neq \varepsilon$ such that $N(p, w, q) \geqq 2$. In this case, for the original machine M, $N(f, \# w \#, f) \geqq 2$. Thus by Lemma 2, M is exponentially ambiguous, a contradiction. Furthermore, using the same argument, we observe that $L(M_r) \cap L(M_s) \neq \varnothing$ for $r \neq s$. Now it is easy to construct a ufa M' with the following properties: (i) $|M'| \leqq \sum_q |M_q| + 1$; and (ii) $L(M') = L_k$. The construction of M' is by simply introducing a new start state and having ε-transitions to the start states of all M_q's and then removing the ε-transitions. Hence, M' is unambiguous. We now show that $L(M') = L_k$. Since it is obvious that $L(M') \subseteq L_k$, we need only to show that $L_k \subseteq L(M')$. Let $\# w \# \in L_k$. Then, $\# w \# \# w \#$ is in A_k. Since M is a restricted machine, any computation (i.e., any sequence of states) on the input $\# w \#$ will terminate in f, and hence $f \in \delta(f, \# w \#)$. Thus $w \in L(M_q)$ for some q, so $w \in L(M')$. This completes the proof that $L_k = L(M')$. The bound on the number of states in M' is obvious. Thus, if $|M| = t$, then $|M'| \leqq t^2 + 1$. From Lemma 1, we see that $t^2 + 1 \geqq |M'| \geqq \prod_i p_i > 2^k$, and thus $t \geqq 2^{k/2}$ (for $k \geqq 2$). This proves the desired claim. □

Table 1 summarizes the results on succinctness for unary and bounded languages.

4. Concurrent conciseness of representations. Concurrent conciseness was introduced in [KINT86] as an important generalization of "usual" conciseness for the following reasons:

> Assume that we have three classes of automata C_1, C_2, and C_3. One is often able to prove the conciseness of C_2 over C_1 using a sequence of languages $\{A_n\}$, and the conciseness of C_3 over C_2 using another sequence of languages $\{B_n\}$. The problem is that $\{A_n\}$ and $\{B_n\}$ have often very little in common since $\{A_n\}$ exploits the advantages of C_2 over C_1 and $\{B_n\}$ exploits the advantages of C_3 over C_2. Concurrent conciseness addresses the question whether there is *one* sequence of languages which concurrently establishes the conciseness of C_2 over C_1 and that of C_3 over C_2.

Following [KINT86], we quantify the concurrent conciseness as follows.

DEFINITION. Say that C_1 is *f,g-concurrently concise* over C_2 and C_3 denoted by $C_1 - f(n) \to C_2 - g(n) \to C_3$ for some $f(n)$ and $g(n)$ if C_1 is f-concise over C_2 and C_2 is g-concise over C_3 for the same sequence of languages.

TABLE 1
Succinctness results for unary and bounded inputs.

	UNA	FNA	PNA	ENA	2-DFA
DFA	<	<	<	<	<
UNA		<	<	<	<
FNA			=	=	<
PNA				=	<
ENA					<

Since our primary concern in this paper has been to contrast the polynomial growth and the superpolynomial growth, we need the following coarser definition. We write (A, B, C) to denote the fact that $C - f(n) \to B - g(n) \to A$, where f and g are both superpolynomials. Thus (A, B, C) means that, in a strong sense, A, B, and C form a hierarchy with respect to concise representations. Our next result is to show that (DFA, UNA, PNA). This result settles an open problem mentioned by Stearns and Hunt [STEA85].

THEOREM 4. PNA-$2^{\sqrt{n}/(4\sqrt{2})} \to$ UNA-$(n/16)^{(\log n - 4)} \to$ DFA.

Proof. Consider the collection $\{J_k\}$ of languages defined as follows:

$J_k = \{x1y1 | x, y \in \{0, 1\}^*, |y| = k^2$ and x has a substring of the form $1z1$ where $|z| = k\}$.

We show the following.

CLAIM 1. J_k can be accepted by a k^2+k+4 state pna.

