SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0001 $01.00/0

DETERMINING THE CHROMATIC NUMBER OF A GRAPH*

COLIN McDIARMID†

Abstract. Certain branch-and-bound algorithms for determining the chromatic number of a graph are proved usually to take a number of steps which grows faster than exponentially with the number of vertices in the graph. A similar result holds for the number of steps in certain proofs of lower bounds for chromatic numbers.

Key words. graph, chromatic number, algorithm, proof

1. Introduction. Graph coloring problems arise in many practical situations, for example in various timetabling and scheduling problems (see for example [13], [14]). It would be very useful to be able to determine quickly the chromatic number of a graph. However, it is well known that this problem is *NP*-hard, and thus we do not expect to find good algorithms for the problem ([1], [9]). A class of branch-and-bound coloring algorithms, which we call "Zykov" algorithms (see [5]) has been proposed. We branch on whether or not two nonadjacent vertices will have the same color and bound by using the fact that the chromatic number is at least the size of any complete subgraph. Zykov algorithms always explore at least a "pruned Zykov tree" for a graph. We shall prove that for almost all graphs G_n on n vertices every pruned Zykov tree has size (number of vertices) at least $c^{n \log^{1/2} n}$, where c is a constant >1. It follows that for any Zykov algorithm the number of steps usually required grows faster than exponentially with the size of the graph.

E. L. Lawler [10] has recently noted that a simple algorithm involving the maximal stable sets of a graph requires a number of steps which grows only (!) exponentially with the size of the graph. The Lawler algorithm is then asymptotically faster than any Zykov algorithm. This result contrasts with the conclusions of D. G. Corneil and B. Graham [5].

In the next section we give some preliminary definitions, including those of Zykov trees and Zykov algorithms. In § 3 we investigate the size of (unpruned) Zykov trees. (The standard algorithm for determining the chromatic polynomial of a graph involves the exploration of a Zykov tree—see for example [2, chap. 15].) Then in § 4 we investigate the size of pruned Zykov trees and deduce that Zykov algorithms are slow. Finally in § 5 we give an interpretation of our earlier results in terms of the lengths of certain proofs concerning chromatic numbers. The results in this section are similar in spirit to some recent results of V. Chvátal [4], and indeed the research reported here was initially inspired by discussions with Chvátal concerning his results. He was interested in certain "recursive" proofs for establishing upper bounds for stability numbers of graphs, and showed that for almost all graphs with a (sufficiently large) linear number of edges, the number of steps in any such proof grows at least exponentially with the size of the graph. This result implies that for a certain (wide) class of algorithms which determine the stability number of a graph each member algorithm is "slow".

Further related results are given in the forthcoming paper [12]. Both this paper and the paper [12] are based on the technical report [11].

* Received by the editors May 31, 1977, and in final revised form January 13, 1978.

† Corpus Christi College, Oxford, England. Presently at London School of Economics, London, England.

2. Preliminaries. We consider only graphs without loops or multiple edges. A (*proper*) *coloring* of a graph G is a coloring of the vertices of G so that no two adjacent vertices receive the same color; and the *chromatic number* $\chi(G)$ is the least number of colors in a proper coloring of G. A graph is *complete* if every two vertices are adjacent; and the *clique number* $\omega(G)$ is the greatest number of vertices in a complete subgraph of G. A set of vertices is *stable* if no two are adjacent; and the *stability number* $\alpha(G)$ is the greatest number of vertices in a stable set. A *proper partition* of G is a partition of the vertex set into stable sets. Thus proper partitions and proper colorings are closely related.

Let n be a positive integer. We denote by $\mathcal{G}_n$ the set of all graphs with vertex set $\{1, 2, \cdots, n\}$, and by $\mathcal{G}_n^*$ the set of all graphs with vertex set the sets of a partition of $\{1, 2, \cdots, n\}$. We may fail to distinguish between an integer k and the singleton set $\{k\}$ containing it, and for example consider that $\mathcal{G}_n \subseteq \mathcal{G}_n^*$. The use of sets to label vertices is simply a notational convenience.

We adopt in this paper a very simple probability model. (A more general model is considered in [12].) Throughout the paper p will be a real number with $0 < p < 1$ and q will be $1-p$: usually p and q will be constants. We induce a probability distribution on the set $\mathcal{G}_n$ of graphs by stipulating that each edge occurs independently with probability p. Thus for example the number of edges in a graph in $\mathcal{G}_n$ is a binomial random variable $B = B\left(\binom{n}{2}, p\right)$ with parameters $\binom{n}{2}$ and p.

We now move on towards the definitions of Zykov trees and Zykov algorithms. Suppose that x and y are nonadjacent vertices in a graph H in $\mathcal{G}_n^*$. Following [5] we define the reduced graphs H'_{xy} and H''_{xy} (or simply H' and H''). The former, H'_{xy}, is obtained from H by simply adding an edge joining x and y; and the latter, H''_{xy}, is obtained from H by replacing the vertices x and y by a single new vertex adjacent to each vertex to which x or y was adjacent. We say that H' and H'' are obtained from H by an "edge-addition" and a "vertex-contraction" respectively. In any proper coloring of H either x and y have different colors or they have same color. Thus we have the simple and well known result (see [15]) that

$$\chi(H) = \min\{\chi(H'), \chi(H'')\}. \tag{2.1}$$

Suppose that we have a graph H in $\mathcal{G}_n^*$ which is itself a leaf in a binary tree. Then *branching* at H involves choosing nonadjacent vertices x and y in H and giving H the leftson H'_{xy} and the rightson H''_{xy}. Of course we cannot branch at H if H is complete. Now let G be a graph in $\mathcal{G}_n$. If we start with the single node G, the root of our binary tree, and branch repeatedly we obtain a *partial Zykov tree* for G. By (2.1) we know that $\chi(G)$ is the minimum value of $\chi(L)$ over all leaves L of any partial Zykov tree for G. A *Zykov tree* for G is a partial Zykov tree in which each leaf is a complete graph, that is in which we have branched until we can branch no more. We give below an example of a Zykov tree for a graph in $\mathcal{G}_4$. (See also [2, chap. 15] and [5].)

Example. See Fig. 2.1.

We have now described the "branching" process to be used in our branch-and-bound algorithms. The "bounding" process depends on the obvious result that for any graph G

$$\chi(G) \geqq \omega(G). \tag{2.2}$$

A *Zykov algorithm* is a branch-and-bound algorithm for determining the chromatic number of a graph, using branch and bound processes as described above.

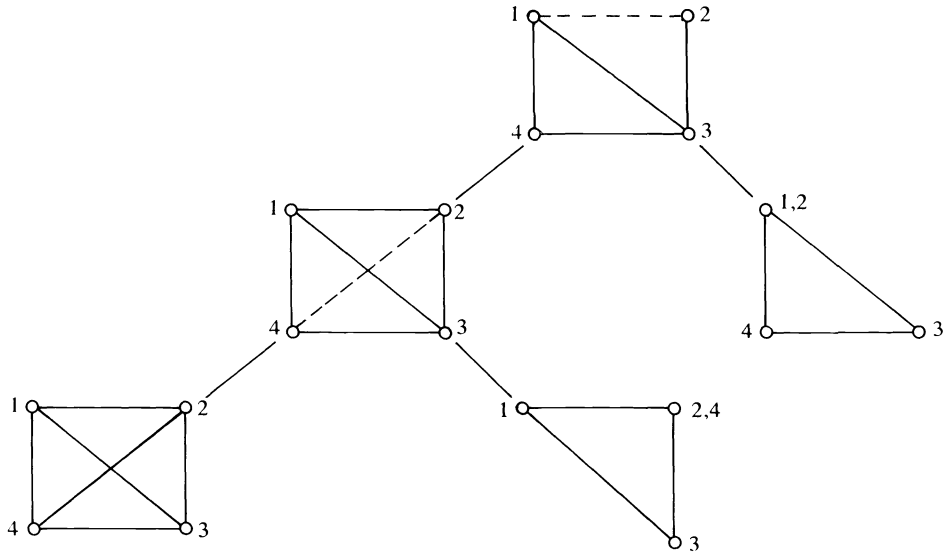


FIG. 2.1.

Such an algorithm has a subroutine for determining for each graph H a lower bound $\omega'(H)$ for the clique number $\omega(H)$ (for example by finding a complete subgraph of H). Also it maintains a current best upper bound for the chromatic number, which is always at most the number of vertices in any graph encountered. It operates as follows on a graph G. It begins to (construct and) explore a partial Zykov tree for G, starting with the root G. Suppose that at some stage we have explored a partial Zykov tree T for G and we have an upper bound b for $\chi(G)$. The algorithm chooses a leaf L of T with $\omega'(L) < b$ if there is such a leaf, then it branches at L and updates the upper bound: if there is no such leaf L the algorithm returns $\chi(G) = b$ and stops. Examples of Zykov algorithms are investigated in [5] and [12].

It is easy to see that after a finite number of steps a Zykov algorithm returns the correct value for the chromatic number and stops. Further if say it conducts a depth-first search of the partial Zykov tree the storage requirement need only be say $O(n^3)$. We shall see, however, that the number of steps required grows very quickly with n, even if we suppose that the subroutine can always determine $\omega(H)$ exactly and without cost, and that we can always start with the upper bound at the actual value of the chromatic number. (Both these suppositions are of course rather unlikely, since we would be solving *NP*-hard problems [1].)

Given a Zykov tree T for a graph G, and a positive integer k, the corresponding Zykov tree *pruned at k* consists simply of the root G if $\omega(G) \geqq k$ and otherwise is the unique maximal rooted subtree of T containing as internal nodes precisely the nodes H of T with $\omega(H) < k$: the corresponding *pruned Zykov tree* T^* is the tree pruned at $k = \chi(G)$. Any Zykov algorithm for determining the chromatic number of a graph G must explore at least some pruned Zykov tree for G. We shall prove that pruned Zykov trees are usually very large and thus that Zykov algorithms are usually very slow.

Finally note that all logarithms are natural; and for any real number x we let $\lceil x \rceil$ denote the least integer not less than x and $\lfloor x \rfloor$ denote the greatest integer not more than x.

3. Zykov trees. In this section we investigate the sizes of Zykov trees. We have three main reasons for doing this. Firstly the sizes of Zykov trees are of interest if we wish for example to determine the chromatic polynomial of a graph ([2, chap. 15]); secondly some knowledge of the sizes of Zykov trees helps us to interpret results on the sizes of pruned Zykov trees; and thirdly some of the arguments brought up here will be useful later.

The first result in this section shows in particular that every Zykov tree for a given graph has the same size, that is the same number of nodes. Given a graph G let us denote by $C(G)$ the number of proper partitions of G (that is, the number of colorings with "color indifference").

PROPOSITION 3.1. *Every Zykov tree T for a given graph G has $2C(G)-1$ nodes.*

Proof. It is not hard to check that the vertex sets of the leaves of T are in 1–1 correspondence with the proper partitions of G. Alternatively we may use an easy inductive proof.

We are interested in the size of Zykov trees for a graph G_n on n vertices. By the above proposition we may state results in terms of the number $C(G_n)$ of proper partitions of G_n, and we choose to do so. The following proposition requires no proof.

PROPOSITION 3.2. (a) *If G'_n is a subgraph of G_n then $C(G'_n) \geqq C(G_n)$, with equality only if the graphs are the same.*

(b) *Let ϕ_n and K_n denote respectively the null (edge-less) and complete graphs on n vertices. Then $C(K_n)=1$, and $C(\phi_n)$ is simply the number of partitions of a set of size n, so that* (see for example [16])

$$\log C(\phi_n) = n(\log n - \log\log n - 1 + o(1)).$$

The "extreme" properties of $C(G_n)$ are thus easily handled. We may also determine quite closely the "usual" properties.

THEOREM 3.3. (a) *The expected value $E(C_n)$ of $C(G_n)$ for graphs G_n in $\mathcal{G}_n$ satisfies*

$$\log E(C_n) = n(\log n - (2\log(1/q)\log n)^{1/2} - \tfrac{1}{2}\log\log n + O(1)).$$

(b) *With probability $1-o(e^{-n})$*

$$n(\log n - 3(\tfrac{1}{4}\log(1/q)\log^2 n)^{1/3}) \leqq \log C(G_n) \leqq n(\log n - (2\log(1/q)\log n)^{1/2}).$$

Proof. (a) We first show that $\log E(C_n)$ is at least the value given above. Let $d = d(n)$ be an integer-valued function such that $d(n)\to\infty$ as $n\to\infty$ but say $d(n) = O(n/\log n)$. We shall choose d below. Let $\mathcal{R}_n$ be the set of partitions of $\{1,\cdots,n\}$ into $k = \lfloor n/d \rfloor$ sets each of size d and (possibly) the $(n-kd)$ singleton sets $\{kd+1\},\cdots,\{n\}$. Then the number of partitions in $\mathcal{R}_n$ equals

$$\frac{(kd)!}{k!(d!)^k} \geqq \frac{(n-d)!}{(n/d)!(d!)^{n/d}},$$

and the probability that a partition in $\mathcal{R}_n$ is proper equals

$$q^{\binom{d}{2}k} \geqq q^{(1/2)nd}.$$

Hence the logarithm of the expected number of proper partitions in $\mathcal{R}_n$ is at least

$$\begin{aligned}(n-d)\log(n-d) - (n/d)\log(n/d) - (n/d)(d\log d) - \tfrac{1}{2}nd\log(1/q) + O(n)\\ = n(\log n - (\log n)/d - \log d - \tfrac{1}{2}d\log 1/q + O(1)).\end{aligned} \tag{3.1}$$

Now let

$$f_n(x) = (\log n)/x + \log x + \tfrac{1}{2}\log(1/q)$$

for $x>0$. Then $f_n(x)$ achieves a unique minimum for $x>0$ at $x=((2\log(1/q)\log n+1)^{1/2}-1)/\log(1/q)$ and this minimum equals

$$(2\log(1/q)\log n)^{1/2}+\tfrac{1}{2}\log\log n+O(1). \tag{3.2}$$

We set $d(n)=\lfloor(2\log(1/q)\log n)^{1/2}\rfloor$ and find that the right hand side in (3.1) equals the right hand side in the statement of (a).

We now prove the reverse inequality in (a). Let $k=k(n)$ be an integer i such that the expected number of proper partitions into i sets is a maximum. Then of course $E(C_n)$ is at most n times the expected number of proper partitions into k nonempty sets. Let $d=d(n)=n/k$. (Thus d is not necessarily an integer.)

Let Q be a partition of $\{1,\cdots,n\}$ into k sets, of sizes $s_1,\cdots,s_k$. Then as in [8] we see that the probability that Q is proper equals

$$\prod_{i=1}^{k} q^{(1/2)s_i(s_i-1)}=q^{(1/2)(\sum s_i^2-n)}\leqq q^{(1/2)(n^2/k-n)}.$$

Also the number of partitions of $\{1,\cdots,n\}$ into k nonempty sets is at most $k^n/k!$. Hence

$$E(C_n)\leqq n\frac{k^n}{k!}q^{(1/2)(n^2/k-n)},$$

and so

$$\begin{aligned}\log E(C_n)&\leqq n\log k-k\log k-\tfrac{1}{2}(\log(1/q))n^2/k+O(n)\\&=n\log n-n\log d-(n/d)\log n-\tfrac{1}{2}(\log(1/q))nd+O(n)\\&=n(\log n-f_n(d)+O(1)).\end{aligned}$$

But by (3.2)

$$f_n(d)\geqq(2\log(1/q)\log n)^{1/2}+\tfrac{1}{2}\log\log n+O(1).$$

This completes the proof of part (a) of the theorem.

Proof of (b). The right hand inequality here follows from (a) and the standard result that for any nonnegative random variable X and any real number x,

$$E(X)\geqq x\ \text{Prob}\{X\geqq x\}.$$

The left hand inequality follows from Lemma 3.4 below and the discussion preceding it.

Suppose that we have functions $l(n)$ and $r(n)$ with nonnegative integer values. For each positive integer n we let $T_n(l,r)$ be the set of graphs G_n in $\mathcal{G}_n$ such that in some Zykov tree for G_n we may reach a leaf by starting at the root G_n and descending through the tree making at most $l(n)$ left turns and $r(n)$ right turns. If a graph G_n in $\mathcal{G}_n$ is not in $T_n(l,r)$ then certainly every Zykov tree for G_n has at least $\binom{l+r}{r}$ nodes. We wish to find functions $l(n)$ and $r(n)$ so that Prob $T_n(l,r)\to 0$ as $n\to\infty$ and $\binom{l+r}{r}$ is as large as possible.

LEMMA 3.4. *There exist functions $l(n)$ and $r(n)$ such that*

$$\text{Prob } T_n(l,r)=o(e^{-n})\quad \textit{as } n\to\infty,$$

and

$$\log \binom{l+r}{r} \geqq n(\log n - 3(\tfrac{1}{4}\log(1/q)\log^2 n)^{1/3})$$

for n sufficiently large.

Lemma 3.4 may be proved along the lines of the proof in the next section of the more important Lemma 4.7: we do not give a proof here. The lemma is in a sense best possible (see [12]).

4. Pruned Zykov trees. In this section we investigate the size of pruned Zykov trees. We do not manage to find out as much about the pruned trees as we found out about the unpruned trees in the last section, but we are able to prove a greater than exponential lower bound on their size. This result shows that Zykov algorithms for determining the chromatic number of a graph usually require more than exponential time.

We have seen that every Zykov tree for a given graph G_n has the same size (which is less than n^n). Thus certainly if we have to construct a Zykov tree there is no point in spending time choosing a "best" way of branching. The situation is quite different when we look at pruned Zykov trees.

PROPOSITION 4.1. *There is a sequence (G_n^*) of graphs on n vertices such that a smallest pruned Zykov tree for G_n^* has 3 nodes and a largest pruned Zykov tree for G_n^* has $n^{n(1+o(1))}$ nodes.*

Proof. For each integer $k \geqq 5$ let H_k be the pentagon C_5 together with $(k-5)$ vertices adjacent to each other vertex. It is easy to check that $\omega(H_k) = k-3$ and $\chi(H_k) = k-2$; and that every pruned Zykov tree for H_k has exactly 3 nodes. Now for each positive integer n let

$$k = k(n) = \lfloor n(\log n)^{-1/2} \rfloor.$$

Then $k(n) \geqq 5$ for $n \geqq 7$, and we can let G_n^* be the graph H_k together with $(n-k)$ isolated vertices (see Fig. 4.1).

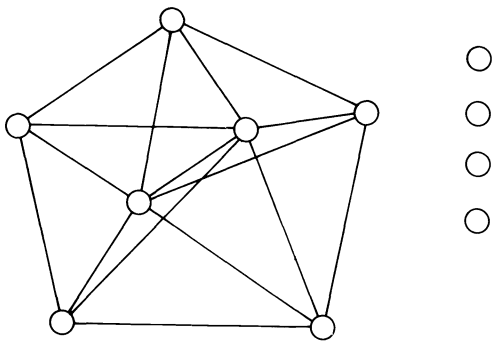


FIG. 4.1. G_{11}^* *is* H_7 *plus* 4 *isolated vertices.*

By branching within the H_k component of G_n^* we see that the size of a smallest pruned Zykov tree for G_n^* is 3. By branching first amongst the isolated vertices of G_n^* we see there is a pruned Zykov tree T_n for G_n^* "containing" as a subtree those nodes K of a Zykov tree for ϕ_{n-k} (the null graph on $n-k$ vertices) with $\omega(K) < \chi(G_n^*) = k-2$. Hence the number $|T_n|$ of nodes in T_n is at least the number of partitions of a set

of size $(n-k)$ into at most $(k-3)$ sets. Let

$$d = d(n) = \lfloor (n-k)/(k-3) \rfloor .$$

Then, much as in the proof of Theorem 3.3(a)

$$|T_n| \geqq \frac{((k-3)d)!}{(k-3)!(d!)^{k-3}} \geqq \frac{(n-2k)!}{n^k} = n^{n(1+o(1))}.$$

We have now seen that the sizes of pruned Zykov trees for a given graph G may vary wildly. For lower bounds on running times of Zykov algorithms we are of course interested in the minimum size, say $Z^*(G)$, of a pruned Zykov tree for G. We investigate below both the "extremal" and the "usual" properties of $Z^*(G_n)$.

Consider first the extremal properties. It is clear that $Z^*(G) = 1$ if and only if $\omega(G) = \chi(G)$. However, it is not clear how large $Z^*(G_n)$ may be for graphs G_n with n vertices.

PROPOSITION 4.2. *There is a sequence (G_n^*) of graphs on n vertices such that*

$$Z^*(G_n^*) \geqq n^{(1+o(1))n/10}.$$

To prove Proposition 4.2 we need one lemma. For each positive integer k we define a graph H_k on $5k$ vertices. The graph H_k is obtained from the pentagon C_5 by "expanding" each vertex into a complete graph on k vertices. More formally we let H_k have disjoint sets of vertices $S_1, \cdots, S_5$ each of size k; and let distinct vertices x in S_i and y in S_j be adjacent if $i-j \equiv 0, \pm 1 \pmod 5$. (See Fig. 4.2.)

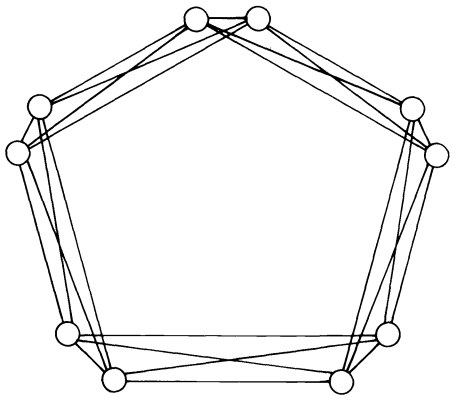


FIG. 4.2. H_2.

LEMMA 4.3. *Let k and x be positive integers with $x \leqq k$. Then every subgraph of H_k with $2k+x$ vertices misses at least kx edges.*

Proof. Let H be an (induced) subgraph of H_k with $2k+x$ vertices such that the number m of edges missing is a minimum. We must show that $m \geqq kx$. For $i = 1, \cdots, 5$ let n_i be the number of vertices of H in S_i. There are two cases to consider.

(a) At least two of the integers n_i are equal to k. We may suppose (without loss of generality) that $n_1 = k$. Then

$$m \geqq k(n_3 + n_4)$$

and so we may suppose that n_3 and n_4 are less than k. But now we may assume that $n_2 = k$, and so

$$m \geqq k(n_3 + n_4 + n_5) = kx.$$

(b) At most one of the n_i equals k. Note first that the n_i are not all equal, for then we would have

$$m = 5\left(\frac{2k+x}{5}\right)^2 \geqq \tfrac{3}{5}k^2 + kx > kx.$$

It follows that we may assume that $n_1 = \max(n_i)$ and $n_1 > n_3$. Note that $n_2, n_5 < k$. Suppose that $n_4 > 0$, and consider the graph H' obtained from H by removing a vertex in S_4 and adding one in S_5. Then the number of edges missing on H' equals m less the positive number $n_1 - n_3$, which contradicts our choice of H. Thus $n_4 = 0$. But now $n_1 > n_4$ and so arguing as above we have $n_3 = 0$. Hence

$$n_2 + n_5 = 2k + x - n_1 \geqq k + x,$$

and so

$$m = n_2 n_5 \geqq kx.$$

This completes the proof of the lemma.

Proof of Proposition 4.2. Let n be an integer at least 5 and let $k = \lfloor n/5 \rfloor$. Let G_n^* be H_k together with $n - 5k$ vertices adjacent to each other vertex. Then $Z^*(G_n^*) = Z^*(H_k)$ and so it suffices to prove that

$$Z^*(H_k) = k^{k((1/2)+o(1))}. \tag{4.1}$$

Note first that every stable set in H_k contains at most two vertices, and so

$$\chi(H_k) \geqq \lceil 5k/2 \rceil.$$

(In fact we have equality but this is not needed.)

For each k let

$$r(k) = \lceil k/2 \rceil - \lceil k/\log k \rceil$$

and

$$l(k) = \lceil k^2/\log k \rceil - 1.$$

Let T be any Zykov tree for H_k, and let K be any node in T which we may reach from the root by descending through the tree making at most $l(k)$ left turns and $r(k)$ right turns. If $\omega(K) \geqq \chi(H_k)$ then H_k must have a subgraph on

$$\chi(H_k) - r(k) \geqq 2k + \lceil k/\log k \rceil$$

vertices which misses at most $l(k)$ edges; and by Lemma 4.3 this is not possible. Thus the node K is in the pruned tree T^* corresponding to T. Hence

$$|T^*| \geqq \binom{l+r}{r} = k^{k((1/2)+o(1))}.$$

This establishes (4.1) and so completes the proof of the proposition.

It is possible to prove that for the sequence (G_n^*) of graphs constructed above we actually have

$$Z^*(G_n^*) = n^{(1+o(1))n/10}.$$

Perhaps every such sequence (G_n^*) of graphs satisfies

$$Z^*(G_n^*) \leqq n^{(1+o(1))n/10},$$

so that Proposition 4.2 is in a sense best possible?

We now move on towards our main result, which concerns the "usual" behavior of the minimum size $Z^*(G_n)$ of a pruned Zykov tree for graphs G_n. We need a number of lemmas. The first concerns the chromatic number of a random graph, and is taken essentially from [8].

LEMMA 4.4. $\text{Prob}\{\chi(G_n) \leqq n \log(1/q)/(2 \log n)\} = o(n^{-k})$ *for any* k *as* $n \to \infty$.

Proof. Recall that the stability number $\alpha(G_n)$ satisfies $\alpha(G_n)\chi(G_n) \geqq n$. Let

$$s = s(n) = \lceil 2 \log n / \log(1/q) \rceil .$$

Then

$$\begin{aligned} &\text{Prob}\{\chi(G_n) \leqq n \log(1/q)/(2 \log n)\} \\ &\quad \leqq \text{Prob}\{\alpha(G_n) \geqq s\} \\ &\quad \leqq \binom{n}{s} q^{\binom{s}{2}} \leqq \left(\frac{ne}{s} q^{(1/2)(s-1)}\right)^s \\ &\quad = \exp s(-\log s + O(1)) \\ &\quad = o(n^{-k}) \quad \text{for any } k. \end{aligned}$$

We now look at the number of edges in a "contraction" of a random graph. Let Q be a family of disjoint subsets of the vertex set of a graph G. We say that Q is *proper* for G if each set in Q is stable. The "contracted" graph G_Q has vertices the sets in Q and an edge between two of these sets if there is an edge in G between some two vertices one from each set. Clearly G_Q may be formed from G by a sequence of vertex-contractions if and only if Q is proper for G. We are interested in the number of edges we are likely to have in G_Q.

Given two random variables X and Y we write $X \leqq Y$ *in distribution* if $F_X(t) \geqq F_Y(t)$ for each real number t, where F_X and F_Y are the distribution functions of X and Y respectively.

LEMMA 4.5. *Suppose that* X, Y, Z *are random variables, that* $X \leqq Y$ *in distribution, and that the pairs* X, Z *and* Y, Z *are independent. Then* $X + Z \leqq Y + Z$ *in distribution.*

Proof. For any real number t,

$$\begin{aligned} F_{X+Z}(t) &= \int F_X(t-u)\, dF_Z(u) \\ &\geqq \int F_Y(t-u)\, dF_Z(u) = F_{Y+Z}(t). \end{aligned}$$

Let m and n be positive integers, and let q be a real number with $0 < q < 1$. Let $B(q)$ be a binomial random variable with parameters $\binom{m}{2}$ and $1-q$, and for each partition Q of $\{1, \cdots, n\}$ let the random variable $N(Q) = N(Q, n, 1-q)$ be the number of edges in the contracted graph G_Q, for graphs G in $\mathcal{G}_n$ with edge-probability $1-q$.

LEMMA 4.6. *For each partition* Q *of* $\{1, \cdots, n\}$ *into* m *sets*

$$N(Q) \leqq B(q^{(n/m)^2}) \quad \textit{in distribution.} \tag{4.2}$$

Proof. We may of course assume that $m \geqq 2$. We prove first that

$$N(Q) \leqq B(q^{\lceil n/m \rceil^2}) \quad \text{in distribution.} \tag{4.3}$$

(The inequality (4.3) is in fact good enough for the purposes of this paper.) Let

$Q=(S_1, \cdots, S_m)$ be a partition of $\{1, \cdots, n\}$ into m sets, and suppose that $|S_1|+1 \leqq |S_2|-1$. Let Q' be the partition obtained from Q by switching one element from S_2 to S_1. In order to prove (4.3) it is sufficient to prove that

$$N(Q) \leqq N(Q') \quad \text{in distribution.} \tag{4.4}$$

For $1 \leqq i < j \leqq m$ let $X_{ij}=1$ if S_i and S_j are adjacent in G_Q and let $X_{ij}=0$ otherwise. The random variables X_{ij} are of course all independent. Define independent random variables X'_{ij} from Q' in a similar way. Note that $X'_{ij}=X_{ij}$ for $i>2$, and let the random variable Z be the sum of all such X_{ij} (or X'_{ij}). Then

$$N(Q)=X_{12}+\sum_{j=3}^{m}(X_{1j}+X_{2j})+Z,$$

and

$$N(Q')=X'_{12}+\sum_{j=3}^{m}(X'_{1j}+X'_{2j})+Z.$$

But it is straightforward to prove that

$$X_{12} \leqq X'_{12} \quad \text{in distribution,}$$

and for $j=3, \cdots, m$

$$X_{1j}+X_{2j} \leqq X'_{1j}+X'_{2j} \quad \text{in distribution.}$$

The result (4.3) now follows by repeated use of Lemma 4.5.

We now use (4.3) to prove (4.2). Given a set S of positive integers and a positive integer k let kS be the set of positive integers i such that $\lceil i/k \rceil$ is in S. Given a partition $Q=(S_1, \cdots, S_m)$ of $\{1, \cdots, n\}$ let kQ be the partition $(kS_1, \cdots, kS_m)$ of $\{1, \cdots, kn\}$. (For example if Q is the partition $(\{1,2\},\{3\})$ of $\{1,2,3\}$ then $2Q$ is the partition $(\{1,2,3,4\},\{5,6\})$ of $\{1,2,\cdots,6\}$.)

It is easy to see that for each positive integer k

$$N(Q, n, 1-q)=N(kQ, kn, 1-q^{1/k^2}) \quad \text{in distribution.}$$

Hence by (4.3) for each k

$$N(Q, n, 1-q) \leqq B(q^{\lceil kn/m \rceil^2/k^2}) \quad \text{in distribution.}$$

But $\lceil kn/m \rceil^2/k^2 \to (n/m)^2$ as $k \to \infty$, and so (4.2) holds. This completes the proof of Lemma 4.6.

We need one more lemma in order to prove the main result. Suppose that we have a positive constant t and functions $l(n)$ and $r(n)$ with nonnegative integer values. For each positive integer n let $T_n^t(l, r)$ be the set of graphs G_n in $\mathcal{G}_n$ such that in some Zykov tree for G_n we may reach a leaf or node H with $\omega(H) \geqq t\chi(G_n)$ by starting at the root and descending through the tree making at most $l(n)$ left turns and $r(n)$ right turns. (Compare with the definition of $T_n(l, r)$ preceding Lemma 3.4 in the last section.) If a graph G_n in $\mathcal{G}_n$ is not in $T_n^t(l, r)$ then certainly every Zykov tree for G_n has at least $\binom{l+r}{r}$ nodes H with $\omega(H) < t\chi(G_n)$. In the case $t=1$ we see that if G_n is not in $T_n^1(l, r)$ then every pruned Zykov tree for G_n has at least $\binom{l+r}{r}$ nodes. We wish to find functions $l(n)$ and $r(n)$ such that Prob $T_n^t(l, r) \to 0$ as $n \to \infty$ and $\binom{l+r}{r}$ is as

large as possible.

LEMMA 4.7. *For any positive constant t there exist functions $l(n)$ and $r(n)$ such that as $n \to \infty$*

$$\text{Prob } T_n^t(l, r) = o(n^{-k}) \quad \textit{for any } k$$

and

$$\log \binom{l+r}{r} \sim tn(\tfrac{1}{27} \log (1/q) \log n)^{1/2}.$$

We may take l and r so that

$$l(n) = n^{5/3+o(1)}$$

and

$$r(n) = \lfloor tn(\log (1/q)/(12 \log n))^{1/2} \rfloor.$$

Lemma 4.7 above of course is similar to Lemma 3.4 in the last section, and we noted there that that lemma is in a sense best possible. Lemma 4.7 is also in a sense best possible [12].

Proof. Let k be any positive integer. Let $l(n)$ and $r(n)$ be functions with nonnegative integer values, which we shall choose later. Let

$$b(n) = \lfloor tn \log (1/q)/(2 \log n) \rfloor$$

and let

$$B_n = \{G \in \mathcal{G}_n : \chi(G) < n \log (1/q)/(2 \log n)\}.$$

Let $C_n(l, r)$ be the set of graphs G in $\mathcal{G}_n$ such that in some Zykov tree for G we may reach a leaf or node H with $\omega(H) \geqq b(n)$ by starting at the root and descending through the tree making (as usual) at most $l(n)$ left turns and $r(n)$ right turns. Then

$$T_n^t(l, r) \subseteq B_n \cup C_n(l, r).$$

By Lemma 4.4 Prob $B_n = o(n^{-k})$ as $n \to \infty$. Thus we wish to choose functions $l(n)$ and $r(n)$ such that

$$\text{Prob } C_n(l, r) = o(n^{-k}) \quad \text{as } n \to \infty \tag{4.5}$$

and $\binom{l+r}{r}$ is as large as possible.

Let $\mathcal{R}$ be the collection of all families of b disjoint subsets of $\{1, \cdots, n\}$ with union containing at most $r+b$ elements. For each family Q in $\mathcal{R}$ let T_Q be the set of graphs G in $\mathcal{G}_n$ such that the "contracted" graph G_Q misses at most l edges. Now if G is a graph in $C_n(l, r)$ then some graph obtained from G by performing at most r vertex-contractions contains a subgraph on b vertices missing at most l edges; and so $G \in T_Q$ for some family Q (proper for G) in $\mathcal{R}$. Hence

$$C_n(l, r) \subseteq \cup \{T_Q : Q \in \mathcal{R}\}. \tag{4.6}$$

Next we find an upper bound for the Prob T_Q. Let N be a binomial random variable with parameters $\binom{b}{2}$ and q^{x^2}, where $x = r/b + 1$. By Lemma 4.6 for each Q in $\mathcal{R}$

$$\text{Prob } T_Q \leqq \text{Prob } \{N \leqq l\}. \tag{4.7}$$

Now let $l(n) = \lfloor \frac{1}{2}E(N) \rfloor$. Then by a standard inequality concerning the binomial distribution (see for example [6, pp. 17, 18])

$$\text{Prob}\{N \leqq l\} \leqq \exp(-(\tfrac{1}{8})E(N)). \tag{4.8}$$

But $\mathcal{R}$ of course contains at most n^n families Q. Hence by (4.6), (4.7) and (4.8)

$$\text{Prob } C_n(l, r) \leqq n^n \exp(-(\tfrac{1}{8})E(N)). \tag{4.9}$$

Let

$$r(n) = \lfloor un(\log(1/q)/\log n)^{1/2} \rfloor$$

for some constant u to be chosen with $0 < u < \frac{1}{2}t$. Then

$$x(n) = (1 + o(1))(2u/t)((\log n)/\log(1/q))^{1/2}$$

and so

$$\begin{aligned} E(N) &= \binom{b}{2} q^{x^2} \\ &= \exp(2 \log n - (4u^2/t^2) \log n + o(\log n)) \\ &= n^{(2-4u^2/t^2+o(1))}. \end{aligned}$$

But $4u^2/t^2 < 1$ and so (4.5) holds by (4.9).

It remains to choose u to maximize $\binom{l+r}{r}$. But

$$\begin{aligned} \log \binom{l+r}{r} &= r(\log l - \log r + O(1)) \\ &= un(\log(1/q)/\log n)^{1/2}(1 - 4u^2/t^2 + o(1)) \log n \\ &= (u - 4u^3/t^2 + o(1))(\log(1/q) \log n)^{1/2} n. \end{aligned}$$

The maximum value of $u - 4u^3/t^2$ for $u > 0$ is attained at $u = 12^{(-1/2)}t < \frac{1}{2}t$. Thus we give u this value, and find that $\log \binom{l+r}{r}$ is as in the statement of the lemma. The functions $l(n)$ and $r(n)$ are now also as in the lemma. This completes the proof.

Suppose that for each positive integer n we have a subset A_n of $\mathcal{G}_n$, the set of all graphs on $\{1, \cdots, n\}$. We shall make statements like "the event A_n occurs for almost all G_n" if the sum of the probabilities that each A_n fails to occur is convergent. (This definition corresponds to embedding all our probability spaces $\mathcal{G}_n$ in a single space and using a Borel–Cantelli lemma—compare with [8].) Thus for example by Lemma 4.4, for almost all graphs G_n we have

$$\chi(G_n) \geqq n \log(1/q)/(2 \log n).$$

From Lemma 4.7 and the discussion preceding it we may now deduce immediately our main result.

THEOREM 4.8. *If t is a positive constant then for almost all graphs G_n every Zykov tree for G_n contains at least*

$$\exp[(1 + o(1))tn(\tfrac{1}{27} \log(1/q) \log n)^{1/2}]$$

nodes H such that $\omega(H) < t\chi(G_n)$.

Recall that $Z^*(G)$ is the minimum size of a pruned Zykov tree for a graph G.

COROLLARY 4.9. *For almost all graphs* G_n

$$Z^*(G_n) \geqq \exp[(1+o(1))n(\tfrac{1}{27}\log(1/q)\log n)^{1/2}].$$

COROLLARY 4.10. *For almost all graphs* G_n *the number of steps needed by any Zykov algorithm to determine* $\chi(G_n)$ *is at least the quantity given above.*

In the case $p=q=\frac{1}{2}$ Corollary 4.9 yields

COROLLARY 4.11. *The proportion of graphs* G_n *on n vertices such that*

$$Z^*(G_n) \geqq \exp(.157n\log^{1/2} n)$$

tends to 1 *as* $n\to\infty$.

Corollary 4.10 above shows that the time taken by Zykov algorithms for determining chromatic numbers grows faster than exponentially with the number of vertices; and thus that these algorithms are slower asymptotically than the algorithm considered by E. L. Lawler [10].

M. R. Garey and D. S. Johnston [7] have shown that the problem of determining the chromatic number of a graph to within a factor less than 2 is *NP*-hard. By analogy one might possibly have expected some effect in Theorem 4.8 at any $t=\frac{1}{2}$, but none is apparent (see also Corollary 5.1 below).

5. Lengths of proofs. The above results may be phrased in terms of the lengths of certain kinds of proof which determine chromatic numbers or which establish lower bounds for chromatic numbers. We then obtain results concerning chromatic numbers which are similar in spirit to recent results of V Chvátal [4] concerning stability numbers. Indeed this paper was initially motivated by discussions with Chvátal concerning his results.

If k is an integer at least as great as $\chi(G)$ then there is a short proof that $\chi(G)\leqq k$—namely we may exhibit a proper coloring of G using at most k colors. In general such a proof is hard to find but it must of course exist. However, if k is at most $\chi(G)$ then it is not clear if there is necessarily a short proof of this fact.

Consider the following proof system for establishing lower bounds for chromatic numbers. A *statement* is simply a pair (G, b) where G is a graph and b is a nonnegative integer. (Such a statement is to be interpreted as the inequality $\chi(G)\geqq b$, which may of course be false.) A *recursive proof* of (G, b) is a sequence of statements (G_k, b_k) $(k=1,\cdots,m)$ such that $(G_m, b_m)=(G, b)$ and for each $1\leqq k\leqq m$ either $\omega(G_k)\geqq b_k$ or there are integers $1\leqq i, j<k$ such that G_i and G_j are a pair of reduced graphs for G_k and $b_k\leqq\min(b_i, b_j)$. We call the integer m the *length* of the proof. If there is a recursive proof of (G, b) then by (2.1) and (2.2) we have $\chi(G)\geqq b$; and conversely if $\chi(G)\geqq b$ then we can construct a recursive proof of (G, b) from any Zykov tree for G pruned at b. In fact if $\chi(G)\geqq b$ there is close correspondence between recursive proofs of (G, b) and Zykov trees for G pruned at b. In particular the minimum length of a recursive proof of (G, b) equals the minimum size of a Zykov tree for G pruned at b.

From Theorem 4.8 we obtain

COROLLARY 5.1. *Let* $0<t\leqq 1$. *Then for almost all graphs* G_n *on n vertices, every recursive proof of* $(G_n, t\chi(G_n))$ *has length at least* $c^{n(\log n)^{1/2}}$, *where c is a constant* >1.

From Corollary 4.11 we obtain

COROLLARY 5.2. *Consider the property for graphs* G_n *on n vertices that every recursive proof of* $(G_n, \chi(G_n))$ *has length at least*

$$\exp(.157n\log^{1/2} n).$$

The proportion of graphs on n vertices with this property tends to 1 *as* $n\to\infty$.

The reader is reminded that further related results are given in [12].

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] C. BERGE, *Graphs and Hypergraphs*, North-Holland, London, 1973.
[3] N. CHRISTOFIDES, *An algorithm for the chromatic number of a graph*, Comput. J., 14 (1971), pp. 38–39.
[4] V. CHVÁTAL, *Determining the stability number of a graph*, this Journal, 6 (1977), pp. 643–662.
[5] D. G. CORNEIL AND B. GRAHAM, *An algorithm for determining the chromatic number of a graph*, this Journal, 2 (1973), pp. 311–318.
[6] P. ERDOS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York and London, 1974.
[7] M. R. GAREY AND D. S. JOHNSON, *The complexity of near-optimal graph coloring*, J. Assoc. Comput. Mach., 23 (1976), pp. 43–49.
[8] G. R. GRIMMETT AND C. J. H. MCDIARMID, *On coloring random graphs*, Math. Proc. Camb. Philos. Soc., 77 (1975), pp. 313–324.
[9] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.
[10] E. L. LAWLER, *A note on the complexity of the chromatic number problem*, Information Processing Lett., 5 (1976), pp. 66–67.
[11] C. J. H. MCDIARMID, *Determining the chromatic number of a graph*, Technical Report STAN-CS-76-576, Stanford University, 1976.
[12] ———, *Determining the chromatic number of a graph II*, to appear.
[13] D. J. A. WELSH AND M. B. POWELL, *An upper bound to the chromatic number of graph and its application to time-tabling problems*, Comput. J., 10 (1967), pp. 85–86.
[14] D. C. WOOD, *A technique for coloring a graph applicable to large scale time-tabling problems*, Ibid., 12 (1969), pp. 317–319.
[15] A. A. ZYKOV, *On some properties of linear complexes*, Mat. Sb., 24 (1949), pp. 163–188; English translation, Amer. Math. Soc. Translation 79, 1952.
[16] N. DE BRUIJN, *Asymptotic Methods in Analysis*, 2nd ed., North-Holland, 1961.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0002 $01.00/0

A MINIMUM LINEAR ARRANGEMENT ALGORITHM FOR UNDIRECTED TREES*

YOSSI SHILOACH†

Abstract. The minimum linear arrangement problem is a special case of more general placement problems which are discussed in Hanan and Kurtzberg [5] and might occur in solving wiring problems as well as many other placement problems. It is also a special case of the quadratic assignment problem [5] and has a lot in common with job sequencing problems (Adolphson and Hu [1, § 4]).

The minimum linear arrangement problem for general undirected graphs is *NP* complete as shown in Garey et al. [2]. The corresponding problem for acyclic directed graphs is also *NP* complete (Evan and Shiloach [4]). D. Adolphson and T. C. Hu [1] solved the problem for rooted trees by an $O(n \log n)$ algorithm. In this paper we solve the problem for undirected trees by an $O(n^{2.2})$ algorithm.

Key words. algorithm, linear arrangement, trees

1. Introduction. Let $G=(V, E)$ be an undirected loopless graph with n vertices. A *linear arrangement* π (henceforth denoted by *arr.*) is a 1-1 mapping of V onto $\{1, \cdots, n\}$. $C[\pi, G]$—the *cost* of π, is defined by:

$$C[\pi, G]=\sum_{(v_i,v_j)\in E}|\pi(v_i)-\pi(v_j)|.$$

π is a *minimum arr.* of G if there is no other arr. of G with smaller cost.

Finding a minimum arr. for general undirected graphs is *NP* complete as shown in [2] and [4]. In this paper we present an $O(n^{2.2})$ algorithm solving the minimum arr. problem for undirected trees.

In § 2 we provide the definitions for the basic concepts which are necessary in order to understand the algorithm, which is given in § 3.

In § 4 we develop the basic tools which we used in the validity proof of the algorithm. These tools are three operators on arrangements (i.e. they transform a given arr. into another one). Under certain conditions, these operators do not increase the cost. We introduce the theorems which specify these conditions.

Section 5 provides the validity proof of the algorithm. It is shown there that there exists a minimum arr. having one of two specific types. This is done by applying the non-increasing-cost operators on an arbitrary arr. transforming it into another arr. of one of these two types, without increasing the cost.

In § 6 we analyze the complexity of the algorithm. Its polynomiality is due to the fact that the search for a minimum arr. is done on a very restricted set of arrangements.

2. Basic concepts.

2.1. Types of arrangements.

DEFINITION 2.1.1. Let π be an arr. of a given tree T; then $\bar{\pi}$ denotes the arr. obtained from π by reversing the order of the vertices (i.e. $\bar{\pi}(u)<\bar{\pi}(v)$ iff $\pi(u)>\pi(v)$).

DEFINITION 2.1.2. Let v_* be a vertex of T. Deleting v_* and its incident edges from T, yields several subtrees of T. Each of them is called *a subtree of T mod.* v_*. For each edge (v, v_*) there is a unique subtree of T mod. v_*, say T', such that $v \in T'$. The vertex v is *the root of* T' *mod.* v_*. (See Fig. 1).