Proof. We describe informally a pna accepting J_k. M_k uses $k+3$ states to verify the existence of the substring $1z1$, and counts k^2 characters following a 1 to accept strings in J_k. It is easy to see that M_k satisfies our requirements. □

CLAIM 2. Any dfa accepting J_k requires at least 2^{k^2+1} states.

Proof. Let M be a dfa accepting J_k. For any strings $x, y \in 1^{k+3}\Sigma^{k^2+1}$, $x \neq y$ implies $\delta(q_0, x) \neq \delta(q_0, y)$. Thus M must have 2^{k^2+1} states. □

CLAIM 3. There exists a $(2^{k+3}+k^2+1)$-state una accepting J_k.

Proof. We first construct a dfa M'_k with 2^{k+3} states to accept the language $\{x1y1 | x, y \in \{0, 1\}^*, |y| = k\}$. M'_k has as its states q_σ where σ in $\{0, 1\}^*$ with $|\sigma| \leqq k+2$. q_λ is the start state. The δ-function of M'_k is constructed in such a way that $\delta(q_\lambda, w) = q_w$ if $|w| \leqq k+2$, else $\delta(q_\lambda, w) = q_{w'}$ where w' is a suffix of w of length $k+2$. The accepting states of M'_k are $F = \{q_w | w = 1w'1$ for some w' of length $k\}$. We modify M'_k to M_k as follows:

(A) Introduce new states $p_0, p_1, \cdots, p_{k^2}$, and introduce the following transitions:

$$\delta(p_i, 0) = \delta(p_i, 1) = \{p_{i+1}\} \text{ if } i < k^2,$$

(B) Remove the transitions from all the states of F in M'_k and add the following transitions:

$$\delta(p, 0) = \{p\}, \delta(p, 1) = \{p, p_0\} \text{ for all } p \text{ in } F.$$

(C) The only accepting state of M_k is the state p_{k^2}.

Figure 12 shows the construction of M_k from M'_k:

Clearly, M_k is an una such that $L(M_k) = J_k$. □

The final step in the proof is the following claim:

CLAIM 4. Any una accepting the language J_k has at least $2^{k/4}$ states.

Proof. The basic machinery used in this proof is an argument based on linear independence that was originally proposed by Schmidt [SCHM78] (Theorem 3.9). This technique was also used in [SIPS79].

Let M be a ufa accepting J_k. We want to show that $|M| \geqq 2^{k/4}$. In the rest of the proof, we follow the notation used in Theorem 3.9 of [SCHM78].

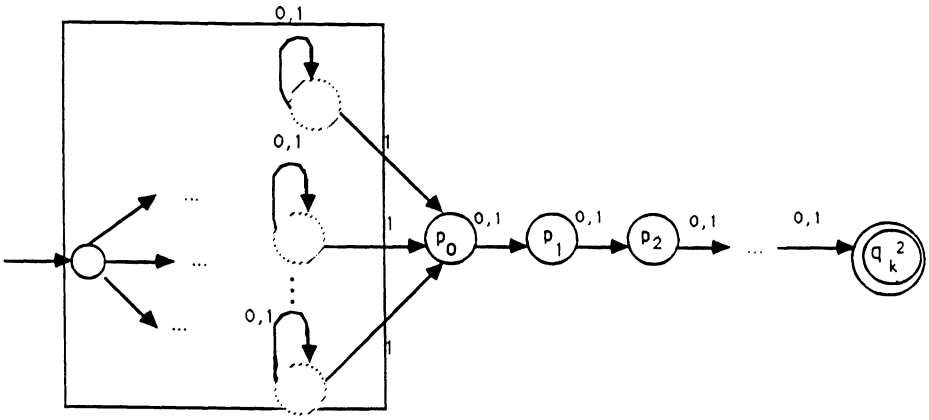


FIG. 12. M_k *constructed from* M'_k.

Let $x \in \{0,1\}^{k+1}$, and assume that $K_x = \{q_1, q_2, \cdots, q_{f(x)}\}$ is the set of states reachable from q_0 via x. Define for each i, $(1 \leqq i \leqq f(x))$, the set $A_i^x = \{y \text{ in } \{0,1\}^{k+1} |$ there exists a $q \in F \cap \delta(q_i, y10^{k^2})\}$. Thus A_i^x is the set of words of length $k+1$ that lead from the state q_i (in K_x) to acceptance. Define $A = \{\{A_i^x\} | i = 1, 2, \cdots, f(x), x \in \{0,1\}^{k+1}\}$. Let $B_1, B_2, \cdots, B_m$ be a listing of the members of A without repetitions. Let $K = \{q | q \in \delta(q_0, x)$ for some x in $\{0,1\}^{k+1}\}$. For any $q \in K$, associate the set $B_j = \{y$ in $\{0,1\}^{k+1} |$ there exists a q in $F \cap \delta(q, y10^{k^2})\}$. Since this mapping is surjective (onto), $m \leqq |K|$. Also since $|K| \leqq |M|$, we have $m \leqq |M|$. We complete the proof of Claim 4 by showing that $m \geqq 2^{k/4}$.

To do this, we interpret the subsets of $\{0,1\}^{k+1}$ as elements of the 2^n-dimensional vector-space over the field Z_2. Let x_i, $0 \leqq i \leqq 2^{k+1} - 1$, be the binary representation of i with a padding of zeros (at the left) to make $|x_i| = k+1$. With each $C \subseteq \{0,1\}^{k+1}$, we associate the vector $\bar{C} = (c_0, c_1, \cdots, c_{2^{k+1}-2}, c_{2^{k+1}-1})$, where $c_j = 1$ if $x_j \in C$ and $C_j = 0$ otherwise. For simplicity, assume that k is odd. The proof can be easily modified to handle the case when k is even.

Now let $T \subseteq \{0,1\}^{k+1}$ be defined as: $T = \{w |$ number of 0's in w is $(k+1)/2\}$. For any $x \in T$, consider the set $A_x = \bigcup_{i=1}^{f(x)} A_x^i$ defined as above. A_x thus consists of y's such that $xy10^{k^2} \in J_k$. Since M is unambiguous, the sets $A_x^1, \cdots, A_x^{f(x)}$ are mutually disjoint and furthermore, all the A_x^i's occur among the B_i's, the vector $\overline{A_x}$ can be written as a linear combination of vectors $\{\overline{B_i}\}$, $i = 1, 2, \cdots, m$, i.e., there exist $t_1, t_2, \cdots, t_m \in \{0,1\}$ such that

$$\overline{A_x} = \sum_{j=1}^{m} t_j \overline{B_j}. \tag{3}$$

Consider the set of vectors $V = \{\overline{A_y} | y \in T\}$. We shall show that V is linearly independent, as follows. Let $y_1, y_2, \cdots, y_r$ (r, the number of elements in $T = \binom{k+1}{(k+1)/2}$) be an arbitrary but fixed ordering of T. We define a Boolean matrix U of order $r \times 2^k$ as follows:

$$U = \begin{bmatrix} \overline{A_{y_1}} \\ \overline{A_{y_2}} \\ \overline{A_{y_3}} \\ \cdots \\ \overline{A_{y_r}} \end{bmatrix}.$$

Thus the rows of U are the vectors $\overline{A_{y_1}}, \overline{A_{y_2}}, \cdots, \overline{A_{y_r}}$. Proving that V is linearly independent is equivalent to proving that the rank of U is r. We prove the latter by showing that there exists a submatrix $\bar{U}$ (of U) of order $r \times r$ that can be derived by permuting the columns of the complement of the permutation matrix $\overline{I_r}$ given by

$$\overline{I_r} = \begin{bmatrix} 011 & \cdots & 1 \\ 101 & \cdots & 1 \\ \cdots & \cdots & \cdots \\ 111 & \cdots & 0 \end{bmatrix}.$$