* Received by the editors March 11, 1976, and in final revised form February 16, 1978.
† Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

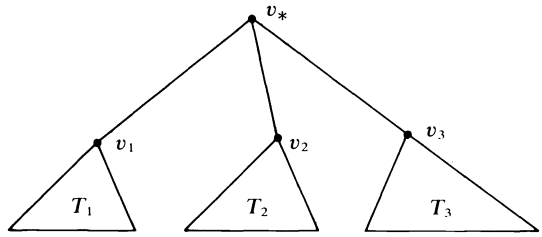


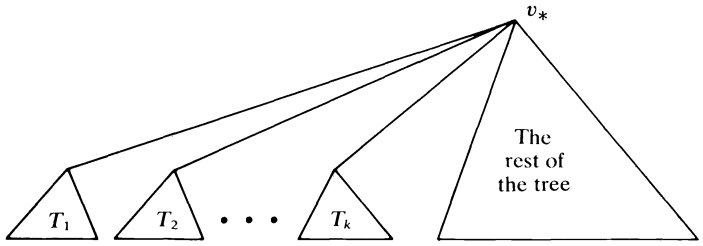
FIG 1. *T_1 T_2 and T_3 are subtrees of T mod, v_*, v_1, v_2 and v_3 are their roots mod. v_* respectively.*

DEFINITION 2.1.3. Let $T_1, \cdots, T_k$ be subtrees of T mod. v_* and let π be an arr. of T. π is of type $(T_1, \cdots, T_k|v_*)$ if the following holds:

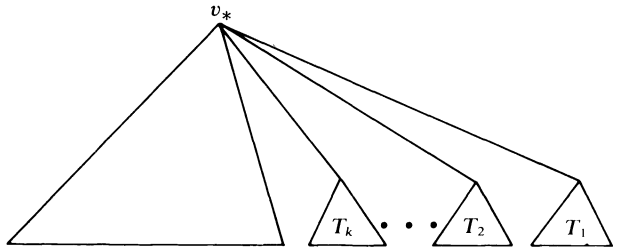
$$\sum_{i=1}^{j-1} n_i < \pi(v) \leqq \sum_{i=1}^{j} n_i < \pi(v_*) \tag{2.1}$$

for all $v \in T_j$ and for all $1 \leqq j \leqq k$, Here n_i denotes the number of vertices of T_i, $i = 1, \cdots, k$.[1]

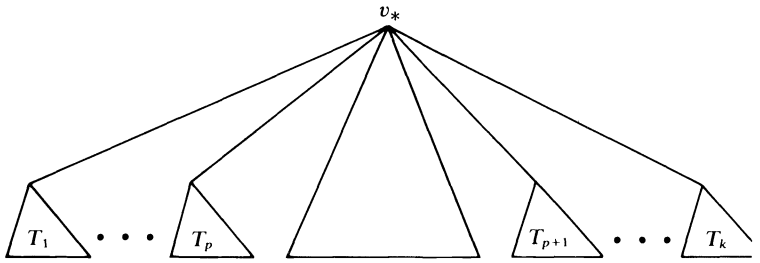
π is of the type $(v_*|T_k, \cdots, T_1)$ if $\bar{\pi}$ is of type $(T_1, \cdots, T_k|v_*)$. π is of type $(T_1, \cdots, T_p|v_*|T_{p+1}, \cdots, T_k)$ if it is of both types $(T_1, \cdots, T_p|v_*)$ and $(v_*|T_{p+1}, \cdots, T_k)$. These three types are illustrated in Fig. 2.



(a) *An arrangement of type* $(T_1, \cdots, T_k|v_*)$.



(b) *An arrangement of type* $(v_*|T_k, \cdots, T_1)$.



(c) *An arrangement of type* $(T_1, \cdots, T_p|v_*|T_{p+1}, \cdots, T_k)$.

FIG. 2

[1] Henceforth, whenever a tree T_α is concerned, n_α will denote its number of vertices for any index α.

DEFINTION 2.1.4. Let $T_1, \cdots, T_k$ be subtrees of T mod. v_*. $T-(T_1, \cdots, T_k)$ denotes the tree obtained from T by removing the vertices of $T_1, \cdots, T_k$ and their incident edges.

2.2. Anchored trees. Let T_0 be a subtree of T mod. v_* and let v_0 be its root mod. v_*. Assume that we know that there exists a minimum arr. of T of type $(T_0|v_*)$. Providing minimum arrangements for T_0 and $T-T_0$ separately is wrong, since we have no control on the length of the edge (v_0, v_*).

In Fig. 3 we split (v_0, v_*) into three segments.

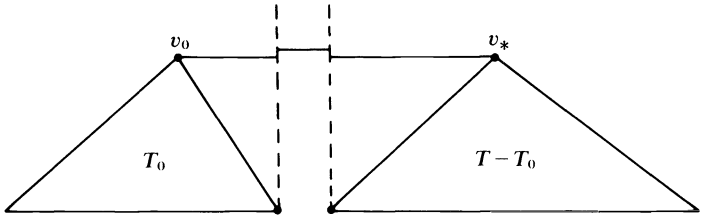


FIG. 3

The segment in the middle is of one unit length. The left segment is connected to T_0 at v_0 which is its left endpoint. It covers the distance between v_0 and the right-most vertex of T_0. This segment is a *right anchor* of T_0. The segment in the right is connected to $T-T_0$ at v_* which is its right endpoint. It covers the distance between v_* and the left-most vertex of $T-T_0$. This segment is a *left anchor* of $T-T_0$.

This informal description motivates the following definition: Let T be an n-vertix tree, let $v \in T$ and let π be an arr. of T. T is called a *right anchored tree* (at v) and denoted by $\vec{T}(v)$ when its cost is defined by:

$$C[\pi, \vec{T}(v)] = C[\pi, T] + n - \pi(v). \tag{2.2}$$

It is called a *left anchored tree* (at v) and denoted by $\overleftarrow{T}(v)$ when its cost is defined by:

$$C[\pi, \overleftarrow{T}(v)] = C[\pi, T] + \pi(v) - 1. \tag{2.3}$$

Note that $n-\pi(v)$ in (2.2) and $\pi(v)-1$ in (2.3) stand for the length of the right and left anchors of T, respectively.

Note that if we know that a tree T has a minimum arr. of type $(T_0|v_*)$, it is sufficient to find minimum arrangements for $\vec{T}(v_0)$ and for $\overleftarrow{T-T_0}(v_*)$ separately (see Theorem 3.1.2.a)). This fact suggests two new problems, namely, finding minimum arrangements for right and left anchored trees. These problems are equivalent since by reversing the order of the vertices a right-anchored tree becomes a left-anchored while the cost is unchanged.

The problem of finding a minimum arr. for right-anchored trees will be solved simultaneously with the corresponding problem for free (not anchored) trees.

2.3. The central vertex theorem.

THEOREM 2.3. *There exists a vertex v_* satisfying: If $T_0, \cdots, T_k$ are all the subtrees of T mod. v_* then*

$$n_i \leqq \left\lfloor \frac{n}{2} \right\rfloor \quad \text{for } i = 0, \cdots, k. \tag{2.4}$$

Proof. By removing an edge (v_i, v_j) from T, we split it into two subtrees T_i and T_j, such that $v_i \in T_i$ and $v_j \in T_j$. For each edge (v_i, v_j) let $d_{i,j} = |n_i - n_j|$. Let

$$d_{i_0, j_0} = \min_{(v_i, v_j) \in E} d_{i,j}. \tag{2.5}$$

Assuming that $n_{j_0} \geqq n_{i_0}$ we set $v_* = v_{j_0}$. T_{i_0} is a subtree of T mod. v_* and $n_{i_0} \leqq \lfloor \frac{n}{2} \rfloor$ since $n_{i_0} + n_{j_0} = n$ and $n_{j_0} \geqq n_{i_0}$. Let $T_1, \cdots, T_k$ be all the other subtrees of T mod. v_*, numbered so that $n_1 \geqq n_2 \geqq \cdots \geqq n_k$ It is enough to show that $n_{i_0} \geqq n_1$.

Assume to the contrary that $n_{i_0} < n_1$.

$$d_{i_0, j_0} = n - 2n_{i_0}. \qquad d_{1, j_0} = |(n - n_1) - n_1|.$$

If $d_{1, j_0} = n - 2n_1 < n - 2n_{i_0}$ we contradict (2.5). T_1 and T_{i_0} are vertex-disjoint subtrees of T and v_* does not belong to any of them, hence: $n_{i_0} < n - n_1$ and $n_1 < n - n_{i_0}$. Thus $(n - n_{i_0}) - n_{i_0} > n_1 - (n - n_1)$. Therefore, assuming that $d_{1,j} = 2n_1 - n$ also yields a contradiction to (2.5). Q.E.D.

3. The main theorem and the algorithm. In this section we present two theorems which motivate the algorithm which follows them. We deal with free and (right) anchored trees simultaneously, using a parameter α. $\alpha = 0$ for free trees and $\alpha = 1$ for anchored trees. Theorem 3.1.1 is the heart of this paper. Its proof (for the free trees version) is given in § 5. The main differences between free and anchored trees are discussed in § 3.3.

It is assumed through this section that v_* is a vertex of T which satisfies (2.4) if T is a free tree. If T is an anchored tree, v_* denotes the vertex in which the anchor is connected to T. $T(\alpha)$ denotes:

$$\begin{cases} T & \text{if } \alpha = 0, \\ \vec{T}(v_*) & \text{if } \alpha = 1. \end{cases}$$

In both cases $T_0, \cdots, T_k$ are all the subtrees of $T(\alpha)$ mod. v_*. They are numbered so that $n_0 \geqq n_1 \geqq \cdots \geqq n_k$. The roots of $T_0, \cdots, T_k$ mod. v_* are $v_0, \cdots, v_k$ respectively.

3.1. The motivating theorem. Let p_α denote the greatest integer satisfying:

$$n_i > \left\lfloor \frac{n_0 + 2}{2} \right\rfloor + \left\lfloor \frac{n_* + 2}{2} \right\rfloor \quad \text{for } i = 1, \cdots, 2p_\alpha - \alpha, \text{ where}$$

$$n_* = n - \sum_{i=0}^{2p_\alpha - \alpha} n_i. \tag{3.1}$$

THEOREM 3.1.1. a). *If $p_\alpha = 0$ then $T(\alpha)$ has a minimum arr. of type $(T_0 | v_*)$.*

b) *If $p_\alpha > 0$ then $T(\alpha)$ has a minimum arr. of one of the following types*:

A: $(T_0 | v_*)$,

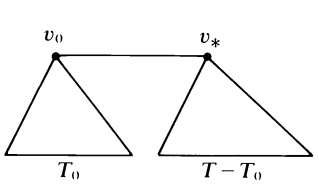
B: $(T_1, T_3, \cdots, T_{2p_\alpha - 1} | v_* | T_{2p_\alpha - 2\alpha}, \cdots, T_4, T_2)$.

Figure 4(a) and 4(b) illustrate arrangements of types A and B for T and $\vec{T}(v_*)$ respectively.

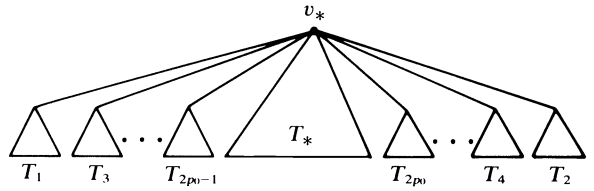
Let $T_* = T(\alpha) - (T_1, \cdots, T_{2p_\alpha - \alpha})$ and let:

$$S_0 = (n_3 + n_4) + 2(n_5 + n_6) + \cdots + (p_0 - 1)(n_{2p_0 - 1} + n_{2p_0}) + p_0(n_* + 1),$$

$$S_1 = (n_2 + n_3) + 2(n_4 + n_5) + \cdots + (p_1 - 1)(n_{2p_1 - 2} + n_{2p_1 - 1}) + p_1(n_* + 1) - 1.$$

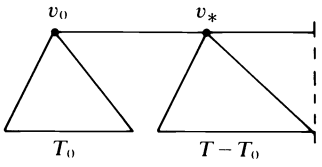


Type A



Type B

FIG. 4(a)



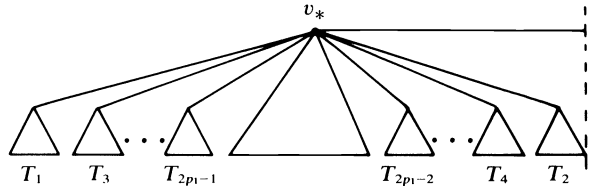


FIG. 4(b)

THEOREM 3.1.2. a) *If π is an arr. of $T(\alpha)$ of type* A *then*

$$C_\alpha(A) = C[\pi, T(\alpha)] = \begin{cases} C[\pi, \vec{T}_0(v_0)] + C[\pi, \overleftarrow{T-T_0}(v_*)] + 1 & \text{if } \alpha = 0. \\ C[\pi, \vec{T}_0(v_0)] + C[\pi, T-T_0] + n - n_0 & \text{if } \alpha = 1. \end{cases}$$

(b) *If π is an arr. of $T(\alpha)$ of type* B, *then*

$$C_\alpha(B) = C[\pi, T(\alpha)] = \sum_{\substack{i=1 \\ i \text{ is odd}}}^{2p_\alpha - 1} C[\pi, \vec{T}_i(v_i)] + \sum_{\substack{i=1 \\ i \text{ is even}}}^{2p_\alpha - 2\alpha} C[\pi_i, \overleftarrow{T}_i(v_i)] + C[\pi, T_*] + S_\alpha.$$

The proof of Theorem 3.1.2 is by a straightforward calculation which follows the elementary definitions. We omit the details. Theorem 3.1.2 yields the following important corollary:

COROLLARY 3.2.1. a) *If π is a minimum arr. of $T(\alpha)$ of type* A *then*: π/T_0 (π, *restricted to* T_0) *is a minimum arr. of $\vec{T}_0(v_0)$ and $\pi/T-T_0$ is a minimum arr. of $\overleftarrow{T-T_0}(v_*)$ (or $T-T_0$ in case $\alpha = 1$).*

b) *If π is a minimum arr. of $T(\alpha)$ of type* B *then*: π/T_i *are minimum arrangements of $\vec{T}_i(v_i)$ for $i = 1, 3, \cdots, 2p_\alpha - 1$ and of $\overleftarrow{T}_i(v_i)$ for $i = 2, 4, \cdots, 2p_\alpha - 2\alpha$. π/T_* is a minimum arr. of T_*.*

Proof. The proof of a) is immediate. b) is true since S_α is independent of π.

3.2. The algorithm. The algorithm is a straightforward implementation of Theorem 3.1.1 and Corollary 3.1.2. Using the parameter α, the algorithms for free and anchored trees are combined together. Each of them is recursive and uses both of them as subroutines for smaller trees.

ALGORITHM.

1. Find a vertex v_* satisfying (2.4). (See Theorem 2.3.) (In an anchored tree, v_* is the vertex at which the anchor is connected to the tree.)
2. Find minimum arrangements π_0 for $\vec{T}(v_0)$ and π_0' for $\overleftarrow{T-T_0}(v_*)$ (or for $T-T_0$ if T is anchored.).
3. Determine $C_\alpha(A)$. (See Theorem 3.1.2.)
4. Determine the value of p_α. If $p_\alpha = 0$ go to 9.

5. Find minimum arrangements π_i, $i=1,\cdots,2p_\alpha-\alpha$, for $\vec{T}_i(v_i)$ $i=1,3,\cdots,2p_\alpha-1$ and for $\bar{T}_i(v_i)$ $i=2,4,\cdots,2p_\alpha-2\alpha$ and a minimum arr. π_* for $T-(T_1,\cdots,T_{2p_\alpha-\alpha})$.

6. Determine $C_\alpha(B)$.

7. If $C_\alpha(A)\leqq C_\alpha(B)$ go to 9.

8. The arrangement π_m of type $(T_1, T_3,\cdots, T_{2p_\alpha-1}|v_*|T_{2p_\alpha-2\alpha},\cdots, T_4, T_2)$ determined by π_i on T_i for $i=1,\cdots,2p_\alpha-\alpha$ and by π_* on $T-(T_1,\cdots,T_{2p_\alpha-\alpha})$ is a minimum arr. of T. $C[\pi, T(\alpha)]=C_\alpha(B)$.

Stop.

9. The arrangement π_m of type $(T_0|v_*)$ determined by π_0 on T_0 and by π_0' on $T-T_0$ is a minimum arr. of T. $C[\pi, T(\alpha)]=C_\alpha(A)$.

Stop.

3.3. The main differences between free and anchored trees.

1. If T is free, then v_* is its "central" vertex. If T is anchored, then v_* is the connection point of T with its anchor. The centrality of v_* is necessary to prove Theorem 3.1.1. a) for $\alpha=0$.

When $\alpha=1$, this statement can be proved using the fact that v_* is the left end-point of the right anchor.

Part a) of Theorem 3.1.1 is a crucial statement for the complexity estimations in § 6.

2. Type B is different in the cases $\alpha=0$ and $\alpha=1$. When $\alpha=0$ the number of the T_i's to the left of v_* is equal to the number of T_i's to the right of v_*. When $\alpha=1$ we have one more subtree to the left of v_* than to the right of v_*. This is caused by the right anchor, playing the role of a subtree on the right-hand side of v_*.

4. Non-increasing-cost operators. The N.I.C. operators which are defined on the set of all arrangements are used to transform an arbitrary arr. into another one of type A or B without increasing the cost.

Let $v_*\in T$, let T_0 be a subtree of T mod. v_* and let π be an arr. of T. $L_1(\pi, T_0)$ $[R_1(\pi, T_0)]$ is the arr. of type $(T_0|v_*)$ $[(v_*|T_0)]$ in which the internal order of the vertices of T_0 and $T-T_0$ is preserved as it is in π. Denoting the length of an edge e in a given arr. π by $l_\pi(e)$, it is easily seen that:

$$C[L_1(\pi, T_0)]=C[\pi, T_0]+C[\pi, T-T_0]+l_{L_1(\pi,T_0)}((v_0, v_*)).$$

(As usual, v_0 denotes the root of T_0 mod. v_*. See Fig. 5(a).) $l_{L_1(\pi,T_0)}((v_0, v_*))$ is exactly 1+the number of vertices between v_0 and v_*. This number may be decreased when we reverse (if necessary) the order of the vertices of T_0 and/or the order of the vertices of $T-T_0$ so that the following inequalities are satisfied:

$$|\{v|v\in T_0 \text{ and } \pi(v)>\pi(v_0)\}|\leqq|\{v|v\in T_0 \text{ and } \pi(v)<\pi(v_0)\}|,$$

$$|\{v|v\in T-T_0 \text{ and } \pi(v)<\pi(v_*)\}|\leqq|\{v|v\in T-T_0 \text{ and } \pi(v)>\pi(v_*)\}|.$$

The modified arr. is denoted by $L_2(\pi, T_0)$ (see Fig. 5(c)). $R_2(\pi, T_0)$ is defined in a symmetric way.

The definition of $L_2(\pi, T_0)$ and $R_2(\pi, T_0)$ implies that:

$$l_{L_2(\pi,T_0)}((v_0, v_*))=l_{R_2(\pi,T_0)}((v_0, v_*))\leqq\left\lfloor\frac{n_0-1}{2}\right\rfloor+\left\lfloor\frac{n-n_0-1}{2}\right\rfloor. \tag{4.1}$$

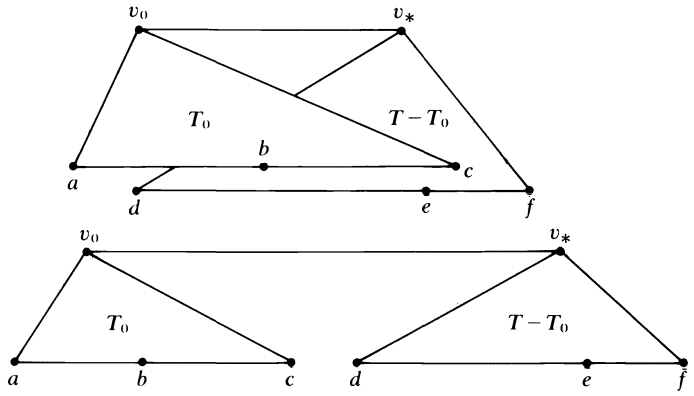


FIG. 5(a). π *(above)*; $L_1(\pi, T_0)$.

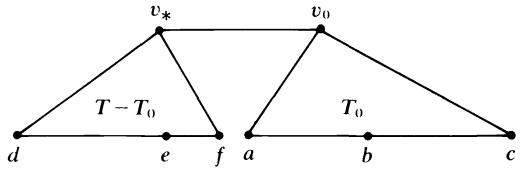


FIG. 5(b). $R_1(\pi, T_0)$.

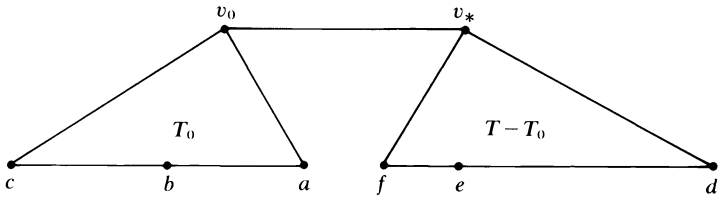


FIG. 5(c). $L_2(\pi, T_0)$.

THEOREM 4.1. *Let $v_* \in T$, let T_0 be a subtree of T mod. v_* and let π be an arr. of T.*
a) *If $\pi^{-1}(1) \in T_0$ and $\pi^{-1}(n) \notin T_0$ then $C[L_1(\pi, T_0), T] \leqq C[\pi, T]$.*
b) *If $\pi^{-1}(1) \notin T_0$ and $\pi^{-1}(n) \in T_0$ then $C[R_1(\pi, T_0), T] \leqq C[\pi, T]$.*
c) *If $\pi^{-1}(1), \pi^{-1}(n) \in T_0$ and $n_0 \leqq \lceil n+2/2 \rceil$ then*

1. $C[L_2(\pi, T_0), T] \leqq C[\pi, T]$,
2. $C[R_2(\pi, T_0), T] \leqq C[\pi, T]$.

Proof. We prove only a) and c–1) since b) and c–2) are symmetric to a) and c–1) respectively.

Proof of a). Let v_0 be the root of T_0 mod. v_*. The definition of $L_1(\pi, T_0)$ implies that:

$$l_{L_1(\pi,T_0)}(e) \leqq l_\pi(e) \quad \forall e \neq (v_0, v_*). \tag{4.2}$$

Let

$$A = \{v | v \in T_0 \text{ and } \pi(v) > \pi(v_*)\},$$

$$B = \{v | v \in T - T_0 \text{ and } \pi(v) < \pi(v_0)\}.$$

$A \cup B$ are all the vertices which are between v_0 and v_* in the arr. $L_1(\pi, T_0)$

and were not there in π. Hence we have:

$$l_{L_1(\pi,T_0)}((v_0, v_*))-l_\pi((v_0, v_*))\leqq|A|+|B|. \tag{4.3}$$

If $v\in A$ then $\pi(v_*)<\pi(v)<n$. Since v_* and $\pi^{-1}(n)\in T-T_0$, v contributes one unit to the length of, at least, one edge of $T-T_0$ in the arr. π. ("v contributes one unit to the length of an edge (v_i, v_j)" means that $\pi(v_i)<\pi(v)<\pi(v_j)$ or $\pi(v_j)<\pi(v)<\pi(v_i)$.) But $v\in T_0$ and therefore contributes nothing to the length of any edge of $T-T_0$ in the arr. $L_1(\pi, T_0)$. A symmetric argument shows that if $v\in B$, it contributes one unit length to, at least, one edge of T_0, in π—and does not do so to any edge of T_0 in $L_1(\pi, T_0)$. Thus, we "gain" at least $|A|+|B|$ length units by transforming π into $L_1(\pi, T_0)$. The proof follows now immediately from (4.2) and (4.3).