For any i, $1 \leqq i \leqq r$, define $g(i)$ as the integer obtained by complementing the bits in the string $y_i = x_{g(i)}$. Now the submatrix $\bar{U}$ is defined by the set of columns $\{g(i)\}$, $i = 1, 2, \cdots, r$ of U. We claim that, in the column $g(i)$ of U, the only 0 entry is $U_{i,g(i)}$. For a string x, let $\bar{x}$ denote the string obtained by interchanging the 0's and 1's in x. Clearly, the string $y_i \overline{y_i} 10^{k^2} \notin J_k$ since for any pair of positions $(t, \bar{t})$ separated by k in $y_i \overline{y_i}$, the tth and $\bar{t}$th letters are complementary. Thus, $\overline{y_i} \notin A_{y_1}$ implying $U_{i,g(i)} = 0$. Next

observe that $y_j\overline{y_1}10^{k^2}$ is in J_k as seen from the following argument. Since $y_j \neq y_i$, there exists a t, $1 \leqq t \leqq k$, such that the tth letter of y_j is the same as the tth letter of $\overline{y_i}$. Thus, $y_j\overline{y_i}10k^2 \in J_k$. This implies $\overline{y_i} \in A_{y_j}$ implying $U_{j,g(i)} = 1$ for any $j \neq i$. Thus the rank of $\bar{U}$ and U is r and hence V is linearly independent. From (1) it follows that V is a subset of the vector space generated by $\overline{B_1}, \overline{B_2}, \cdots, \overline{B_m}$. Therefore $m \geqq r = \binom{k+1}{(k+1)/2} \geqq 2^{k/4}$. □

The claim made in Theorem 4 follows from the proof of Claims 1-4. □

Since both $2^{\sqrt{n}/(4\sqrt{2})}$ and $(n/16)^{\log n - 4}$ are superpolynomials, we obtain the following corollary.

COROLLARY 1. (DFA, UNA, PNA).

This corollary settles the question raised in [STEA85] of whether it is possible to have simultaneous nonpolynomial blowups among the three collections, DFA, UNA, and ENA. In fact, our result proves a stronger claim.

It appears that the same candidate languages $\{J_k\}$ can be used to prove (DFA, FNA, PNA). This can be done by showing a claim stronger than Claim 4 of Theorem 4, namely, any fna accepting J_k requires $f(k)$ states for some superpolynomial $f(.)$.

5. Closure properties of classes of finite-state devices. In this section, we consider the closure properties of various classes of finite-state devices. Let A be a class of finite state devices that is universal in the sense that every regular language can be accepted by some machine in A. (Note that all the classes of finite state devices considered in this paper are universal.) We ask the following question. "If L_1 and L_2 are accepted by machines M_1 and M_2 of type A, is it true that $L_1 \bullet L_2$ can also be accepted by a machine of type A, whose size is bounded by a polynomial in the sizes of M_1 and M_2, where $\bullet$ is a regularity preserving operation?" If the answer is yes, then we say that A is *polynomially closed* under $\bullet$. Many of the succinctness claims made earlier can be sharpened by studying the polynomial closure of various classes of devices under some fundamental regularity preserving operations such as concatenation and complementation.

The following definition is essentially due to Sakoda and Sipser [SAKO78].

DEFINITION. Let A be a collection of devices, and let $\bullet$ be a regularity preserving operation. A is said to be closed under $\bullet$ if there exists a polynomial $p(.)$ such that the following holds: For any $M_1, M_2 \in A$, there is an $M \in A$ such that $L(M) = L(M_1) \bullet L(M_2)$ and $|M| \leqq p(\max\{|M_1|, |M_2|\})$.

We study the closure of the collections DFA, UNA, FNA, PNA, ENA, and 2-DFA under the following fundamental regularity preserving operations: union, intersection, complementation, concatenation, Kleene star, and reversal. Table 2 contains a summary of results. A Y denotes that the closure property holds and an N denotes that it does not. A question mark indicates that the problem is open. A number next to a Y or N (such as $Y(3)$, etc.) denotes its location in the proof of Theorem 5. (Trivial proofs of closure are omitted.)