Proof of c–1). As in part a) we have:

$$l_{L_2(\pi,T_0)}(e)\leqq l_\pi(e)\quad \forall e\neq(v_0, v_*). \tag{4.4}$$

From (4.1) we have:

$$l_{L_2(\pi,T_0)}((v_0, v_*))-l_\pi((v_0, v_*))\leqq\left\lceil\frac{n_0-1}{2}\right\rceil+\left\lceil\frac{n-n_0-1}{2}\right\rceil. \tag{4.5}$$

If $v\in T-T_0$ then $1<\pi(v)<n$. Since $\pi^{-1}(1)$ and $\pi^{-1}(n)\in T_0$, v contributes one unit length to at least one edge of T_0 in π. Since T_0 and $T-T_0$ are separated in $L_2(\pi, T_0)$ there is no contribution of any vertex of $T-T_0$ to the length of any edge of T_0 in $L_2(\pi, T_0)$. Thus

$$C[L_2(\pi, T_0), T]\leqq C[\pi, T]+\left\lceil\frac{n_0-1}{2}\right\rceil+\left\lceil\frac{n-n_0-1}{2}\right\rceil-(n-n_0).$$

Since

$$n_0\leqq\left\lceil\frac{n+2}{2}\right\rceil,\qquad n-n_0\geqq\left\lceil\frac{n-2}{2}\right\rceil\geqq\left\lceil\frac{n_0-1}{2}\right\rceil+\left\lceil\frac{n-n_0-1}{2}\right\rceil.$$

This completes the proof. Q.E.D.

THEOREM 4.2. *If π is an arr. of T such that $\pi^{-1}(n)=v_*(\pi^{-1}(1)=v_*)$, then there exists an arr. π' of T of type $(T_{i_0},\cdots, T_{i_k}|v_*)$ $((v_*|T_{i_k},\cdots, T_{i_0}))$ such that $C[\pi', T]\leqq C[\pi, T]$.*

Proof. The proof is by induction on k, the number of subtrees of T mod. v_*.

The case $k=1$ is trivial.

Assume that $k>1$ and $\pi^{-1}(1)\in T_{i_0}$. Since $\pi^{-1}(n)\notin T_{i_0}$ we have (Theorem 4.1.a).

$$C[L_1(\pi, T_{i_0}), T]\leqq C[\pi, T]. \tag{4.6}$$

Let $\bar{T}_{i_0}=T-T_{i_0}$, then

$$C[L_1(\pi, T_{i_0}), T]=C[\pi, \vec{T}_{i_0}(v_{i_0})]+n-n_{i_0}+C[\pi, \bar{T}_{i_0}]. \tag{4.7}$$

$\bar{T}_{i_0}$ has k subtrees mod. v_* which is its right-most vertex in the arr. $\pi/\bar{T}_{i_0}$. We can apply the inductive hypothesis to yield an arr. π'_{i_0} of $\bar{T}_{i_0}$ of type $(T_{i_1},\cdots, T_{i_k}|v_*)$ such that

$$C[\pi'_{i_0}, \bar{T}_{i_0}]\leqq C[\pi, \bar{T}_{i_0}]. \tag{4.8}$$

Let π' be the arr. of T of type $(T_{i_0},\cdots, T_{i_k}|v_*)$ which coincides with π (and $L_1(\pi, T_{i_0})$) on T_{i_0} and with π'_{i_0} on $\bar{T}_{i_0}$. By (4.7), (4.8) and (4.6) we have

$$C[\pi', T]\leqq C[L_1(\pi, T_{i_0}), T]\leqq C[\pi, T].$$ Q.E.D.

THEOREM 4.3. *Let* π_1 *and* π_2 *be two arrangements of* T *of type* $(T_{i_1}, \cdots, T_{i_r}|v_*|T_{j_1}, \cdots, T_{j_r})$, *such that*

$$C[\pi_1, \bar{T}_{i_k}(v_{i_k})] = C[\pi_2, \bar{T}_{i_k}(v_{i_k})],$$

$$C[\pi_1, \bar{T}_{j_k}(v_{j_k})] = C(\pi_2, \bar{T}_{j_k}(v_{j_k})], \qquad k = 1, \cdots, r.$$

Let $T_* = T - (T_{i_1}, \cdots, T_{i_r}, T_{j_1}, \cdots, T_{j_r})$; *then* $C[\pi_1, T] \leqq C[\pi_2, T]$ *iff* $C[\pi_1, T_*] \leqq C[\pi_2, T_*]$.

Proof. The proof follows immediately from Theorem 3.1.2.b). Q.E.D.

DEFINITION. Let π be an arr. of T. $(T_1, \cdots, T_k|v_*)/\pi$ is the arr. of type $(T_1, \cdots, T_k|v_*)$ in which the internal order of the vertices of $T_1, \cdots, T_k$ and $T - (T_1, \cdots, T_k)$ is the same as it is in π. In a similar way we define $(v_*|T_1, \cdots, T_k)/\pi$ and $(T_1, \cdots, T_p|v_*|T_{p+1}, \cdots, T_k)/\pi$.

THEOREM 4.4. *Let* $T_1, \cdots, T_k$ *be subtrees of* T *mod.* v_* *satisfying*

$$n_1 \geqq n_2 \geqq \cdots \geqq n_k.$$

a) *If* π *is an arr. of type* $(T_{i_1}, \cdots, T_{i_k}|v_*)$ $(\{i_1, \cdots, i_k\} = \{1, \cdots, k\})$, *then*

$$C[(T_1, \cdots, T_k|v_*)/\pi, T] \leqq C[\pi, T].$$

b) *If* π *is of type* $(v_*|T_{i_1}, \cdots, T_{i_k})$ *then*

$$C[(v_*|T_k, \cdots, T_1)/\pi, T] \leqq C[\pi, T].$$

c) *If* π *is of type* $(T_{i_1}, \cdots, T_{i_m}|v_*|T_{j_1}, \cdots, T_{j_m})$ *and* $\{i_1, \cdots, i_m, j_1, \cdots, j_m\} = \{1, \cdots, (= 2m)\}$, *then*

$$C[(T_1, T_3, \cdots, T_{k-1}|v_*|T_k, \cdots, T_4, T_2)/\pi, T] \leqq C[\pi, T].$$

Proof. We prove a) and c). b) is symmetric to a).

Proof of a). It would suffice to show that if $n_{i_q} \leqq n_{i_{q+1}}$ then $C[\pi', T] \leqq C[\pi, T]$, where $\pi' = (T_{i_1}, \cdots, T_{i_{q-1}}, T_{i_{q+1}}, T_{i_q}, T_{i_{q+2}}, \cdots, T_{i_k}|v_*)/\pi$. Let v_{i_q}, $v_{i_{q+1}}$ be the roots of T_{i_q}, $T_{i_{q+1}}$ mod. v_* respectively. Transforming π into π' only (v_*, v_{i_q}) and $(v_*, v_{i_{q+1}})$ change their length. $l_\pi((v_*, v_{i_q}))$ is decreased by $n_{i_{q+1}}$ and $l_\pi((v_*, v_{i_{q+1}}))$ is increased by n_{i_q}. This completes the proof of a) since $n_{i_q} \leqq n_{i_{q+1}}$.

Proof of c). The proof is by induction on m. The case $m = 1$ is trivial.

Assume that $m > 1$ and $i_p = 1$.

Case: 1: $2 = j_q$. Using the same argument as before, we can show that:

$$C[T_1, T_{i_1}, \cdots, T_{i_{p-1}}, T_{i_{p+1}}, \cdots, T_{i_m}|v_*|T_{j_m}, \cdots, T_{j_{q+1}}, T_{j_{q-1}}, \cdots, T_{j_1}, T_2)/\pi, T] \leqq C[\pi, T].$$

Let $T_* = T - (T_1, T_2)$. The inductive hypothesis implies:

$$C[(T_3, T_5, \cdots, T_{k-1}|v_*|T_k, \cdots, T_4)/\pi, T_*] \leqq C[\pi, T_*],$$

and the proof is completed by using Theorem 4.3.

Case 2: $2 = i_q$. We may assume that $p < q$. Let $T_* = T - (T_{i_1}, \cdots, T_{i_p}, T_{j_1}, \cdots, T_{j_p})$ and let $\bar{\pi}$ be obtained from π be reversing the order of the vertices of T_* (see Fig. 6). Obviously $C[\pi, T_*] = C[\bar{\pi}, T_*]$, and by Theorem 4.3 $C[\pi, T] = C[\bar{\pi}, T]$. $\bar{\pi}$ satisfies the conditions of Case 1, i.e. T_1 and T_2 are on different sides of v_*. Thus $C[(T_1, T_3, \cdots, T_{k-1}|v_*|T_k, \cdots, T_2)/\pi, T] \leqq C[\bar{\pi}, T] = C[\pi, T]$. Q.E.D.

Let T_0, T_1, T_2 be subtrees of T mod. v_* and let π be an arr. of T of type $(T_1|v_*|T_2)$. Let $T' = T - (T_1, T_2)$. We now define the arr. $L_3(\pi, T_0, T_1, T_2)$ in the following way. First, we apply $L_2(\pi, T_0)$ to T', Keeping T_1 to the left, and T_2 to the

right of T'. Let π_L be the arr. obtained by this step. $L_3(\pi, T_0, T_1, T_2)$ is defined as $L_1(\pi_L, T_0)$. Replacing every L by R we obtain the definition of $R_3(\pi, T_0, T_1, T_2)$. (See Fig. 7.)

THEOREM 4.5. *Let T_0, T_1, T_2 be subtrees of T mod. v_*, let π be an arr. of type $(T_1|v_*|T_2)$ such that $\pi^{-1}(n_1+1), \pi^{-1}(n-n_2) \in T_0$, and let $T_* = T-(T_0, T_1, T_2)$.*

a) *If*

$$n_1 \leqq \left\lfloor \frac{n_0+2}{2} \right\rfloor + \left\lfloor \frac{n_*+2}{2} \right\rfloor$$

then $C[L_3(\pi, T_0, T_1, T_2), T] \leqq C[\pi, T]$.

b) *If*

$$n_2 \leqq \left\lfloor \frac{n_0+2}{2} \right\rfloor + \left\lfloor \frac{n_*+2}{2} \right\rfloor$$

then $C[R_3(\pi, T_0, T_1, T_2), T] \leqq C[\pi, T]$.

Proof. We prove only a) since b) is symmetric. Let v_0 be the root of T_0 mod. v_*. Transforming π into $L_3(\pi, T_0, T_1, T_2)$, only (v_0, v_*) may become longer. From the definition of $L_3(\pi, T_0, T_1, T_2)$ we have:

$$l_{L_3(\pi, T_0, T_1, T_2)}((v_0, v_*)) - l_\pi((v_0, v_*)) \leqq \left\lfloor \frac{n_0-1}{2} \right\rfloor + \left\lfloor \frac{n_*-1}{2} \right\rfloor + n_1.$$

Since $\pi^{-1}(1), \pi^{-1}(n) \notin T_0$, every vertex of T_0 contributes a unit to the length of at least one edge of $T-T_0$. Such contributions do not occur in $L_3(\pi, T_0, T_1, T_2)$. Thus we "gain" at least n_0 length units in this way. Moreover, $\pi^{-1}(n_1+1), \pi^{-1}(n-n_2) \in T_0$ and therefore each vertex of T_* contributes one unit to the length of at least one edge

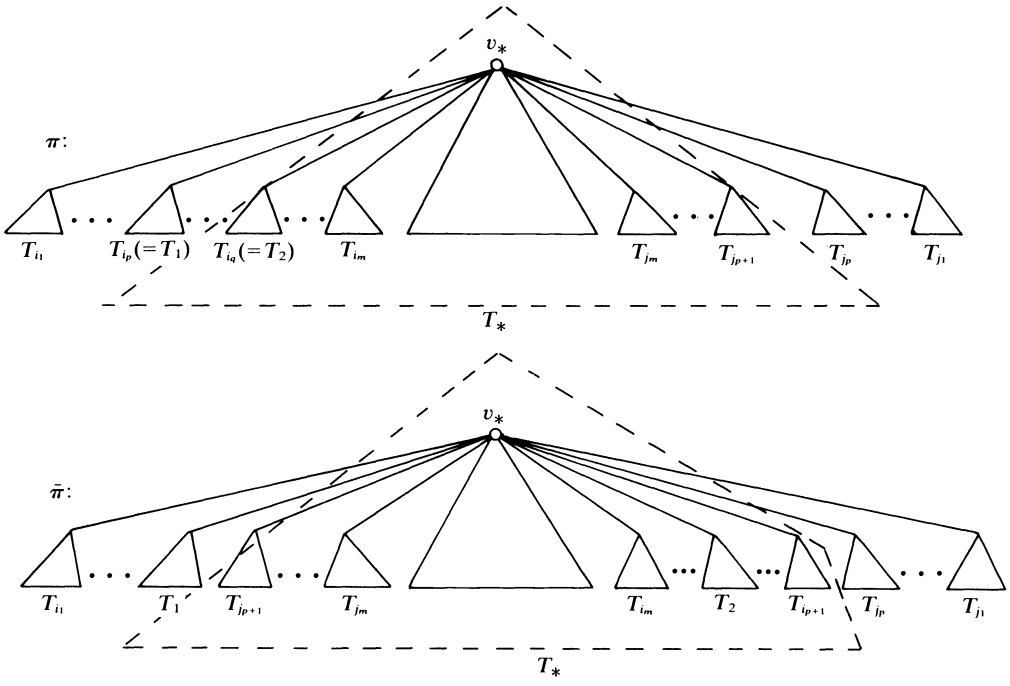


FIG. 6. π (*above*); $\bar{\pi}$.

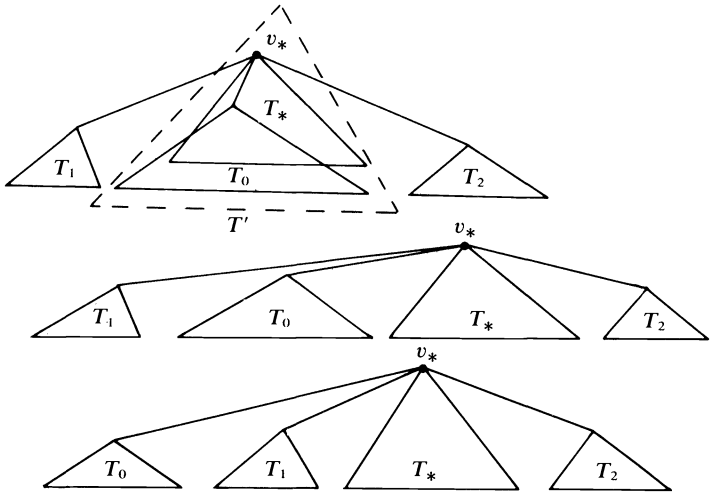


FIG. 7. π (*above*); π_L (*middle*); $L_3(\pi, T_0, T_1, T_2)$.

of T_0 in π. No such contribution occurs in $L_3(\pi, T_0, T_1, T_2)$. Thus we have:

$$C[\pi, T] - C[L_3(\pi, T_0, T_1, T_2), T]$$

$$\geqq n_0 + n_* - \left\lfloor \frac{n_0 - 1}{2} \right\rfloor - \left\lfloor \frac{n_* - 1}{2} \right\rfloor - n_1 = \left\lfloor \frac{n_0 + 2}{2} \right\rfloor + \left\lfloor \frac{n_* + 2}{2} \right\rfloor - n_1 \geqq 0. \qquad \text{Q.E.D.}$$

5. Proof of Theorem 3.1.1. The proof of Theorem 3.1.1 is the heart of this paper. In fact we prove the theorem only for $\alpha = 0$. The proof for the case $\alpha = 1$ is very similar and does not contain any argument which is not used in the proof below.

Given a free tree T, we choose $v_* \in T$ which satisfies (2.4). $T_0, T_1, \cdots, T_k$ are all the subtrees of T mod. v_*. $v_0, \cdots, v_k$ are their roots mod. v_* respectively, and through the whole section we assume:

$$\left\lfloor \frac{n}{2} \right\rfloor \geqq n_0 \geqq n_1 \geqq \cdots \geqq n_k. \tag{5.1}$$

The proof technique is as follows: We start with an arbitrary arr. π of T. We assume that:

(5.2) There is no arr. π' of T of type $(T_0|v_*)$ (and therefore no arr. of type $(v_*|T_0)$) such that $C[\pi', T] \leqq C[\pi, T]$.

We then use π in order to define a nonnegative integer p. Using (5.1) and (5.2) we show that $p \geqq 1$ (Lemma 5.3) and that there exists an arr. π' of type $(T_1, T_3, \cdots, T_{2p-1}|v_*|T_{2p}, \cdots, T_4, T_2)$ such that $C[\pi', T] \leqq C[\pi, T]$. Finally we show (Lemma 5.7) that $p = p_0$. The reader can easily verify that when everything is shown, the proof is completed.

Let us use π to define a new order on $T_1, \cdots, T_k$. This is the order in which they appear in the arr. π. The "left-most" subtree is first, the "right-most" is second, the second from left is third and so on, until we reach T_0.

More formally, let l_1 be defined by the requirement: The left-most vertex of T (mod. π) belongs to T_{l_1}.

$$\pi_{l_1} \triangleq L_2(\pi, T_{l_1}).$$

r_1 is defined by the requirement: The right-most vertex of T (mod. π_{l_1}) belongs to T_{r_1}.

$$\pi_{r_1} \triangleq L_1(\pi_{l_1}, T_{r_1}), \qquad \bar{T}_1 \triangleq T-(T_{l_1}, T_{r_1}).$$

Assume that l_j, r_j, π_{l_j}, π_{r_j} and $\bar{T}_j$ have already been defined for $j=1, \cdots, i-1$. Then l_i satisfies: The left-most vertex of $\bar{T}_{i-1}$ (mod. $\pi_{r_{i-1}}$) belongs to T_{l_i}. π_{l_i} is the arr. of type $(T_{l_1}, \cdots, T_{l_i}|v_*|T_{r_{i-1}}, \cdots, T_{r_1})$ which is obtained from $\pi_{r_{i-1}}$ by applying $L_2(\pi_{r_{i-1}}, T_{l_i})$ on $\bar{T}_{i-1}$, keeping the rest unchanged. r_i satisfies: The right-most vertex of $\bar{T}_{i-1}$ mod. π_{l_i} belongs to T_{r_i}. π_{r_i} is the arr. of type $(T_{l_1}, \cdots, T_{l_i}|v_*|T_{r_i}, \cdots, T_{r_1})$ which is obtained from π_{l_i} by applying $R_1(\pi_{l_i}, T_{r_i})$ on $\bar{T}_{i-1}$ keeping the rest unchanged. $\bar{T}_i \triangleq \bar{T}_{i-1}-(T_{l_i}, T_{r_i})$. p is defined by the requirement: $l_i, r_i \neq 0$ for $i=1, \cdots, p$; $l_{p+1}=0$ or $r_{p+1}=0$, i.e. the process is stopped when we reach T_0.

In order to show that the definition of l_i and r_i is legitimate (for $i=1, \cdots, p$) we must show that v_* is neither the left-most nor the right-most of any of the $\bar{T}_i$'s for $i=0, \cdots, p$. ($\bar{T}_0 \triangleq T$.) This is done in Lemma 5.2.

In Fig. 8 we have a tree T with six subtrees mod. v_* $(T_0, \cdots, T_5)$.

An arr. π of T is shown for which $l_1=1$, $r_1=5$, $l_2=4$, $r_2=2$ ($p=2$). The arrangements π_{l_1}, π_{r_1}, π_{l_2} and π_{r_2} are also shown.

LEMMA 5.1. *$C[\pi_{l_j}, T] \leqq C[\pi, T]$, $C[\pi_{r_j}, T] \leqq C[\pi, T]$ for $j=1, \cdots, p$.*

Proof. We shall prove a much stronger statement, namely: $C[\pi_{r_j}, T] \leqq C[\pi_{l_j}, T] \leqq C[\pi_{r_{j-1}}, T]$, $j=1, \cdots, p$ (where $\pi_{r_0} \triangleq \pi$). Using Theorem 4.3 it is enough to prove that:

$$C[\pi_{r_j}, \bar{T}_{j-1}] \leqq C[\pi_{l_j}, \bar{T}_{j-1}], \tag{5.3}$$

$$C[\pi_{l_j}, \bar{T}_{j-1}] \leqq C[\pi_{r_{j-1}}, \bar{T}_{j-1}]. \tag{5.4}$$

The right-most vertex of $\bar{T}_{j-1}$ mod. π_{l_j} belongs to T_{r_j} while the left-most doesn't (it belongs to T_{l_j}). Thus (5.3) is implied directly from Theorem 4.1.b).

The left-most vertex of $\bar{T}_{j-1}$ mod. $\pi_{r_{j-1}}$ belongs to T_{l_j}. If the right-most doesn't, we can use Theorem 4.1.a) to prove (5.4). If the right-most vertex of $\bar{T}_{j-1}$ also belongs to T_{l_j} we can use Theorem 4.1.c). We just have to show that $n_{l_j} \leqq \frac{1}{2}\bar{n}_{j-1}$. This is true since T_0 and T_{l_j} are subtrees of $\bar{T}_{j-1}$ and $n_0 \geqq n_{l_j}$. Q.E.D.

In order to establish the validity of the definition of l_j, r_j for $j=1, \cdots, p$ we have to prove the following lemma.

LEMMA 5.2. a) *v_* is not the left-most vertex of $\bar{T}_j$ mod. π_{r_j} for $j=0, \cdots, p$ ($\bar{T}_0=T$, $\pi_{r_0}=\pi$).*

b) *v_* is not the right-most vertex of $\bar{T}_j$ mod. $\pi_{l_{j+1}}$ for $j=0, \cdots, p$.*

Proof. We shall prove only a). The proof of b) is almost the same.

If v_* is the left-most vertex of T mod. π we can use Theorems 4.2 and 4.4 to obtain an arr. π' of T of type $(v_*|T_0)$ such that $C[\pi', T] \leqq C[\pi, T]$—contradicting (5.2). Thus, the statement is true for $j=0$.

Assume that $j>0$ and that the lemma holds for $0, \cdots, j-1$. Thus π'_{l_1}, $\pi_{r_1}, \cdots, \pi_{l_j}, \pi_{r_j}$ are well-defined and by Lemma 5.1 $C[\pi_{r_j}, T] \leqq C[\pi, T]$.

Assume that v_* is the left-most vertex of $\bar{T}_j$ mod. π_{r_j}. Applying Theorems 4.2 and 4.4 on $\bar{T}_j$, it can be shown that there exists an arr. π'' of $\bar{T}_j$ of type $(v_*|T_0)$ such that $C[\pi'', \bar{T}_j] \leqq C[\pi_{r_j}, \bar{T}_j]$. π'' can be extended to an arr. of T by identifying it with π_{r_j} on $T_{l_1}, T_{r_1}, \cdots, T_{l_j}, T_{r_j}$. The extended π'' is of type $(T_{l_1}, \cdots, T_{l_j}|v_*|T_0, T_{r_j}, \cdots, T_{r_1})$ and by Theorem 4.3

$$C[\pi'', T] \leqq C[\pi_{r_j}, T] \qquad (\leqq C[\pi, T]).$$

Theorem 4.4 can be used now to yield an arr. π' of type $(v_*|T_0)$ such that $C[\pi', T] \leqq C[\pi'', T]$ contradicting (5.2). Q.E.D.