THEOREM 5. *The closure properties as indicated in Table 2 hold.*

Proof. (1) Let L_1 and $L_2 \subseteq \{0, 1\}^*$ be defined by the following regular expressions $r_1 = (0+1)^*1$ and $r_2 = (0+1)^{n-1}$. There exist dfa with 2 and n states, respectively, accepting L_1 and L_2. It is also easy to see that $L_1 . L_2$ cannot be accepted by a dfa with less than 2^n states.

(2) To prove this result, we first define a large alphabet, similar to the alphabet used in [SAKO78] to obtain a "hardest language" for the problem of conversion from 1-nfa to 2-dfa.

Let B_n be the set of bipartite graphs with $2n$ vertices, whose vertex partition has

TABLE 2
Closure properties.

	DFA	UNA	FNA	PNA	ENA	2-DFA
Union	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*
Intersection	*Y*	*Y*	*Y*	*Y*	*Y*	*Y*
Complement	*Y*	?	*N*(7)	*N*(7)	*N*(7)	*Y*(3)
Reversal	*N*(4)	*Y*(4)	*Y*(4)	*Y*(4)	*Y*(4)	*Y*
Concatenation	*N*(1)	*N*(5)	?	*Y*	*Y*	?
Kleene star	*N*(2)	*N*(6)	?	?	*Y*	?

n vertices each. Pictorially, the graphs are represented as in Fig. 13, with the vertices occurring in two columns (left and right). The directed edges are from left to right and each vertex has outdegree and indegree at most 1. Figure 13 shows two members of the alphabet B_4.

For two graphs G_1 and G_2, the product $G_1 \otimes G_2$ is defined as a graph obtained by superposing the right boundary of G_1 with the left boundary of G_2. Figure 14 shows the graph $G_1 \otimes G_2$ for G_1 and G_2 of Fig. 13.

The notion of product can be extended in an obvious way to several graphs. In a product graph, the start vertex is the vertex in the first row of the leftmost boundary, and the terminal vertex is the last row of the rightmost boundary (u and v, respectively, in Fig. 14).

Now we define a collection of languages L_k as follows: $L_k = \{G_1 \cdots G_r | G_i \in B_k$ for all i, and $G_1 \otimes G_2 \otimes \cdots \otimes G_r$ has a directed path from start to terminal vertex$\}$. It is easy to see that L_k can be recognized by a dfa with k states. We show that any dfa M recognizing L_k^* has a least 2^{k-1} states. For any string $G_1 \cdots G_r$, define $\sigma(G_1 \cdots G_r) \subseteq \{2, 3, \cdots, k\}$ as follows. In the product graph $G_1 \otimes \cdots \otimes G_r$, add new edges from the last row to the first row on all columns except the first and last columns.

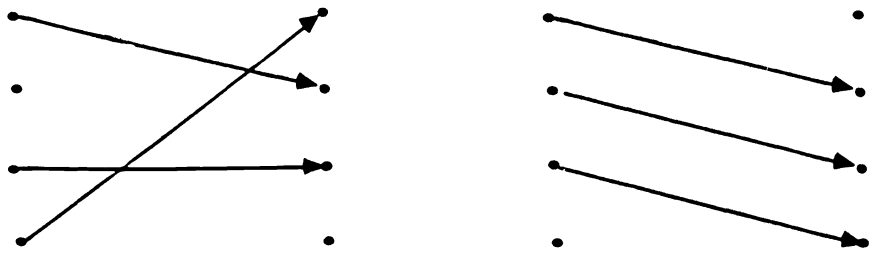


FIG. 13. *Two members of* B_4.