LEMMA 5.3. $p \geqq 1$.

Proof. $p = 0$ means that either $\pi^{-1}(1) \in T_0$ or $\pi^{-1}(n) \in T_0$ or both.

By Theorem 4.1 all these three cases yield a contradiction to (5.2). When both $\pi^{-1}(1)$ and $\pi^{-1}(n)$ belong to T_0, we also use (5.1). This is the only place in which we need the "centrality" of v_*. Q.E.D.

In the following lemmas we shall show that π_{r_p} is an arr. of type B or, at least, can be easily transformed into an arr. of type B without increasing its cost.

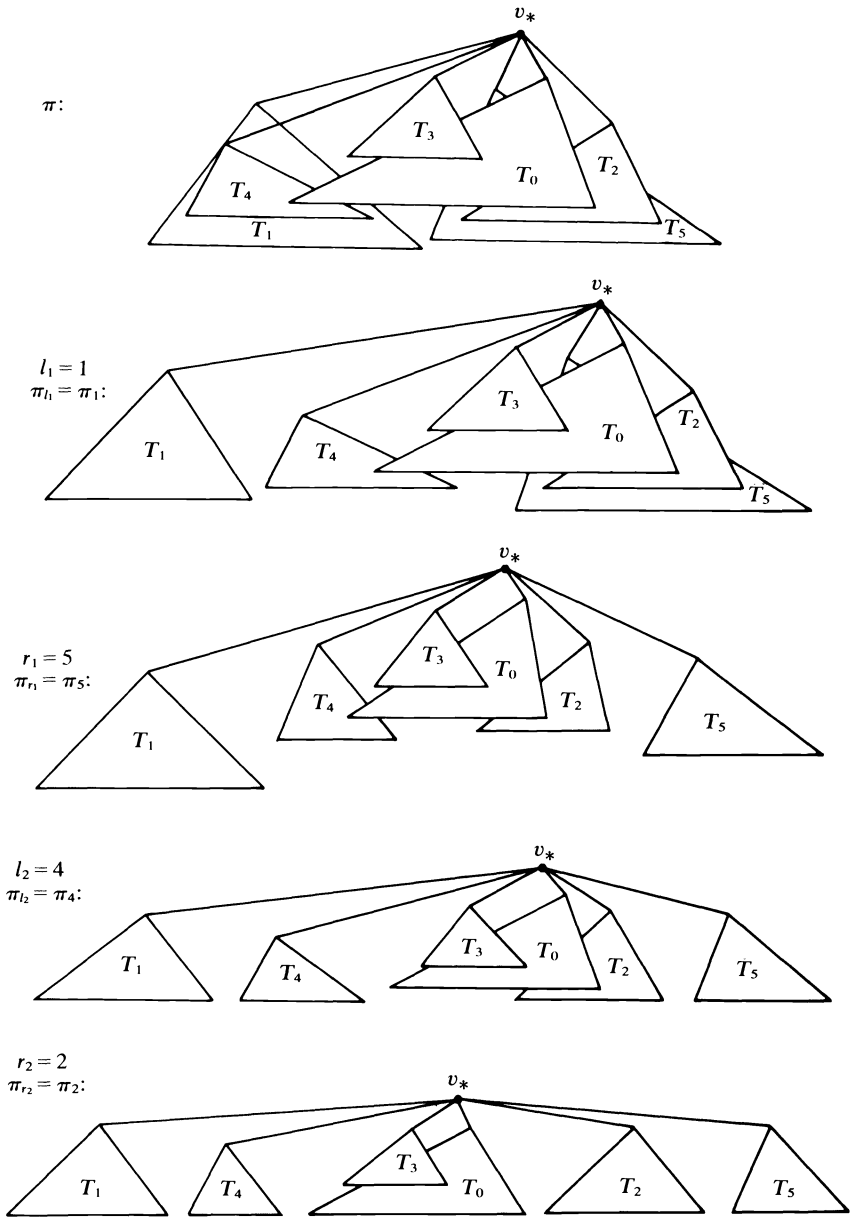


FIG. 8

LEMMA 5.4. *We may assume that*:

$$n_{l_1} \geqq n_{r_1} \geqq \cdots \geqq n_{l_p} \geqq n_{r_p} \quad \textit{or equivalently} \tag{5.5}$$

$$l_1 < r_1 < \cdots < l_p < r_p. \tag{5.6}$$

Proof. If (5.5) does not hold we can transform π_{r_p} into a new arr. π'_{r_p} of type $(T_{l'_1}, \cdots, T_{l'_p} | v_* | T_{r'_p}, \cdots, T_{r'_1})$ such that: $C[\pi'_{r_p}, T] \leqq C[\pi_{r_p}, T]$, $\{l_1, \cdots, r_p\} = \{l'_1, \cdots, r'_p\}$ and $n_{l'_1} \geqq n_{r'_1} \geqq \cdots \geqq n_{l'_p} \geqq n_{r'_p}$. (See Theorem 4.4.) We then use π'_{r_p} as our new π_{r_p} for the discussion below. Q.E.D.

LEMMA 5.5. *The right-most and the left-most vertices of* $\bar{T}_p$ *mod.* π_{r_p} *belong to* T_0.

Proof. By the definition of p one of the two cases holds:

1. The left-most vertex of $\bar{T}_p$ mod. π_{r_p} belongs to T_0.
2. The right-most vertex of $\bar{T}_p$ mod. $\pi_{l_{p+1}}$ belongs to T_0.

Assume that case 1 holds. If the right-most vertex of $\bar{T}_p$ mod. π_{rp} belongs to T_0—the statement of the lemma is true. If not, a contradiction to (5.2) can be obtained by using Theorems 4.1.a), 4.3 and 4.4 in a way which is very similar to that which was used in the proof of Lemma 5.2.

If only case 2 holds then $C[\pi_{l_{p+1}}, T] \leqq C[\pi_{r_p}, T]$.The right-most vertex of $\bar{T}_p$ mod. $\pi_{l_{p+1}}$ belongs to T_0 while the left-most doesn't (it belongs to $T_{l_{p+1}}$). Thus, the same argument as before leads to a contradiction to (5.2). Q.E.D.

LEMMA 5.6. a) $\{l_1, r_1, \cdots, l_p, r_p\} = \{1, 2, \cdots, 2p\}$.

b) $n_{2p+1} < n_{2p}$.

Proof. a). It is enough to show that $\{1, \cdots, 2p\} \subseteq \{l_1, \cdots, r_p\}$ since both sets have the same number of elements.

Assume to the contrary that $j \in \{1, \cdots, 2p\}$ and $j \notin \{l_1, \cdots, r_p\}$. Let $T_* = \bar{T}_p - T_0$. (Fig. 9 shows $\bar{T}_{p-1}$ under the arr. π_{r_p}; note that by Lemma 5.5, the left-most and right-most vertices of $\bar{T}_p$ belong to T_0.)

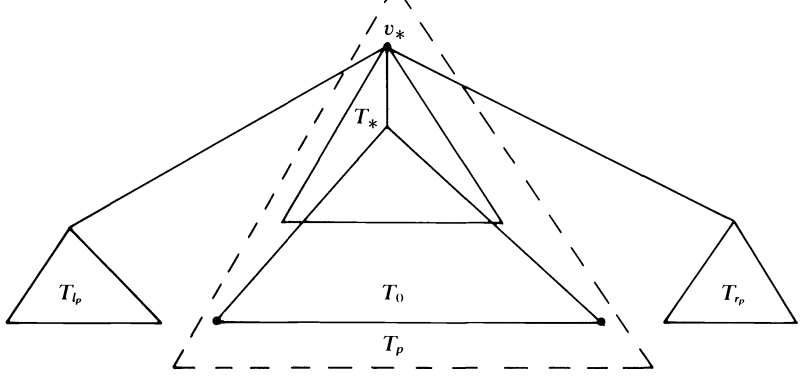


FIG. 9. *The tree* $\bar{T}_{p-1}$ *under the arr.* π_{r_p}.

Since $j \notin \{l_1, \cdots, r_p\}$, T_j is a subtree of T_* and therefore $n_j \leqq n_*$. But $r_p > 2p$ (Lemma 5.4) and $j \leqq 2p$ implies that $j < r_p$ and $n_{r_p} \leqq n_j$.

Since $n_{r_p} \leqq n_0$ we have:

$$n_{r_p} < \left\lfloor \frac{n_0 + 2}{2} \right\rfloor + \left\lfloor \frac{n_* + 2}{2} \right\rfloor. \tag{5.7}$$

Lemma 5.5 together with (5.7) allow us to apply Theorem 4.5 to yield:

$$C[R_3(\pi_{r_p}, T_0, T_{l_p}, T_{r_p}), \bar{T}_{p-1}] \leqq C[\pi_{r_p}, \bar{T}_{p-1}].$$

Let π'' be the arr. of T obtained from π_{r_p} by applying $R_3(\pi_{r_p}, T_0, T_{l_p}, T_{r_p})$ on $\bar{T}_{p-1}$. (See Fig. 10.)

By Theorem 4.3, $C[\pi'', T] \leqq C[\pi_{r_p}, T]$. Moreover by Theorem 4.4, π'' can be transformed into a $(v_*|T_0)$ arr. π' without increasing the cost—a contradiction to (5.2).

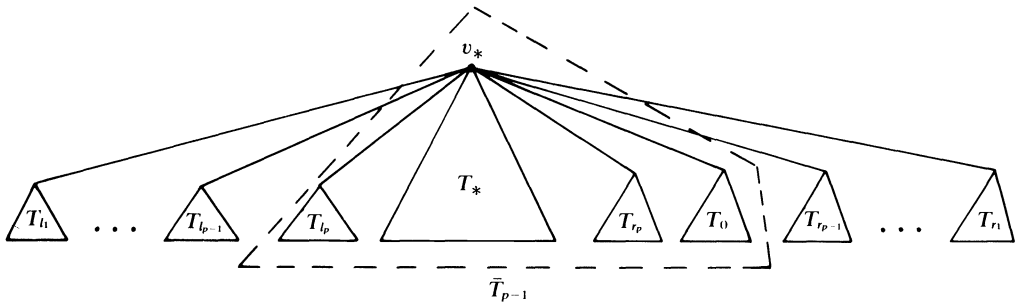


FIG. 10. π''.

b) By part a), $n_{r_p} = n_{2p}$. $2p+1 \notin \{l_1, \cdots, r_p\}$ and therefore T_{2p+1} is a subtree of T_* and $n_{2p+1} \leqq n_*$. If $n_{2p} = n_{2p+1}$, we have

$$n_{r_p} < \left\lfloor \frac{n_o+2}{2} \right\rfloor + \left\lfloor \frac{n_*+2}{2} \right\rfloor$$

which is impossible according to part a). Q.E.D.

COROLLARY 5.6.1. *$l_i = 2i-1$ and $r_i = 2i$ for $i = 1, \cdots, 2p$.* (See Lemma 5.4.)

COROLLARY 5.6.2. *The set of the first $2p$ subtrees—$\{T_1, \cdots, T_{2p}\}$ is uniquely determined by their numbers of vertices* (Lemma 5.6.b).)

The value of p as defined above, seems to depend on the particular choice of an initial arr. π. The following lemma shows that it is not so and, in fact, $p = p_0$. Recall that p_0 was defined as the greatest integer satisfying:

$$n_i > \left\lfloor \frac{n_0+2}{2} \right\rfloor + \left\lfloor \frac{n_*+2}{2} \right\rfloor \quad \text{for } i = 1, \cdots, 2p_0 \tag{5.8}$$

where $n_* = n - \sum_{i=0}^{2p_0} n_i$.

LEMMA 5.7. $p = p_0$.

Proof. It is easy to see that p satisfies (5.8)—otherwise (5.2) is violated as shown in the proof of Lemma 5.6. If $q > p$ also satisfies (5.8) then

$$n_{2q-1} > \left\lfloor \frac{n_0+2}{2} \right\rfloor \quad \text{and} \quad n_{2q} > \left\lfloor \frac{n_0+2}{2} \right\rfloor$$

so that $n_{2q-1} + n_{2q} > n_0$.

Since T_0, T_{2q-1} and T_{2q} are disjoint subtrees of $\bar{T}_p$, we have: $n_0 < \bar{n}_p/2$.

Using Lemma 5.5 and Theorem 4.1.c) we have

$$C[L_2(\pi_{r_p}, T_0), \bar{T}_p] \leqq C[\pi_{r_p}, \bar{T}_p].$$

The proof is completed by Theorems 4.3 and 4.4 as we have done several times before. Q.E.D.

6. Complexity. Let $f_\alpha(n)$ denote the number of elementary computation operations required in order to find a minimum arr. for a tree $T(\alpha)$ with n vertices. The following theorem is implied directly by the algorithm.

THEOREM 6.1. *$f_\alpha(n)$ satisfies the following recursive inequalities*:

$$f_0(n) \leqq f_1(n_0)+f_1(n-n_0)+f_1(n_1)+\cdots+f_1(n_{2p_0}) + f_0(n_0+n_*)+c_0\cdot n,$$

$$f_1(n) \leqq f_1(n_0)+f_0(n-n_0)+f_1(n_1)+\cdots+f_1(n_{2p_1-1}) + f_0(n_0+n_*)+c_1\cdot n,$$

subject to the following constraints:

$$p_\alpha \geqq 1, \tag{6.1}$$

$$n_0 \geqq n_1 \geqq \cdots \geqq n_{2p_\alpha-\alpha} > 0, \tag{6.2}$$

$$n = n_* + n_0 + \cdots + n_{2p_\alpha-\alpha}, \tag{6.3}$$

$$n_i > \left\lfloor \frac{n_0+2}{2} \right\rfloor + \left\lfloor \frac{n_*+2}{2} \right\rfloor \quad \text{for } i = 1, 2, \cdots, 2p_\alpha-\alpha. \tag{6.4}$$

Note that $c_\alpha \cdot n$ are required in order to perform steps 1, 3, 4, 6 of the algorithm.

THEOREM 6.2. *There exists a constant c such that $f_\alpha(n) \leqq cn^{2.2}$ for all n.*

Proof. The proof is by induction on n.

Assume that there exists a constant c such that

$$f_\alpha(n') \leqq cn'^{2.2} \quad \text{for all } n' < n.$$

We have to show that $f_\alpha(n) \leqq cn^{2.2}$ too. Though the proofs for both cases $\alpha = 0$ and $\alpha = 1$ are quite similar, the case $\alpha = 1$ is a little more difficult and we shall prove only this case.

Let $s = 2.2$ and let $q = 2p_1 - 1$. Inequality (6.1) can be rewritten in the form:

$$q \geqq 1. \tag{6.5}$$

The inducive hypothesis together with Theorem 6.1 imply that there exists a constant $c (c \geqq c_1)$ such that

$$f_1(n) \leqq c[n_0^s + n_1^s + \cdots + n_q^s + (n-n_0)^s + (n_0+n_*)^s + n].$$

Let

$$F(n_0, n_1, \cdots, n_q, n_*) = (n_0+\cdots+n_q+n_*)^s - [n_0^s+\cdots+n_q^s+(n_1+\cdots+n_q+n_*)^s+(n_0+n_*)^s+n_0 + \cdots + n_q + n_*].$$

We have to show that $F(n_0, \cdots, n_q, n_*) \geqq 0$ at every point of the domain D determined by the constraints (6.2), $\cdots$, (6.5). Note that the variables $n_0, \cdots, n_q, n_*$ are independent since they are not connected by any equation.

For all $1 \leqq i \leqq q$ we have

$$\frac{\partial F(n_0, \cdots, n_q, n_*)}{\partial n_i}$$
$$= s\left[(n_0+n_1+\cdots+n_q+n_*)^{s-1} - n_i^{s-1} - (n_1+\cdots+n_*)^{s-1} - \frac{1}{s}\right]$$
$$\geqq s\left[(n_0+\cdots+n_q+n_*)^{s-1} - n_0^{s-1} - (n_1+\cdots+n_*)^{s-1} - \frac{1}{s}\right]$$
$$> 0,$$

since $(a+b)^{s-1} - a^{s-1} - b^{s-1} - 1/s > 0$ for all $a, b > 2$. F is therefore strictly increasing in each of the variables $n_1, \cdots, n_q$ at every point of D.

Since $q \geqq 1$ and $n_i > (n_0+n_*)/2$ for $i = 1, \cdots, q$, it would be enough to prove that

$$F\left(n_0, \frac{n_0+n_*}{2}, 0, \cdots, 0, n_*\right)$$
$$= (\tfrac{3}{2}(n_0+n_*))^s - \left[n_0^s + \left(\frac{n_0+n_*}{2}\right)^s + \left(\frac{n_0+3n_*}{2}\right)^s + (n_0+n_*)^s + \tfrac{3}{2}(n_0+n_*)\right]$$
$$\geqq 0.$$

Claim. $n_0^s + ((n_0+3n_*)/2)^s < (n_0+n_*)^s + ((n_0+n_*)/2)^s$.

Proof of the claim. Let

$$x_1 = n_0, \quad y_1 = \frac{n_0+3n_*}{2}, \quad x_2 = n_0+n_*, \quad y_2 = \frac{n_0+n_*}{2}.$$

We have to show that $x_1^s + y_1^s < x_2^s + y_2^s$. But $x_1 + y_1 = x_2 + y_2$. It would, therefore, be enough to show that $|x_1 - y_1| < |x_2 - y_2|$ $(= (n_0+n_*)/2)$. $x_1 - y_1 = (n_0 - 3n_*)/2 < (n_0+n_*)/2$ and

$$y_1 - x_1 = \frac{3n_* - n_0}{2} < \frac{n_0+n_*}{2}$$

since $n_0 > n_*$. (See (6.2) and (6.4).) This completes the proof of the claim.

The previous claim reduces the problem to showing that $(\frac{3}{2}(n_0+n_*))^s \geqq 2(n_0+n_*)^s + 2((n_0+n_*)/2)^s + \frac{3}{2}(n_0+n_*)$, or,

$$\left(\frac{3}{2}\right)^s \geqq 2 + 2^{1-s} + \frac{3}{2(n_0+n_*)^{s-1}}. \tag{6.6}$$

But $(\frac{3}{2})^s > 2 + 2^{1-s}$. (Note that this is not true for $s = 2$, in fact s was determined by this inequality.) For a large n_0, $3/(2(n_0+n_*)^{s-1})$ is small enough to satisfy (6.6) and the size of such an n_0 affects only the constant c. Q.E.D.

REFERENCES

[1] D. ADOLPHSON AND T. C. HU, *Optimal linear ordering*, SIAM J. of Appl. Math., 25 (1973), pp. 403–423.

[2] M. R. GAREY, D. S. JOHNSON AND L. STOCKMEYER, *Some Simplified NP-complete graph problems*, Theoret. Comput. Sci., 1 (1976), pp. 237–267.

[3] R. M. KARP, *Reducibility among combinatorial problems*, Proc. of 6th Annual ACM Symp. on Theory of Computing, 1974, pp. 47–63.

[4] S. EVAN AND Y. SHILOACH, *NP-Completeness of several arrangement problems*, technical report no. 43, Computer Science Dept., The Technion, Haifa, Israel.

[5] M. HANAN AND J. M. KURTZBERG, *A review of the placement and quadratic assignment problems*, SIAM Rev., 14 (1972), pp. 324–342.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0003 $01.00/0

A PARTIAL ANALYSIS OF RANDOM HEIGHT-BALANCED TREES*

MARK R. BROWN†

Abstract. The collection of nodes nearest to the external nodes in a random height-balanced tree is analyzed. We determine the proportion of balanced nodes in this section of the tree, and find the average number of single and double rotations which occur at the minimum height during insertions. If $\bar{B}_n$ denotes the average number of balanced nodes in a random height-balanced tree with n internal nodes, we show that $\frac{10}{21}(n+1) \leqq \bar{B}_n \leqq \frac{6}{7}(n+1)-1$ for $n \geqq 6$.

Key words. analysis of algorithms, AVL tree, height-balanced tree

1. Introduction. Height-balanced binary trees (or AVL trees) [1], [2], [3] are a method of organizing information which allows both fast accessing and fast updating. For example, height-balanced trees may be used to represent arbitrary linear lists of length n such that items can be inserted into and deleted from a list in at most $O(\log n)$ time. In fact, these operations take $\theta(\log n)$ time on the average, because even in a perfectly balanced tree the average successful or unsuccessful search requires $\Omega(\log n)$ time.[1]

A more precise analysis of the average behavior of height-balanced trees seems difficult, however. The most promising situation for study is building a height-balanced tree by random insertions; Knuth [3] gives the results of empirical tests using random numbers for trees of size between 100 and 2000, and tabulates exact results for the case of the 10th random insertion, but gives no analytical results.

In this paper we give a partial analysis of random height-balanced trees. To outline the scope of the analysis it will be useful for us to review the definition of a height-balanced tree here, and to make some new definitions. A binary tree is *height-balanced* if the height of the left subtree of every node differs by at most ± 1 from the height of its right subtree. (The *height* of a tree is the length of the longest path from the root to an external node.) An example of a height-balanced tree is given in Fig. 1; following the conventions of [3], internal nodes are circular and external nodes are square. The balance factor in each internal node is shown as $+$, $\cdot$, or $-$ according as the right subtree height is one greater than, equal to, or one smaller than

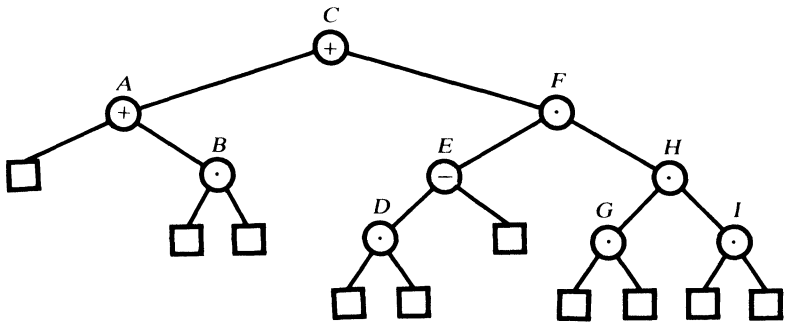


FIG. 1. *A balanced tree.*

* Received by the editors August 5, 1977, and in revised form March 17, 1978.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520.

[1] A function $g(n)$ is $\Omega(f(n))$ if there exist positive constants C and n_0 with $g(n) \geqq Cf(n)$ for all $n \geqq n_0$; it is $\theta(f(n))$ if there exist positive constants C, C', and n_0 with $Cf(n) \leqq g(n) \leqq C'f(n)$ for all $n \geqq n_0$. Hence the "θ" can be read "order exactly" and the "Ω" as "order at least"; see [4] for further discussion of the θ and Ω notations.

the left subtree height. A node whose subtrees have equal height is called *balanced*, while other nodes are *unbalanced*.

We define an internal node in a height-balanced tree to be a *fringe node* if at least one of its two offspring is an external node; the set of all fringe nodes is called the *fringe* of the tree. In the tree of Fig. 1 the fringe contains six nodes, since the only nonfringe nodes are C, F and H. The balanced fringe nodes (nodes B, D, G and I in Fig. 1) must have two external nodes as offspring; the unbalanced fringe nodes (A and E in the figure) have an external node as one offspring, and must have a balanced fringe node as the other.

Our goal is to analyze the fringe of a random height-balanced tree; in particular, we would like to determine the proportion of balanced nodes in the fringe. Besides giving us exact information about the probability of certain events during height-balanced tree insertion (such as the two simplest cases of rebalancing), this will allow us to derive fairly good bounds on the proportion of balanced nodes in the tree as a whole.

The remainder of the paper is divided into two sections. In § 2 we show how the fringe of a height-balanced tree can be viewed as consisting of two classes of subtrees, and prove lemmas which characterize how these subtrees behave under insertion. In § 3 we use these lemmas to perform the analysis. We also discuss the possibility of extending the analysis to include nodes outside of the fringe; this appears to be difficult.

2. The fringe subtrees and insertions. When the nonfringe nodes are removed from a height-balanced tree, the fringe becomes a collection of disjoint subtrees, each having one of the two forms shown in Fig. 2. An *M-subtree* is rooted by a fringe node which is unbalanced, while an *N-subtree* is simply a balanced fringe node which is not the offspring of a fringe node. (The *shapes* of the letters "M" and "N" are intended to suggest, however imperfectly, the shapes of the corresponding subtrees.) Figure 3 shows how the fringe of the height-balanced tree in Fig. 1 decomposes into these subtrees.

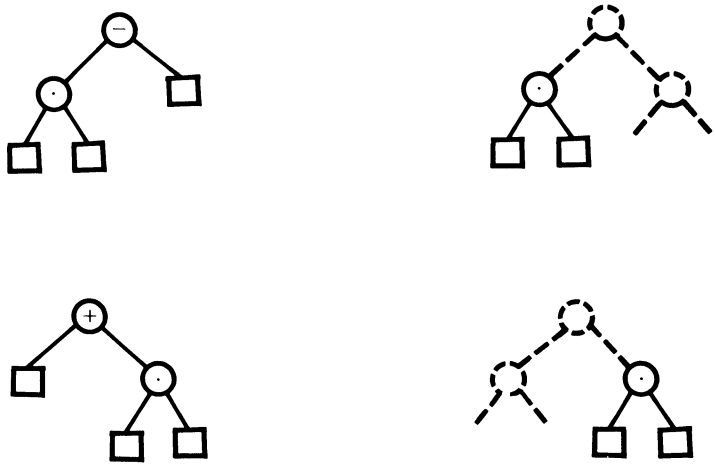


FIG. 2. *The two types of subtrees containing fringe nodes.*

Our analysis is performed in terms of M-subtrees and N-subtrees, rather than directly in terms of balanced and unbalanced fringe nodes; it is clearly possible to

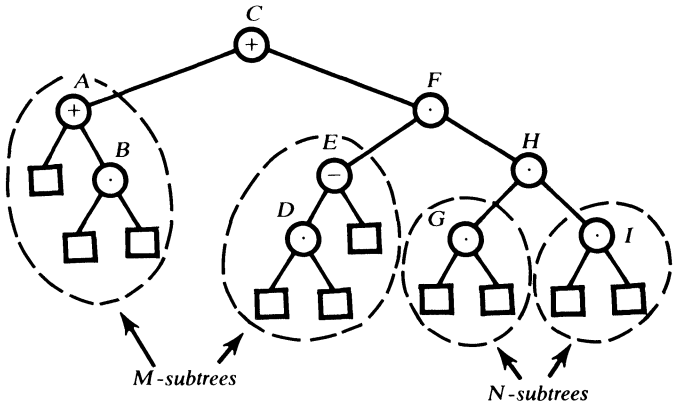
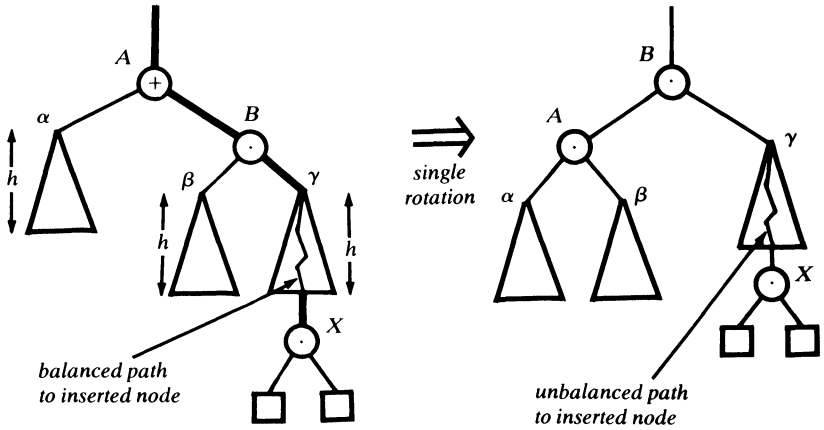


FIG. 3. *Grouping the fringe nodes into subtrees.*

translate between these two views. There is also an obvious dependency between the number of M-subtrees, the number of N-subtrees, and the total number of nodes in a balanced tree:



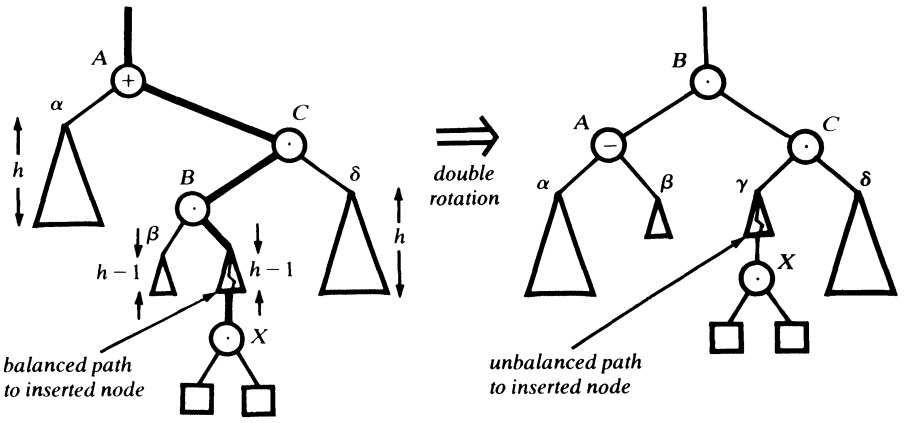


FIG. 4. *Rebalancing transformations after inserting node X. (Mirror images are possible, and the roles of β and γ may be reversed in a double rotation. When h = 0 in a double rotation, the inserted node is B.)*

LEMMA 1. *Let T be an n node height-balanced tree with M M-subtrees and N N-subtrees, where $n > 0$. Then*

$$3M + 2N = n + 1.$$

Proof. Each M-subtree has three external nodes, and each N-subtree has two; hence the left hand side counts the number of external nodes in T. But an n node height-balanced tree has $n+1$ external nodes. □

It is now easy to consider the effect which an insertion may have on the fringe; we begin with a review of the height-balanced tree insertion algorithm. The first step of inserting a new element into a height-balanced tree is to search the tree for this element, using ordinary binary tree search. The search terminates at an external node of the tree, and this external node is replaced by an internal node containing the new element. Then we climb back up the search path from the inserted node, changing the balance factors of balanced nodes from · to + (if the inserted node lies in this node's right subtree, which has grown in height) or − as appropriate, until we encounter a node which is unbalanced. (If no unbalanced node lies on the search path, then the entire tree has grown in height and the algorithm terminates.) This node now either becomes balanced or becomes more unbalanced; in the latter case, the subtree rooted at the unbalanced node is transformed by a single or double notation to restore balance. These transformations are shown in Fig. 4.

From this brief description of the insertion process it is clear that an insertion into a height-balanced tree may change the number of M- and N-subtrees in the fringe. The following two lemmas show that these changes take a very simple form.

LEMMA 2. *An insertion which falls into an external node of an M-subtree reduces the number of* M*-subtrees by one and increases the number of N-subtrees by two.*

Proof. Figure 5 shows what happens when an insertion falls into any of the three external nodes of an M-subtree. (Insertions into the mirror-image tree give mirror-image results.) In each case the M-subtree is transformed into two N-subtrees joined by a nonfringe node; there are no changes higher in the tree because the root of an M-subtree is unbalanced. □

LEMMA 3. *An insertion which falls into an external node of an N-subtree reduces the number of* N*-subtrees by one and increases the number of M-subtrees by one.*

Proof. Figure 6 shows what happens when an insertion falls into either of the two external nodes of an N-subtree. In both cases, the N-subtree is transformed into an M-subtree, but rebalancing may take place higher in the tree since the root of an N-subtree is balanced. It remains to determine what effect this rebalancing can have on the fringe. Rebalancing has no effect on nodes which lie outside of the rebalanced subtree (rooted at node A in Fig. 4), and if the fringe of the rebalanced subtree is entirely contained in subtrees which are moved without change by the rotation (subtrees α, β, γ and δ in Fig. 4), then we can see that the rotation has no effect on the fringe. We can also see (by inspecting Fig. 4) that the only case in which the fringe of the rebalanced subtree is not totally contained in these subtrees and an insertion lands in an N-subtree is the case $h = 1$ of a double rotation. Fortunately there is only one height-balanced subtree (and its mirror-image) in which this case occurs; the two insertions which cause double rotations with $h = 1$ are shown in Fig. 7. In both of these situations, the net effect on the whole fringe is to eliminate one N-subtree and introduce one M-subtree. Thus, although the rebalancing affects the fringe in these cases, it does not change the number of M- and N-subtrees. □

3. Analysis of the fringe. An insertion into a height-balanced tree is said to be a *random insertion* if it is equally likely to fall into each of the external nodes of the tree.

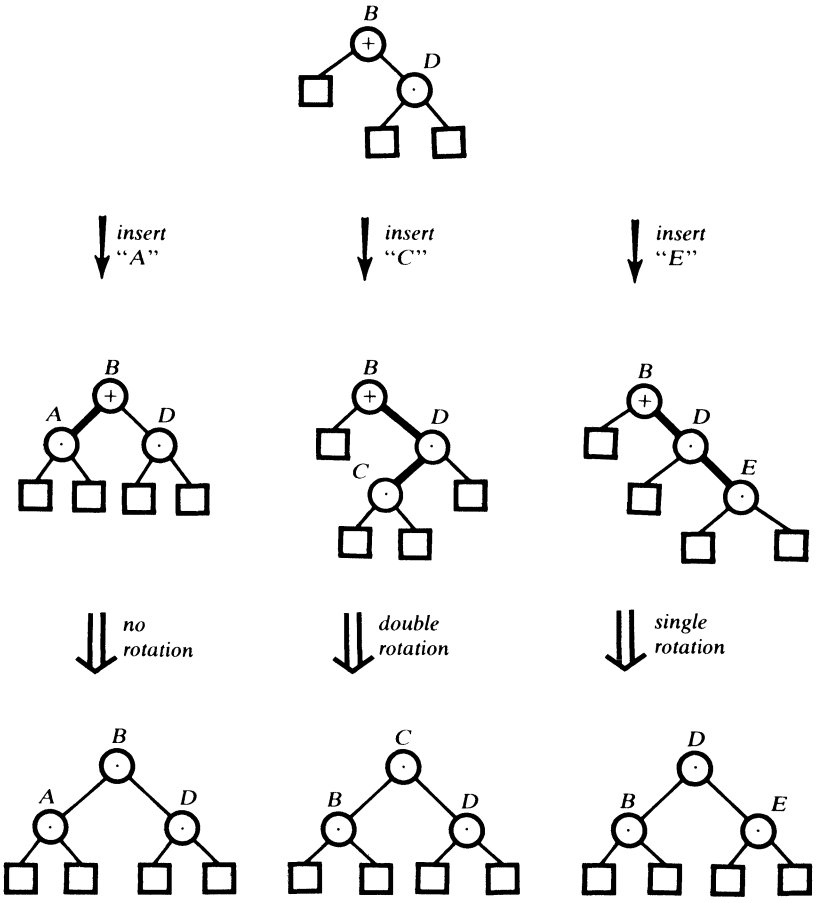


FIG. 5. *Insertion into an M subtree.*

A *random height-balanced tree* of size n is a height-balanced tree constructed by making n successive random insertions into an initially empty height-balanced tree.

Given the lemmas of the previous section, it is relatively easy to determine the average number of M- and N-subtrees in the fringe of a random height-balanced tree of size n. In fact, the required argument is given in Yao's analysis of the lowest level of random 2–3 trees [5]. For completeness we shall restate the argument here.

Let $P_n(M, N)$ denote the probability that a random height-balanced tree of size n contains M M-subtrees and N N-subtrees in the fringe. If we define

$$\bar{N}_n = \sum_{M,N} N \cdot P_n(M, N)$$

then clearly $\bar{N}_n$ is just the average number of N-subtrees in the fringe of a random height-balanced tree with n nodes; the quantity $\bar{M}_n$ is defined analogously.

THEOREM 1. $\bar{N}_n = \frac{2}{7}(n+1)$ *and* $\bar{M}_n = \frac{1}{7}(n+1)$, *for* $n \geqq 6$.

Proof. Let T be an n node height-balanced tree with M M-subtrees and N N-subtrees in the fringe, for some $n > 0$. Then by Lemmas 2 and 3, the next insertion into T changes the number of N-subtrees to $N-1$ or $N+2$, depending on whether this insertion falls into an N-subtree or an M-subtree. On a random insertion into T

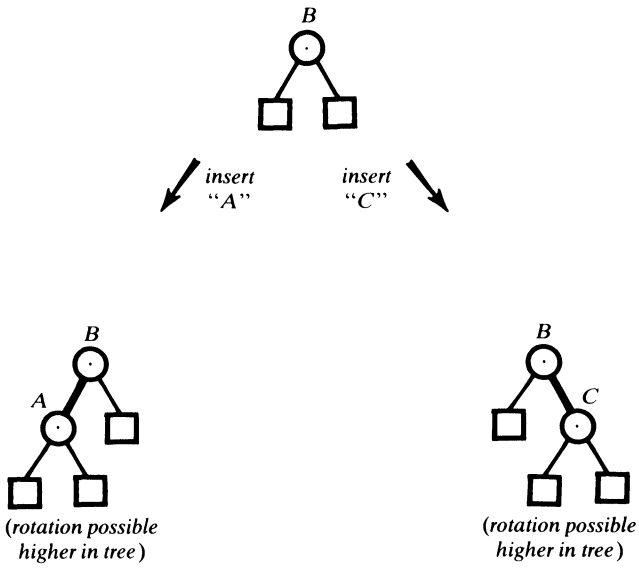


FIG. 6. *Insertion into an N subtree.*

these events happen with probabilities $2N/(n+1)$ and $3M/(n+1) = (1-(2N)/(n+1))$ respectively, so

$$\begin{aligned}\bar{N}_{n+1} &= \sum_{M,N} P_n(M,N)\left(\frac{2N}{n+1}\cdot(N-1)+\left(1-\frac{2N}{n+1}\right)\cdot(N+2)\right)\\ &= \sum_{M,N} P_n(M,N)\left(N-\frac{6N}{n+1}+2\right)\\ &= \left(1-\frac{6}{n+1}\right)\bar{N}_n+2.\end{aligned}$$

Clearly $\bar{N}_1 = 1$, and by using the recurrence above to compute a few values we immediately guess that $\bar{N}_n = \frac{2}{7}(n+1)$ for $n \geqq 6$; this guess is easy to verify by induction. Then the expression for $\bar{M}_n$ follows using Lemma 1. □

This theorem allows us to confirm the accuracy of some of the empirical results on random height-balanced trees given by Knuth. In [3, § 6.2.3 Table 1], the probability that a random insertion into a large random height-balanced tree falls into an M-subtree and causes either no rebalancing, a single rotation, or a double rotation is listed as approximately 0.144 in each case; the corresponding exact probability for the 10th insertion is $\frac{1}{7} = 0.1429571\ldots$ [3, § 6.2.3 Table 2]. The following corollary shows that $\frac{1}{7}$ is the exact answer for every insertion after the sixth.

COROLLARY 1. *The probability that a random insertion into a random height-balanced tree of size n falls into an* M*-subtree is* $\frac{3}{7}$ *for* $n \geqq 6$.

Proof. Since an M-subtree has three external nodes, this probability is

$$\begin{aligned}\sum_{M,N}\frac{3M}{n+1}P_n(M,N) &= \frac{3}{n+1}\sum_{M,N} M\cdot P_n(M,N)\\ &= \frac{3}{n+1}\bar{M}_n = \frac{3}{7} \quad \text{for } n \geqq 6, \text{ by Theorem 1.}\end{aligned}$$

□

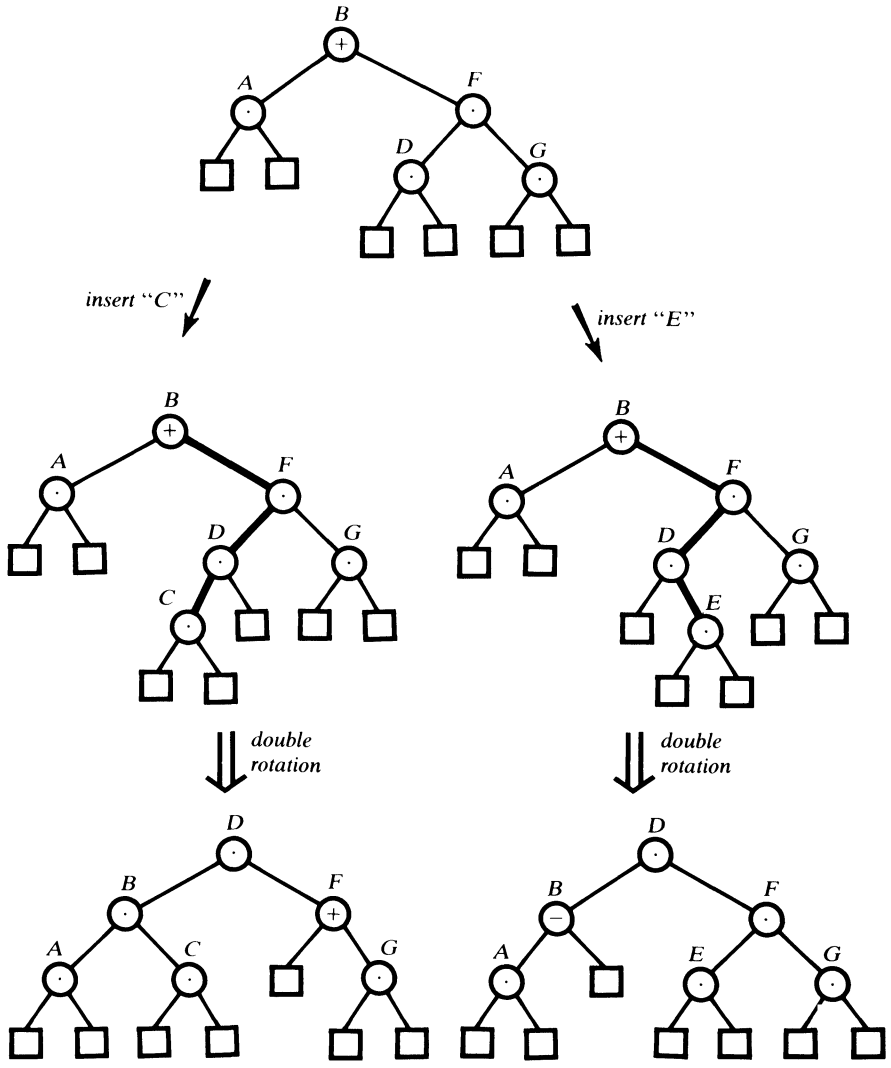


FIG. 7. *Cases in which rotation changes the fringe.*

Theorem 1 also shows that roughly $\frac{4}{7}$ of the internal nodes of a random height-balanced tree lie in the fringe. Hence our knowledge that about 75% of these nodes are balanced allows us to prove bounds on the number of balanced nodes in the entire tree.

THEOREM 2. *Let $\bar{B}_n$ denote the average number of balanced nodes in a random height-balanced tree of size n. Then*

$$\tfrac{10}{21}(n+1) \leqq \bar{B}_n \leqq \tfrac{6}{7}(n+1) - 1, \quad \textit{for } n \geqq 6.$$

Proof. Let T be an n-node height-balanced tree with M M-subtrees and N N-subtrees in the fringe, and let B denote the number of balanced nodes in T. The tree T contains $N+M$ balanced nodes in the fringe (one in each fringe subtree), and $N+M-1$ nodes not in the fringe (one fewer than the number of fringe subtrees). At most all of the nonfringe nodes may be balanced, so

$$B \leqq N+M+(N+M-1).$$

Taking averages gives

$$\bar{B}_n \leqq 2(\bar{N}_n + \bar{M}_n) - 1,$$

so the upper bound follows from Theorem 1.

A similar lower bound argument gives $\frac{3}{7}(n+1) \leqq \bar{B}_n$, since we assume that none of the nonfringe nodes are balanced. This assumption is true for certain cases, but if $N > M$ then some of the nonfringe nodes must be balanced. To see this, consider the set of nodes p such that p is the parent of an N-subtree. We say that p has type 0 if it is balanced (i.e., has two N-subtrees as its offspring), type 1 if its other offspring is an M-subtree, and type 2 if its second offspring is a node of type 0. If we let A_i denote the number of nodes of type i in a balanced tree, then clearly $2A_0 + A_1 + A_2 = N$ for $n \geqq 2$ (the left hand side counts the N-subtrees). But $A_1 \leqq M$ and $A_2 \leqq A_0$, so $N \leqq 2A_0 + M + A_0$, or $A_0 \geqq (N-M)/3$. Hence we conclude that when $N > M$ there are at least $(N-M)/3$ balanced nodes outside the fringe. There are still $N+M$ balanced nodes in the fringe, so

$$\bar{N}_n + \bar{M}_n + \sum_{\substack{M,N \\ N \geqq M}} P_n(M, N) \cdot \left(\frac{N-M}{3}\right) \leqq \bar{B}_n.$$

But we only make this inequality weaker by including terms in the sum for which $(N-M)/3$ is negative. Thus

$$\bar{N}_n + \bar{M}_n + \sum_{M,N} P_n(M, N)\left(\frac{N-M}{3}\right) \leqq \bar{B}_n$$

$$\bar{N}_n + \bar{M}_n + \left(\frac{\bar{N}_n - \bar{M}_n}{3}\right) \leqq \bar{B}_n,$$

and now the lower bound follows from Theorem 1. □

Knuth's experiments indicate that $\bar{B}_n$ is approximately $0.68n$ [3, p. 462]; this is slightly larger than the average of the upper and lower bounds given in Theorem 2. While the gap between the bounds of Theorem 2 is large, these inequalities are an improvement over bounds on $\bar{B}_n$ derived from the minimum and maximum number of balanced nodes possible in any n node balanced tree. The minimum number is about $1/\phi^2 n = 0.38197n$, which is attained asymptotically by the Fibonacci trees [3, Exercise 6.2.3–3]; the maximum number is n, which is attained by the complete binary tree when $n = 2^k - 1$.