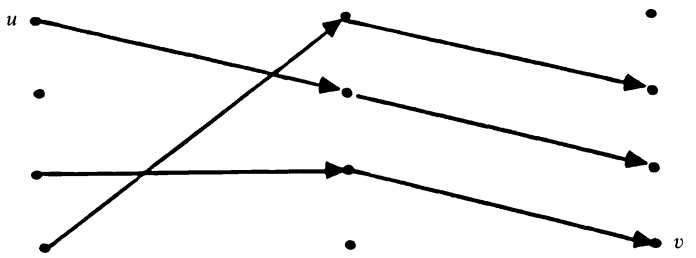


FIG. 14. *The graph of* $G_1 \otimes G_2$.

An integer i, $2 \leqq i \leqq k$, is in $\sigma(G_1 \cdots G_r)$ if in the resulting graph, there is a path from the start vertex to the vertex on the ith row of the last column. Let M be a dfa that accepts L_k^*, and let δ be its transition function. It is easy to see that if $\delta(q_0, G_1 \cdots G_m) = q_i$, and $\delta(q_0, G_1' \cdots G_n') = q_j$, and if $\sigma(G_1 \cdots G_m) \neq \sigma(G_1' \cdots G_n')$, then q_i and q_j are different. Thus, there are at least 2^{k-1} states in M. Finally note that B_k can be mapped to $\{0, 1\}$ and the lower bound claim still holds.

(3) This result has been proved by Sipser [SIPS80] as an application of a general result that any deterministic off-line TM using bounded space can be made halting.

(4) For DFA, the nonclosure follows from the collection of languages $L_n = (0+1)^{n-1}1(0+1)^*$. The closure for the other families follows from the fact that the standard procedure for constructing a machine to accept the reverse of a language (viz, reversing the direction of the arrows, etc.) preserves the type of ambiguity.

(5) To prove that ufa's are not closed under concatenation, we need a result of [SCHM78]. Let $C_k = \{x \# y \| x| = |y| = k, \text{ and } x \neq y\}$. It has been shown in [SCHM78] that any una accepting C_k requires at least 2^k states. We define the languages

$$L_1 = \{x \| x| \leqq k\},$$

$$L_2 = \{ay \# x_1 b x_2 | a, b \in \{0, 1\}, a \neq b, |y| \leqq k-1, |y| = |x_2|, |x_1| + |x_2| = k-1\},$$

$$L_3 = \{x \# y \| x| = |y| = k\}.$$

It is easy to see that L_1 and L_3 can be accepted by una's with $O(k)$ states, and L_2 can be accepted by a una with $O(k^2)$ states. Also note that $L_1 \,.\, L_2 \cap L_3 = C_k$. Since una's are closed under intersection, the desired claim follows from Schmidt's result.

(6) The nonclosure of ufa's under Kleene star follows from the fact that $C_k = (L_1 \cup L_2)^* \cap L_3$.

(7) We show the stronger claim that FNA(u) is not closed under complement. Let p_i be the ith prime number. Consider $L_k = \{a^n | n \not\equiv 0 \pmod{p_i} \text{ for some } 1 \leqq i \leqq k, n \geqq 1\}$. We have noted in Theorem 1 that there exists a finitely ambiguous nfa with $\sum_{i=1}^{k} p_i = O(k^2 \log k)$ states to accept L_k. We shall now show that any nfa M accepting $\overline{L_k}$ requires at least $\prod_{i=1}^{k} p_i > 2^k$ (for $k \geqq 2$) states. First observe that the two shortest strings in $\overline{L_k}$ are of length 0 and $t = \prod_{i=1}^{k} p_i$. Consider a sequence of states in M leading to acceptance on input a^t. (Such a sequence exists since a^t is in $\overline{L_k}$.) Let this sequence be $q_0, \cdots, q_t$. If M has fewer than t states, there exist integers i and j $(1 \leqq i < j \leqq t)$ such that $q_i = q_j$. This implies that a^{t-j+i} is in $\overline{L_k}$, a contradiction since $0 < t-j+i < t$. This completes the proof. □

6. Concluding remarks. In this paper, we have compared the relative succinctness of nondeterministic finite automata of various types of ambiguities and have established that the machines with "higher ambiguity" tend to be more succinct. These results are refinements and/or improvements over the earlier results of similar kind such as presented in [MEYE71] and [SCHM78]. We have presented a result on the concurrent conciseness of NFA over UFA and DFA, solving an open problem due to Stearns and Hunt. We also have studied the succinctness of various classes of finite-state devices through regularity preserving transformations.