Yao's analysis of random 2–3 trees can in principle be carried out to an arbitrary degree of accuracy. (We say "in principle" because the computational effort required by the method quickly becomes too large to contemplate.) It is therefore interesting to consider whether this partial analysis of balanced trees can be extended in a similar way.

The natural method of extending the analysis is to include larger subtrees, and therefore *more* subtrees, in it. The collection of subtrees must form a "closed class" in the sense that an insertion into one of the subtrees of the class always produces either another subtree of the class or produces a tree consisting of two or more subtrees of the class tied together by some extra nodes at the root. It must also be possible to prove results analogous to Lemmas 2 and 3 which define the effect of an insertion into each external node of every subtree in the class.

With height-balanced trees it is not clear how to find a larger class of subtrees satisfying these requirements. The problem is that if one of the subtrees contains

a balanced path from an external node through the root, then an insertion into the subtree can cause a rotation at the node above the root of the subtree. The tree which results from this rotation is not determined by the structure of the subtree where the insertion occurred—it also depends on the other subtree of the node where the rotation takes place. The reason we were able to analyze the class containing M- and N-subtrees is simply that there is only one possible "other subtree" in this case, as shown in the proof of Lemma 3. Thus it appears that a different kind of argument will be required for a more complete analysis of random height-balanced trees.

Acknowledgment. The author would like to thank Lyle Ramshaw and Donald Knuth for their valuable criticisms of an earlier draft, and Phyllis Winkler of Stanford for typing it.

REFERENCES

[1] G. M. A'DELSON-VEL'SKII AND E. M. LANDIS, *An algorithm for the organization of information*, Dokl. Akad. Nauk SSSR, 146 (1962), pp. 263–266; English translation Sov. Math. Dokl., 6 (1963), pp. 1259–1263.

[2] P. L. KARLTON, S. H. FULLER, R. E. SCROGGS AND E. B. KAEHLER, *Performance of height-balanced trees*, Comm. ACM 19, 1 (1976), pp. 23–28.

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

[4] ———, *Big omicron and big omega and big theta*, SIGACT News, 8, 2 (1976), pp. 18–24.

[5] ANDREW C.-C. YAO, *On random* 2-3 *trees*, Acta Informat., 9 (1978), pp. 159–170.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0004 $01.00/0

OPTIMAL 2,3-TREES*

RAYMOND E. MILLER†, NICHOLAS PIPPENGER†, ARNOLD L. ROSENBERG† AND LAWRENCE SNYDER‡

Abstract. The 2,3-trees that are optimal in the sense of having minimal expected number of nodes visited per access are characterized in terms of their "profiles". The characterization leads directly to a linear-time algorithm for constructing a K-key optimal 2,3-tree for a sorted list of K keys. A number of results are derived that demonstrate how different in structure these optimal 2,3-trees are from their "average" cousins.

Key words. 2,3-trees, B-trees, enumeration

Introduction. Many algorithms use as their principal data structure a "search tree" in which records may be located when present, inserted when absent, and deleted when unwanted in time logarithmic in their number. AVL trees and 2,3-trees (a/k/a 3-2 trees, a/k/a 2-3 trees) are examples of this kind of structure. Both have the property that a number of different representations for the same set of records are permissible within the limits of the definitions of the respective structures. The logarithmic performance is guaranteed regardless of which structure arises, but a natural question is, "What, if any, are the quantitative differences among these different representations?"

This paper addresses that question for 2,3-trees and their generalization, B-trees. We derive a characterization of those 2,3-trees (§ 2) and those B-trees (§ 4) that are optimal in the sense of having minimal expected path length per access. Our characterization directly yields a linear-time algorithm for constructing optimal trees. We round out our study by demonstrating how different in structure these optimal trees are from their "typical" cousins and how rare they are in the forests of 2,3-trees and B-trees, respectively (§ 3).

1. 2,3-Trees and their costs. In this section we prepare the way for our study of optimal 2,3-trees. We assume familiarity with trees and their related notions.

(1.1) A 2,3-*tree* is a rooted, oriented tree each of whose nonleaf nodes has either 2 or 3 successors, and all of whose root-to-leaf paths have the same length. We assume the root is not a leaf.

The use of 2,3-trees as balanced search trees (which use originates in unpublished work by Hopcroft) involves placing keys at the nonleaf nodes of the trees—the leaves are dummy nodes—according to the following discipline. A node with s successors ($s = 2,3$) accomodates $s-1$ keys. All keys in the left (resp., right) subtree rooted at a given node are smaller in magnitude (resp., larger in magnitude) than the key(s) resident in the node; should $s = 3$, the keys in the center subtree are strictly intermediate in magnitude between the keys resident in the node; see Fig. 1. The reader familiar with the literature on 2,3-trees will recognize this description as cleaving to the variant presented by Knuth [2, § 6.2.3] rather than that discussed in [1, §§ 4.4, 4.5].

* Received by the editors May 2, 1977.

† Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

‡ Mathematical Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. Permanent Address: Department of Computer Science, Yale University, New Haven, Connecticut 06520.

We now delineate those structural features of 2,3-trees that enter into our characterization of optimal trees.

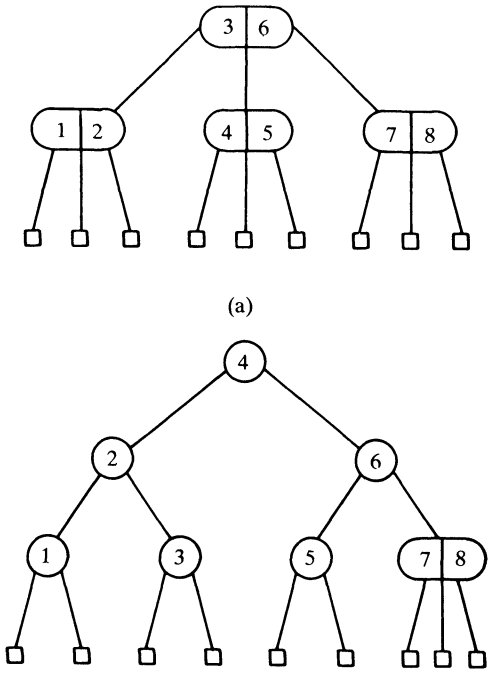


FIG. 1. (a) *A bushy 2,3-tree with 8 keys.* (b) *A scrawny 2,3-tree with 8 keys.*

(1.2) The root of a 2,3-tree is said to be at *level* 0; the direct successors of a node at *level* l are said to be at *level* $l+1$. The *depth* of the tree is the (common) level d of its leaves. The *height* of a node is $d-$(its level).

With each level l of a 2,3-tree, we associate three integers:

β_l = the number of *b*inary (2-successor) nodes at level l;
τ_l = the number of *t*ernary (3-successor) nodes at level l;
ν_l = the *n*umber of nodes at level l.

We combine these integers to yield the following descriptors of the tree.

(1.3) (a) The *profile* of a depth d 2,3-tree is the sequence

$$\Pi = \nu_0, \nu_1, \cdots, \nu_d.$$

(b) The *detailed profile* of the same tree is the sequence

$$\Delta = \langle\beta_0, \tau_0\rangle\langle\beta_1, \tau_1\rangle \cdots \langle\beta_d, \tau_d\rangle.$$

The reader can easily verify the following useful relationships among the quantities we have been discussing:

(1.4) (a) $\nu_0 = 1$;
(b) $\nu_d = 1 +$ (the number of keys in the tree);
(c) $\nu_l = \beta_l + \tau_l$;
(d) $\nu_{l+1} = 2\beta_l + 3\tau_l$;
(e) (the number of keys at level l) $= \beta_l + 2\tau_l = \nu_{l+1} - \nu_l$.

The 8-key trees of Fig. 1 enjoy the following descriptors.

	Tree of Fig. 1(a)	Tree of Fig. 1(b)
Π	1,3,9	1,2,4,9
Δ	$\langle 0,1\rangle\langle 0,3\rangle\langle 0,0\rangle$	$\langle 1,0\rangle\langle 2,0\rangle\langle 3,1\rangle\langle 0,0\rangle$

There are at least two significant measures of the cost of a 2,3-tree, the expected number of key-comparisons per access and the expected number of node-visits per access. The latter measure would likely be the more significant in an environment where a ternary comparator were available or in a paging environment where edge-traversals carried with them the danger of page faults. The former measure would likely be the more significant in an environment where the entire tree resided in main memory and only binary comparators were available. In this paper, we study the latter measure of cost; the last two authors have prepared a paper [3] in which they characterize those 2,3-trees that are optimal with respect to the expected number of key-comparisons.

(1.5) The (node-visit) *cost* of a 2,3-tree T with detailed profile $\Delta = \langle \beta_0, \tau_0\rangle \cdots \langle \beta_d, \tau_d\rangle$ is

$$\mathrm{COST}(T) = \sum_{l=0}^{d-1} (l+1)(\beta_l + 2\tau_l).$$

The cost (1.5) is clearly K times the expected number of nodes per visited access if T contains K keys. Clearly, all trees having the same detailed profile are equally costly. In fact this assertion can be strengthened by removing the qualifier "detailed".

LEMMA 1.1. *If the* 2,3*-tree* T *has profile* $\Pi = \nu_0, \nu_1, \cdots, \nu_d$, *then*

$$\mathrm{COST}(T) = d\nu_d - \sum_{l=0}^{d-1} \nu_l.$$

Hence, trees having the same profile are equally costly.

Proof. If one substitutes equation (1.4e) into the expression (1.5) for COST (T), one finds that

$$\mathrm{COST}(T) = \sum_{l=0}^{d-1} (l+1)(\nu_{l+1} - \nu_l). \tag{1.6}$$

Summing (1.6) by parts yields the result directly. □

Lemma 1.1 affords us one easy technique for deriving costs of 14 and 20, respectively, for the trees of Figure 1(a) and 1(b). In fact, the greater cost of the tree of Figure 1(b) is predicted by the following result which asserts that added depth means added cost.

LEMMA 1.2. *Let* T *and* T' *be* 2,3*-trees, both containing* K *keys, having profiles* $\Pi = \nu_0, \cdots, \nu_d$ *and* $\Pi' = \nu'_0, \cdots, \nu'_e$, *respectively. If* $d < e$, *then* COST (T) < COST (T').

Proof. The positivity of the difference

$$\mathrm{COST}(T') - \mathrm{COST}(T) = e\nu'_e - d\nu_d - \sum_{k=0}^{e-1} \nu'_k + \sum_{l=0}^{d-1} \nu_l$$

is easily established via the following facts: (a) $\nu'_e = \nu_d$ by (1.4b); (b) $e - d \geqq 1$ by

hypothesis; (c)

$$\sum_{k=0}^{e-1} \nu'_k < \nu'_e = \nu_d$$

since a 2,3-tree has more leaves than internal nodes; (d)

$$\sum_{l=0}^{d-1} \nu_l \geqq 1$$

since we insist that roots not be leaves. □

Lemma 1.2 points at a necessary condition for cost-optimality of a 2,3-tree, namely, minimum depth. The nonsufficiency of this condition is illustrated by the two 5-key, depth 2 trees of Fig. 2: the tree of Fig. 2(a) has cost 8 while that of Fig. 2(b) has cost 9. Thus our characterization of optimal trees must await further conditions, which will be developed in the next section.

For the remainder of the paper, we shall adopt the following abbreviations whose motivation will become clear in § 2:

(1.7) A K-key 2,3-tree is *bushy* if its cost (1.5) is minimum among K-key 2,3-trees. The tree is *scrawny* if its cost is maximum among these trees.

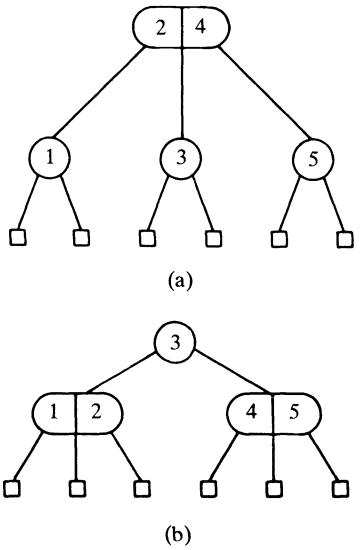


FIG. 2. (a) *A bushy 2,3-tree with 5 keys.* (b) *A scrawny 2,3-tree with 5 keys.*

2. Bushy trees. In this section we develop the two components of our main result. We begin with our characterization of bushy trees; and we follow with the linear-time algorithm that derives from the characterization. We close the section with a discussion of an interesting sidelight of our development.

2.1. The characterization theorem. We lead up to our theorem with two lemmas that expose facets of the structure of bushy trees that are needed in the theorem. These structural properties are of some interest in their own rights.

(2.1) Let the 2,3-tree T have profile $\Pi = \nu_0, \nu_1, \cdots, \nu_d$. The *$k$-prefix* of T ($1 \leqq k \leqq d$), denoted $T^{(k)}$, is the 2,3-tree obtained by replacing all of T's level k nodes by leaves. $T^{(k)}$ thus has profile $\Pi^{(k)} = \nu_0, \nu_1, \cdots, \nu_k$.

LEMMA 2.1. *Every prefix of a bushy tree is bushy.*

Proof. Say for contradiction that the prefix $T^{(k)}$ of the bushy tree T is not bushy. Let T' be a bushy tree with the same number of keys—hence, the same number of leaves—as $T^{(k)}$. Let T^* be the tree obtained by appending to each leaf of T' the subtree rooted at the corresponding leaf of $T^{(k)}$ in T. The construction of T^* should be obvious from Fig. 3.

Now, T^* clearly contains the same number of keys as does T. However, it is a straightforward matter to verify (using Lemma 1.1) that

$$\text{COST}(T^*) \leqq \text{COST}(T) - \text{COST}(T^{(k)}) + \text{COST}(T') < \text{COST}(T),$$

which contradicts the alleged bushiness of T. □

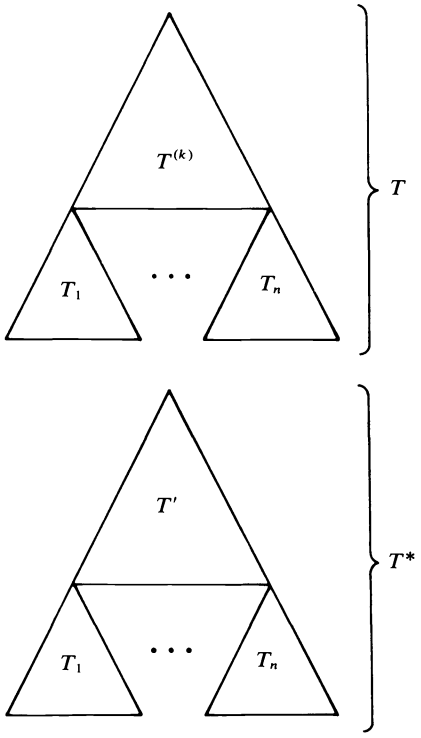


FIG. 3. *The construction of T^* from T and T' in the proof of Lemma* 2.1.

PROPOSITION 2.2. *There exist bushy trees with nonbushy subtrees. Thus, Lemma* 2.1 *cannot be strengthened by replacing "prefix" by "subtree".*

Proof. Immediate upon comparing the trees of Fig. 4. □

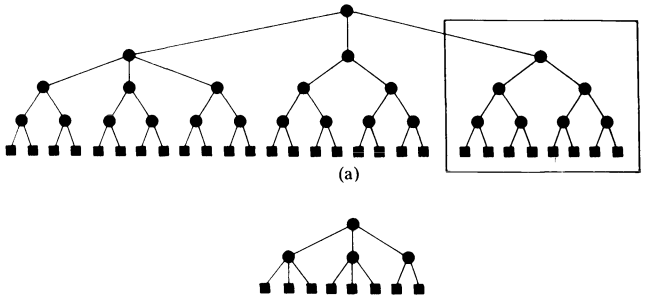


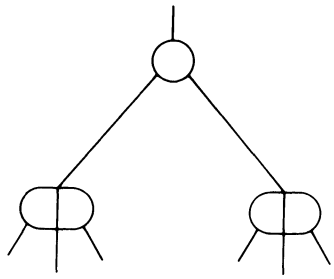
FIG. 4. *Bushy trees with* (a) 27 *and* (b) 7 *keys for the proof of Propositions* 2.2.

The next lemma peers a bit deeper into the structure of bushy trees.

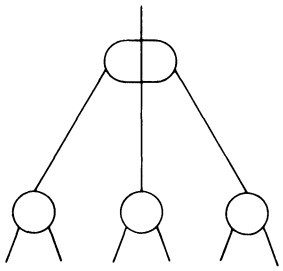
LEMMA 2.3. *If a 2,3-tree has a binary node at level i ($\beta_i > 0$) and two (or more) ternary nodes at level $j > i$ ($\tau_j > 1$), then it is not bushy.*

Proof. Suppose, for contradiction, that the bushy tree T has a binary node at level i and two ternary nodes at level $j > i$. We may assume that $j = i + 1$, for if not, we must have a binary node at level $i + 1$ (since $\nu_{i+1} \geqq 2$), and we may shift attention from the original binary node to this one. Continuing in this way, we must find a level l with $\beta_l > 0$ and $\tau_{l+1} > 1$.

Since trees with the same profile are equal in cost (Lemma 1.1), we may assume further that the ternary nodes are direct successors of the binary node. Thus we have the configuration



in T (or in a tree that shares T's profile). It is easy to verify, however, that the cost (1.5) can be reduced by replacing this configuration with the configuration



This contradicts T's alleged bushiness. □

Although the necessary condition of Lemma 2.3 is not sufficient—cf. the 7-key trees of Fig. 4—it combines with the depth condition of Lemma 1.2 to yield the sought characterization. The conjoined conditions are best presented in the following numerological setting.

(2.2) The profile $\Pi = \nu_0, \cdots, \nu_d$ of a K-key 2,3-tree is *dense* if
(a) $d = \lceil \log_3 (K+1) \rceil$;
(b) $\nu_l = \min (3^l, \lfloor \nu_{l+1}/2 \rfloor)$ for $1 \leqq l \leqq d-1$.
Note that $\nu_0 = 1$ and $\nu_d = K + 1$ automatically.

THEOREM 2.4. *A 2,3-tree is bushy iff it has a dense profile.*

Proof. It will suffice to show that a bushy tree has a dense profile, for once this is done the following argument gives the converse. Let T have a dense profile and let T' be bushy. Then T' has a dense profile. Since there is only one dense profile, T' has the same profile as T. Since the profile determines the cost (Lemma 1.1), T' has the same cost as T. Thus T is also bushy.

Suppose T is a bushy tree with K keys. We seek to show that its profile satisfies (2.2a) and (2.2b). The first of these is easy, for $d = \lceil \log_3 (K+1) \rceil$ is clearly the

minimum possible depth of a 2,3-tree with K keys, and by Lemma 1.2 a deeper tree would have a greater cost.

For the rest, we proceed by induction. The result is trivial for 1 or 2 keys; let us assume that it holds for all trees with fewer than K keys and prove it for those with K keys.

It will suffice to prove (2.2b) for $l = d-1$, for once this is done, we may consider the prefix of T of depth $d-1$. By Lemma 2.1, this is bushy. By inductive hypothesis, it has a dense profile. This gives (2.2b) for the remaining values of l.

It remains to prove that

$$\nu_{d-1} = \min(3^{d-1}, \lfloor \nu_d/2 \rfloor).$$

Clearly,

$$\nu_{d-1} \leqq 3^{d-1},$$

for a 2,3-tree cannot have more than 3^{d-1} nodes at level $d-1$. Furthermore,

$$\nu_{d-1} \leqq \lfloor \nu_d/2 \rfloor,$$

for every node at level $d-1$ has at least 2 successors. We must show that one of these bounds is attained. Suppose, on the contrary, that

$$\nu_{d-1} < 3^{d-1}$$

and

$$\nu_{d-1} < \lfloor \nu_d/2 \rfloor.$$

From the first of these it follows that there is a binary node at or above level $d-2$, and from the second it follows that there are at least two ternary nodes at level $d-1$. Thus, by Lemma 2.3, T is not bushy, a contradiction. □

It follows immediately from the Theorem that a bushy tree has at most two "active" or unsaturated levels.

PROPOSITION 2.5. *If* $\Pi = \nu_0, \cdots, \nu_d$ *is a dense profile, then* $\nu_i = 3^i$ *for all* $i < d-2$.

Proof. Since $d = \lceil \log_3 \nu_d \rceil$ by (2.2a), we know that $\nu_d > 3^{d-1}$. Hence, $\nu_d/8 > 3^{d-1}/8 = 3^d/24 > 3^d/27 = 3^{d-3}$, so that $\nu_{d-3} = 3^{d-3}$ by (2.2b), since $\lfloor \lfloor \lfloor \nu_d/2 \rfloor/2 \rfloor/2 \rfloor > \nu_d/8 - 1$. Moreover, since

$$3^k \leqq \lfloor 3^{k+1}/2 \rfloor \quad \text{for all } k,$$

we are assured that $\nu_i = 3^i$ for all $i \leqq d-3$, as was claimed. □

2.2. An algorithm for constructing bushy trees. Our characterization of bushiness in terms of dense profiles yields directly an algorithm for constructing a bushy tree for a given set of keys. If the input set of keys is already sorted, then the algorithm is linear in the size of the set; otherwise, the algorithm operates in time $O(K \log K)$. (No better timing could be expected since the set is sorted once it resides in the tree.)

We interlace our description of the algorithm with an example.

THE ALGORITHM. Our algorithm can best be described in four phases.

Phase 1. Given the cardinality K of the set of keys to be stored, use the prescription (2.2) to construct the profile of a bushy K-tree.

(2.3) If $K = 14$, then $\Pi = 1, 3, 7, 15$.

Phase 2. Using the equations

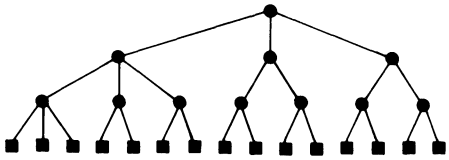
(2.4) $$\beta_i = 3\nu_i - \nu_{i+1}, \qquad \tau_i = \nu_{i+1} - 2\nu_i,$$

construct the detailed profile of the tree from its profile.

(2.5) Given the profile Π of (2.3), we have $\Delta = \langle 0, 1\rangle\langle 2, 1\rangle\langle 6, 1\rangle\langle 0, 0\rangle$.

Phase 3. Construct the "skeleton" of the tree from its detailed profile; that is, decide how to place the binary and ternary nodes at those levels that have both. Clearly, this decision will not affect the cost of the resulting tree, but the layout may affect the efficiency of subsequent transactions with the key set. Other things being equal, a decision to *left-bias* the tree by forcing all ternary nodes as far to the left as possible is as good as any other.

(2.6) The left-biased tree with the detailed profile Δ of (2.5) has the following appearance:



Phase 4. Traverse the tree of Phase 3 in FILLORDER, dropping off the keys in ascending order as one goes.

(2.7) To traverse a tree in FILLORDER, follow the ensuing recursive prescription.

1. Visit the left subtree in FILLORDER.
2. Visit the root, and deposit a key.
3. Visit the center subtree in FILLORDER.
4. Visit the root, and deposit a key.
5. Visit the right subtree in FILLORDER.

(3, 4, 5): for ternary roots only.