Among the problems that remain open, we mention the following:

(1) Prove that (FNA, PNA).

(2) Prove that (PNA, ENA).

(3) Prove that FNA is not polynomially closed under concatenation.

It appears that the same candidate languages $D_k \subseteq \{0, 1\}^*$ are suitable for problems (1) and (3). The collection D_k is: $D_k = \{w | w = x1y1z, x, y, z \text{ in } \{0, 1\}^*, |y| = k\}$. It is easy to see that D_k can be accepted by a pna with $O(k)$ states. It remains to be shown that any fna accepting D_k requires at least $\Omega(\tau^k)$ for some $\tau > 0$. In addition to solving

open problems (1) and (3), this claim would also settle an unproven conjecture stated in § 4, namely, (DFA, FNA, PNA).

Acknowledgment. The authors are grateful to an anonymous referee for a remarkably thorough reading of an earlier version of this paper that led to many corrections and improvements.

REFERENCES

[ABRA87] K. ABRAHAMSON, *Succinct representation of regular sets using gotos and Boolean variables*, J. Comput. System Sci., 34 (1987), pp. 129–148.

[APOS76] T. APOSTOL, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, Berlin, 1976.

[BERM79] P. BERMAN, *A note on sweeping automata*, in Proceedings of the International Colloquium on Automata, Languages and Programming 1979, pp. 91–97.

[CHAN83] T. CHAN AND O. IBARRA, *On the finite-valuedness problem for sequential machines*, Theoret. Comput. Sci., 23 (1983), pp. 95–101.

[CHRO86] M. CHROBAK, *Finite automata and unary languages*, Theoret. Comput. Sci., 47 (1986), pp. 146–158.

[DENE85] J. DENES, K. H. KIM, AND F. W. ROUCH, *Automata on One Symbol*, Studies in Pure Mathematics 1985, pp. 127–134.

[EHRE76] A. EHRENFEUCHT AND P. ZEIGER, *Complexity measures for regular expressions*, J. Comput. System Sci., 12 (1976), pp. 134–146.

[IBAR86] O. IBARRA AND B. RAVIKUMAR, *On sparseness, ambiguity and other decision problems for acceptors and transducers*, in Proc. Third Annual Symposium on Theoretical Aspects of Computer Science, Orsay, France, 1986, pp. 171–179.

[IBAR88] ———, *Sub-logarithmic-space Turing machines, nonuniform space complexity and closure properties*, Math. Systems Theory, 21 (1988), pp. 1–17.

[JACO77] G. JACOB, *Un algorithme calculant le cardinal, fini on infini, des demi-groupes des matrices*, Theoret. Comput. Sci., 5 (1977), pp. 183–204.

[KINT80] C. KINTALA AND D. WOTSCHKE, *Amounts of nondeterminism in finite automata*, Acta Inform., (1980), pp. 199–204.

[KINT86] ———, *Concurrent conciseness of degree, probabilistic, nondeterministic and deterministic finite automata*, in Proc. Fourth Symposium on Theoretical Aspects of Computer Science, 1986, pp. 291–305.

[MAND73] R. MANDL, *Precise bounds associated with the subset construction on various classes of nondeterministic finite automata*, 7th Princeton Conference on Information and System Sciences, (1973), pp. 263–267.

[MAND77] A. MANDEL AND I. SIMON, *On finite semi-groups of matrices*, Theoret. Comput. Sci., 5 (1977), pp. 183–204.

[MEYE71] A. MEYER AND M. FISCHER, *Economy of description by automata, grammars, and formal systems*, in Proc. 12th IEEE Symposium on Switching and Automata Theory, IEEE. Computer Society, Washington, DC, 1971, pp. 188–191.

[MOOR71] F. MOORE, *On the bounds for state-set size in the proofs of equivalence between nondeterministic and two-way automata*, IEEE Trans. Comput., 20 (1971), pp. 1211–1214.

[NIVE60] I. NIEVEN AND H. ZUCKERMAN, *An Introduction to the Theory of Numbers*, John Wiley, New York, 1960.

[PAUL79] W. PAUL, *Kolmogorov complexity and lower bounds*, in Proc. 2nd International Conference on Fundamentals of Computation Theory, 1979.

[RABI59] M. RABIN AND D. SCOTT, *Finite automata and their decision problems*, IBM J. Res. Develop., 3 (1959), pp. 114–125.

[RABI63] M. RABIN, *Probabilistic automata*, Inform. Control, 6 (1963), pp. 230–245.

[REUT77] C. REUTENAUER, *Propietes arithmetiques et topologiques de series rationelles en variables non commutatives*, These troisieme cycle, Université de Paris VI, Paris, France, 1977.

[ROSS62] B. ROSSER AND L. SCHOENFELD, *Approximate formulas for some functions of prime numbers*, Illinois J. Math., 6 (1962), pp. 64–94.

[SAKO78] W. SAKODA AND M. SIPSER, *Nondeterminism and the size of two-way finite automata*, in Proc. 10th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 275–286.

[SCHM78] E. SCHMIDT, *Succinctness of descriptions of context-free, regular and finite languages*, Ph.D. thesis, Cornell University, Ithaca, NY, 1978.

[SIPS79] M. SIPSER, *Lower bounds on the size of sweeping automata*, in Proc. 11th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1979, pp. 360-364.

[SIPS80] ———, *Halting space-bounded computations*, Theoret. Comput. Sci., (1980), pp. 335-338.

[STEA85] R. STEARNS AND H. HUNT, *On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata*, SIAM J. Comput., 14 (1985), pp. 598-611.

[WEBE86] A. WEBER AND H. SEIDL, *On the degree of ambiguity of finite automata*, in Proc. of Math. Foundations of Computing Science, 1986, pp. 620-629.

[WOTC77] D. WOTCHKE, *Degree languages: A new concept of acceptance*, J. Comput. System Sci., 14 (1977), pp. 187-199.

SIAM J. COMPUT.
Vol. 18, No. 6, p. 1283, December 1989

013

ERRATUM: TWO APPLICATIONS OF INDUCTIVE COUNTING FOR COMPLEMENTATION PROBLEMS*

ALLAN BORODIN†, STEPHEN A. COOK†, PATRICK W. DYMOND‡, WALTER L. RUZZO§, AND MARTIN TOMPA¶

Jun Tarui has pointed out a technical error in the last row of Table 1: expected time is not the correct measure for defining the classes *PP* and *PLP*. Gill [2, Prop. 3.1], for instance, shows that any recursively enumerable set can be accepted in *constant* expected time by a probabilistic machine with unbounded two-sided error. Similarly, Jung's result [3] that $PL = PLP$ would have a straightforward proof if expected time were used in the definition of *PLP*. In both cases, the definition should be in terms of worst case time. This error has no effect on the results in the remainder of the paper.

REFERENCES

[1] A. BORODIN, S. A. COOK, P. W. DYMOND, W. L. RUZZO, AND M. TOMPA, *Two applications of inductive counting for complementation problems*, SIAM. J. Comput., 18 (1989), pp. 559–578.

[2] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.

[3] H. JUNG, *On probabilistic time and space*, Automata, Languages, and Programming, Lecture Notes in Comput. Sci., Springer-Verlag, Berlin, 1985, pp. 310–317.

* Received by the editors August 3, 1989; accepted for publication August 30, 1989.

† Department of Computer Science, University of Toronto, Toronto, Ontario, Canada M5S 1A4.

‡ Department of Computer Science and Engineering, C-014, University of California at San Diego, La Jolla, California 92093.

§ Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195.

¶ Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.