(2.8) We finally complete the example of (2.3), (2.5), (2.6). We use the key set $\{1, \cdots, 14\}$ to illustrate in Figure 5 the FILLORDER of the tree of (2.6).

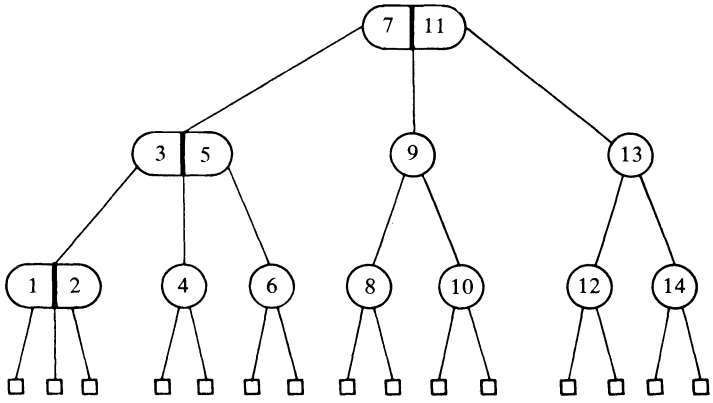


FIG. 5. *The* FILLORDER *of the tree* (2.5).

Timing of the Algorithm. We assume that our algorithm is to be executed on a uniform-cost RAM [1, § 1]. Accordingly, we assess time $O(\log K)$ for the $\lceil \log_3 (K + 1)\rceil$ operations performed in Phase 1 and for the $2\lceil \log_3 (K + 1)\rceil$ linear-form evaluations in Phase 2. Phases 3 and 4, which likely would be done simultaneously in an efficient implementation, can be seen to take time $O(K)$ to perform if the list of keys is

sorted, and time $O(K \log K)$ otherwise. (Note that FILLORDER traversal of a tree is almost identical to depth-first traversal.)

Discussion. The obvious algorithm for constructing a bushy 2,3-tree would construct the tree top-down, making it as ternary as possible, with some backtracking at the high-numbered levels to ensure a "flat bottom." Our use of profiles and detailed profiles in our algorithm obviates this backtracking, thus enhancing the efficiency of the construction. A logical competitor for any direct-construction procedure would be one that constructs a 2,3-tree by successively inserting, in ascending order, say, the keys one is given, according to the insertion algorithm for 2,3-trees [2, § 6.2.3]. The reader can easily reproduce the induction that demonstrates that trees produced in this way are often very far from bushy. Specifically, whenever, $K = 2^n - 1$, the tree so produced is a purely binary tree!

2.3. Characterizing scrawny trees. There is a striking and appealing duality between our characterization of optimal 2,3-trees on the one hand and the analogous characterization for pessimal or *scrawny* 2,3-trees.

METATHEOREM. *In order to reproduce the results of* § 2.1 *for scrawny trees, perform the following transliteration throughout.*

For		*Read*
2, 3	even in bases of logs and exponentials	3, 2
min		max
floor $\lfloor x \rfloor$		ceiling $\lceil x \rceil$
ceiling $\lceil x \rceil$		floor $\lfloor x \rfloor$
τ		β
β		τ

Details are left to the reader.

3. Typical 2,3-trees. We have found the optimal (bushy) and pessimal (scrawny) 2,3-trees; let us have a look at typical, run-of-the-forest 2,3-trees. We shall find that almost all n-leaf 2,3-trees share some remarkable statistical properties involving the golden ratio,

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618\cdots.$$

These properties, which will allow us to predict the cost of a typical 2,3-tree, will be obtained as by-products of an argument for enumerating 2,3-trees.

Let T_n denote the number of n-leaf 2,3-trees. We shall show that

$$T_n = \frac{1}{n}\phi^n U(1), \tag{3.1}$$

where $U(1)$ denotes a factor of the form $\exp O(1)$. More generally, since we shall be dealing with "error factors" more often than with "error terms", we shall let $U(f(n))$ denote a factor of the form $\exp O(f(n))$. In ordinary language, (3.1) determines T_n to within constant factors.

The major steps in our derivation of (3.1) will be as follows. First we shall obtain a recurrence having the sequence T_n as its unique fixed point (Lemma 3.1). Then we shall show that any sequence that is an "approximately fixed" point of the recurrence must be "approximately equal" to the exact fixed point (Lemma 3.2). Finally we shall show that

$$\frac{1}{n}\phi^n$$

is an approximately fixed point of the recurrence (Lemmas 3.3, 3.4, 3.5). In the derivation, the notions "approximately fixed" and "approximately equal" will be given precise meanings by means of error factors.

This method of proceeding leaves unanswered the question of how the solution was found in the first place. Experience with the enumeration of unlabeled trees in general, and consideration of 2,3-trees in particular, suggests that T_n grows exponentially, say as e^{An}. An attempt to prove this, along the lines indicated above, reveals that A must be $\ln \phi$ and that a correction of the form n^B is necessary. Another pass through the proof reveals that $B = -1$. Though with hindsight we might find a more convincing motivation for the solution (by considering, for example, the singularities of the generating function for T_n), this method of iteration is extremely robust and after the first pass one can usually work out the successive corrections with very little wasted motion. An example of a more formidable problem which was solved in the same way will be given later.

LEMMA 3.1. *T_n satisfies the recurrence*

$$T_n = \sum_{2\beta+3\tau=n} \binom{\beta+\tau}{\tau} T_{\beta+\tau}. \tag{3.2}$$

Proof. Given an n-leaf 2,3-tree, consider the nodes at height one, that is, the nodes whose successors are leaves. If β and τ denote the number of binary and ternary nodes, respectively, at height one, then $2\beta + 3\tau = n$.

An n-leaf 2,3-tree can be constructed by the following three-step procedure. First, choose β and τ satisfying $2\beta + 3\tau = n$. Second, choose the structure of the tree below height one. This amounts to choosing τ of the $\beta + \tau$ nodes at height one to be ternary nodes, leaving the remaining β to be binary nodes. This can be done in

$$\binom{\beta+\tau}{\tau}$$

ways. Third, choose the structure of the tree above height one. This amounts to choosing a $(\beta+\tau)$-leaf 2,3-tree, and can be done in $T_{\beta+\tau}$ ways. Since each n-leaf 2,3-tree can be constructed in exactly one way by this procedure, we arrive at (3.2). □

The recurrence (3.2), together with the initial conditions $T_2 = 1$ and $T_3 = 1$, completely determines T_n.

LEMMA 3.2. *If S_n is a positive sequence satisfying*

$$S_n = U(f(n)) \sum_{2\beta+3\tau=n} \binom{\beta+\tau}{\tau} S_{\beta+\tau}, \tag{3.3}$$

where $f(n)$ is an eventually decreasing positive function such that

$$\sum_{0 \le t \le \infty} f(2^t) \tag{3.4}$$

converges, then

$$T_n = S_n U(1). \tag{3.5}$$

Proof. Let A_N denote the maximum of S_n/T_n for $1 \leqq n \leqq N$. If N is large enough that $f(n)$ is decreasing beyond N, and if $N \leqq n \leqq 2N$, then

$$\begin{aligned} S_n &= U(f(n)) \sum_{2\beta+3\tau=n} \binom{\beta+\tau}{\tau} S_{\beta+\tau} \\ &\leqq U(f(n)) \sum_{2\beta+3\tau=n} \binom{\beta+\tau}{\tau} A_N T_{\beta+\tau} \\ &= U(f(n)) A_N T_n \\ &\leqq U(f(N)) A_N T_n, \end{aligned}$$

since $n \leqq 2N$ implies $\beta+\tau \leqq N$, and $f(n) \leqq f(N)$. Thus

$$A_{2N} \leqq A_N U(f(N)),$$

and by induction

$$\begin{aligned} A_{2^t N} &\leqq A_N \prod_{0 \leqq s < t} U(f(2^s N)) \\ &= A_N U\Big(\sum_{0 \leqq s < t} f(2^s N) \Big). \end{aligned}$$

Letting $t \to \infty$ with N fixed, we have

$$A_{2^t N} \leqq U(1),$$

since A_N is positive and (3.4) converges. Thus

$$S_n/T_n \leqq U(1);$$

a similar argument shows that

$$S_n/T_n \geqq U(1),$$

so (3.5) is proved. □

It remains for us to show that

$$\frac{1}{n}\phi^n$$

is an approximately fixed point of our recurrence. Specifically, we shall show that

$$\sum_{2\beta+3\tau=n} \binom{\beta+\tau}{\tau} \frac{1}{\beta+\tau} \phi^{\beta+\tau} = \frac{1}{n}\phi^n U\Big(\frac{\log^{3/2} n}{n^{1/2}}\Big). \tag{3.6}$$

This will be done in three steps as follows. First we shall estimate the summand, separating our estimate into algebraically varying factors (which are $U(\log n)$) and exponentially varying factors (which are $U(n)$). We shall then focus our attention on the exponentially varying factors and see that they impart to the summand a peaking reminiscent of the central limit theorem: the greatest contribution to the sum comes from those terms in which β and τ are in certain fixed ratios to $\beta+\tau$ and hence to n. Finally we shall use this central peaking to estimate the sum.

Successive values of β and τ differ by 3 and 2, respectively; it will be convenient to have an index whose successive values differ by 1. Thus we introduce the index m satisfying

$$\tau = 2m, \qquad \beta = \frac{n}{2} - 3m, \qquad \beta + \tau = \frac{n}{2} - m.$$

This index assumes integral values if n is even and half-integral values if n is odd.

LEMMA 3.3.

$$\binom{\beta+\tau}{\tau}\frac{1}{\beta+\tau}\phi^{\beta+\tau} = U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{1}{2\pi\beta\tau(\beta+\tau)}\right)^{1/2} \exp nE\left(\frac{m}{n}\right), \tag{3.7}$$

where

$$E(\mu) = F(G(\mu)), \qquad F(\lambda) = \frac{H(\lambda)+\ln\phi}{2+\lambda},$$

$$G(\mu) = \frac{4\mu}{1-2\mu}, \qquad H(\lambda) = -\lambda\ln\lambda - (1-\lambda)\ln(1-\lambda),$$

and ln *denotes the natural logarithm.*

Proof. For the binomial coefficient, the estimate

$$\binom{\beta+\tau}{\tau} = U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{\beta+\tau}{2\pi\beta\tau}\right)^{1/2} \exp(\beta+\tau)H\left(\frac{\tau}{\beta+\tau}\right)$$

is an immediate consequence of Stirling's formula. Define λ such that

$$\tau = \lambda(\beta+\tau), \qquad \beta = (1-\lambda)(\beta+\tau), \qquad n = (2+\lambda)(\beta+\tau).$$

Then

$$\binom{\beta+\tau}{\tau}\frac{1}{\beta+\tau}\phi^{\beta+\tau} = U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{1}{2\pi\beta\tau(\beta+\tau)}\right)^{1/2} \exp nF\left(\frac{\tau}{\beta+\tau}\right).$$

Define μ such that

$$m = \mu n.$$

Then λ and μ are related by

$$\lambda = \frac{4\mu}{1-2\mu}, \qquad \mu = \frac{\lambda}{2(2+\lambda)}.$$

Thus

$$\binom{\beta+\tau}{\tau}\frac{1}{\beta+\tau}\phi^{\beta+\tau} = U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{1}{2\pi\beta\tau(\beta+\tau)}\right)^{1/2} \exp nE\left(\frac{m}{n}\right),$$

as was to be shown. □

LEMMA 3.4. *The function* $F(\lambda)$ *assumes its unique maximum* (*for* $0 \leqq \lambda \leqq 1$) *at*

$$\Lambda = \phi^{-2}.$$

At this point

$$F(\Lambda) = \ln\phi, \qquad F'(\Lambda) = 0, \qquad F''(\Lambda) = \frac{-1}{(2+\Lambda)\Lambda(1-\Lambda)},$$

where the primes indicate differentiation. Accordingly, $E(\mu)$ assumes its maximum at

$$M = \frac{\Lambda}{2(2+\Lambda)},$$

and at this point

$$E(M) = \ln \phi, \qquad E'(M) = 0, \qquad E''(m) = -\frac{(2+\Lambda)^3}{\Lambda(1-\Lambda)}.$$

Proof. We shall let $H(0) = H(1) = 0$; this makes $H(\lambda)$, and therefore also $F(\lambda)$, continuous on the closed interval $0 \leqq \lambda \leqq 1$. These functions are in fact analytic in the open interval $0 < \lambda < 1$, and thus $F(\lambda)$ can assume its maximum only where its first derivative vanishes or at an endpoint. We compute the first derivatives

$$H'(\lambda) = \ln(1-\lambda)/\lambda,$$

$$F'(\lambda) = -\frac{H(\lambda)+\ln\phi}{(2+\lambda)^2} + \frac{\ln(1-\lambda)/\lambda}{2+\lambda}.$$

Equating $F'(\lambda)$ with 0 leads to the equation

$$(1-\lambda)^3 = \lambda^2\phi.$$

In the interval $0 < \lambda < 1$ the left side decreases while the right side increases, so there can be at most one solution. This occurs at

$$\Lambda = \phi^{-2},$$

by virtue of the equation

$$1 - \phi^{-2} = \phi^{-1}.$$

This gives

$$F(\Lambda) = \ln \phi,$$

which is obviously larger than $F(\lambda)$ at either of the endpoints. We compute the second derivatives

$$H''(\lambda) = \frac{-1}{\lambda(1-\lambda)}, \qquad F''(\lambda) = 2\frac{H(\lambda)+\ln\phi}{(2+\lambda)^3} - 2\frac{\ln(1-\lambda)/\lambda}{(2+\lambda)^2} - \frac{1}{(2+\lambda)\lambda(1-\lambda)}.$$

Since the first two terms are a multiple of $F'(\lambda)$, they vanish at Λ, leaving

$$F''(\Lambda) = \frac{-1}{(2+\Lambda)\Lambda(1-\Lambda)}.$$

All of this can be carried over to $E(\mu)$, $E'(\mu)$, and $E''(\mu)$ through the derivatives

$$G'(\mu) = \frac{4}{(1-2\mu)^2} = (2+\lambda)^2, \qquad G''(\mu) = \frac{16}{(1-2\mu)^3} = 2(2+\lambda)^3$$

and the chain rule. □

LEMMA 3.5.

$$\sum_m U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{1}{2\pi\beta\tau(\beta+\tau)}\right)^{1/2} \exp nE\left(\frac{m}{n}\right) = U\left(\frac{\log^{3/2} n}{n^{1/2}}\right)\frac{1}{n}\phi^n. \tag{3.8}$$

Proof. The major steps of the derivation are as follows. The central peaking of the summand will be exploited, allowing the tails of the summation to be neglected. The decaudated sum can be simplified since the algebraically varying factors behave like constants in the remaining range of summation. The resulting sum will be estimated with an integral, to which the tails previously removed will be restored. The recaudated integral can be evaluated by standard methods.

Our sum is

$$\sum_m W_m,$$

where

$$W_m = U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{1}{2\pi\beta\tau(\beta+\tau)}\right)^{1/2} \exp nE\left(\frac{m}{n}\right).$$

Since $E(\mu)$ is analytic at M, it can be expanded in a Taylor series about M. The result is

$$E(\mu)=\ln\phi-(\mu-M)^2/\delta^2+O((\mu-M)^3),$$

where

$$\delta=\left(\frac{2\Lambda(1-\Lambda)}{(2+\Lambda)^3}\right)^{1/2}.$$

Thus W_m can be rewritten as

$$W_m = U\left(\frac{1}{\beta}+\frac{1}{\tau}\right)\left(\frac{1}{2\pi\beta\tau(\beta+\tau)}\right)^{1/2} U((m-Mn)^3/n^2)\phi^n V_m,$$

where

$$V_m=\exp-(m-Mn)^2/\delta^2 n.$$

We shall break our sum into three parts,

$$\sum_m W_m=\sum_{m<a} W_m+\sum_{a\leqq m\leqq b} W_m+\sum_{b<m} W_m,$$

where

$$a=Mn-\left(\frac{6\Lambda(1-\Lambda)n\ln n}{(2+\Lambda)^3}\right)^{1/2},$$

$$b=Mn+\left(\frac{6\Lambda(1-\Lambda)n\ln n}{(2+\Lambda)^3}\right)^{1/2}.$$

For any term in the sum over $m<a$,

$$V_m=O(V_a)=O\left(\frac{1}{n^3}\right),$$

and the other factors in W_m are $O(\phi^n)$. Since there are $O(n)$ terms,

$$\sum_{m<a} W_m=O\left(\frac{1}{n^2}\phi^n\right).$$

A similar argument shows that

$$\sum_{b<m} W_m=O\left(\frac{1}{n^2}\phi^n\right),$$

so

$$\sum_m W_m = \sum_{a \leqq m \leqq b} W_m + O\left(\frac{1}{n^2}\phi^n\right). \tag{3.9}$$

For any term in the sum over $a \leqq m \leqq b$,

$$m = MnU\left(\frac{\log^{1/2} n}{n^{1/2}}\right),$$

from which it follows that

$$\tau = \frac{\Lambda n}{2+\Lambda} U\left(\frac{\log^{1/2} n}{n^{1/2}}\right), \qquad \beta = \frac{(1-\Lambda)n}{2+\Lambda} U\left(\frac{\log^{1/2} n}{n^{1/2}}\right), \qquad \beta + \tau = \frac{n}{2+\Lambda} U\left(\frac{\log^{1/2} n}{n^{1/2}}\right),$$

and further that

$$W_m = U\left(\frac{\log^{3/2} n}{n^{1/2}}\right)\left(\frac{(2+\Lambda)^3}{2\pi n^3 \Lambda(1-\Lambda)}\right)^{1/2} \phi^n V_m.$$

Thus

$$\sum_{a \leqq m \leqq b} W_m = U\left(\frac{\log^{3/2} n}{n^{1/2}}\right)\left(\frac{(2+\Lambda)^3}{2\pi n^3 \Lambda(1-\Lambda)}\right)^{1/2} \phi^n \sum_{a \leqq m \leqq b} V_m. \tag{3.10}$$

Now,

$$\sum_{a \leqq m \leqq b} V_m = \int_a^b V_x \, dx + O(1), \tag{3.11}$$

since the total variation of the integrand is $O(1)$. We shall express our integral as the sum of three integrals:

$$\int_a^b V_x \, dx = -\int_{-\infty}^a V_x \, dx + \int_{-\infty}^{+\infty} V_x \, dx - \int_b^{+\infty} V_x \, dx.$$

Integration by parts gives

$$\int_{-\infty}^a V_x \, dx = O\left(\frac{1}{a} V_a\right) = O\left(\frac{1}{n^4}\right).$$

Similar considerations show that

$$\int_b^{+\infty} V_x \, dx = O\left(\frac{1}{n^4}\right),$$

so

$$\int_a^b V_x \, dx = \int_{-\infty}^{+\infty} V_x \, dx + O\left(\frac{1}{n^4}\right). \tag{3.12}$$

Using the transformation

$$x = Mn + \delta n^{1/2} y$$

and the well-known integral

$$\int_{-\infty}^{+\infty} \exp -y^2 \, dy = \pi^{1/2},$$

we obtain

$$\int_{-\infty}^{+\infty} V_x\,dx = \left(\frac{2\pi n\Lambda(1-\Lambda)}{(2+\Lambda)^3}\right)^{1/2}.$$

Working backwards through (3.12), (3.11), (3.10), and (3.9), we arrive at (3.8). □

At last we have

THEOREM 3.6.

$$T_n = \frac{1}{n}\phi^n U(1).$$

Proof. Lemmas 3.3, 3.4, and 3.5, taken together, prove formula (3.6), which, taken together with Lemmas 3.1 and 3.2, proves the theorem. □

The methods we have used to prove this theorem can be used to obtain a fairly complete picture of what a typical n-leaf 2,3-tree looks like. The argument that allowed us to neglect the tails of the sum in Lemma 3.5 shows that, with probability approaching 1 as $n \to \infty$,

$$\beta+\tau = \frac{n}{2+\Lambda}\,U\left(\frac{\log^{1/2} n}{n^{1/2}}\right), \qquad \beta = \frac{(1-\Lambda)n}{2+\Lambda}\,U\left(\frac{\log^{1/2} n}{n^{1/2}}\right), \qquad \tau = \frac{\Lambda n}{2+\Lambda}\,U\left(\frac{\log^{1/2} n}{n^{1/2}}\right).$$

Thus the number of nodes at height one is less than the number of leaves by the factor $2+\Lambda = 2+\phi^{-2} = 2.381\cdots$, and these nodes are partitioned into binary and ternary nodes in the golden ratio $1-\Lambda = \phi^{-1} = 0.618\cdots$, $\Lambda = \phi^{-2} = 0.381\cdots$. The same ratios manifest themselves at greater heights, with the result that, with probability approaching 1 as $n \to \infty$, a 2,3-tree has height $\log_{2+\Lambda} n + O(1)$. This implies that it also has cost $n \log_{2+\Lambda} n + O(n)$. Typical 2,3-trees thus assume a position intermediate between their bushy and scrawny forest-mates:

	bushy	typical	scrawny
cost	$n \log_3 n + O(n)$	$n \log_{2+\Lambda} n + O(n)$	$n \log_2 n + O(n)$

$$(\Lambda = 0.381\cdots)$$

It should be observed that 2,3-trees that are "typical" in the static sense in which we have used the word (with all n-leaf trees considered equally) have nothing to do with those that are "typical" in the dynamic sense of being grown by the standard insertion algorithm (with all $n!$ orders of insertion considered equally). This is easily seen by comparing the average proportions of binary and ternary nodes derived earlier for the static sense with the corresponding average proportion found by Yao [4] for the dynamic sense:

	static	dynamic
binary nodes: $\beta/(\beta+\tau)$	$1-\Lambda = \phi^{-1}$	$\frac{2}{3}$
ternary nodes: $\tau/(\beta+\tau)$	$\Lambda = \phi^{-2}$	$\frac{1}{3}$

The methods of this section can be applied to the number $T_n^{(3)}$ of bushy n-leaf 2,3-trees or to the number $T_n^{(2)}$ of scrawny trees. These numbers do not behave as

smoothly with n as T_n does: for n a power of 3, $T_n^{(3)} = 1$ and for n a power of 2, $T_n^{(2)} = 1$; for other values of n, $T_n^{(3)}$ and $T_n^{(2)}$ may be large. But one can show that

$$T_n^{(3)} = O(n^{-1/2}\psi^n), \qquad T_n^{(2)} = O(n^{-1/2}\psi^n),$$

where $\psi = 1.324\cdots$ is the real root of the equation $\psi^3 = \psi + 1$. These upper bounds are the best possible, in the sense that they become false if $O(\cdots)$ is replaced by $o(\cdots)$. Since $\psi < \phi$, bushy or scrawny trees constitute an exponentially small fraction of all 2,3-trees.

The methods of this section can also be applied to the number P_n of profiles of n-leaf 2,3-trees. The recurrence

$$P_n = \sum_{2\beta+3\tau=n} P_{\beta+\tau}$$

is obtained by analogy with Lemma 3.1. The solution of this recurrence is the same in outline as that of (3.2), but much more elaborate in detail. The result is

$$P_n = U(1)n^{(1/2)\log_2 n - \log_2\log_2 n + \log_2 e - 1/2}(\log_2 n)^{(1/2)\log_2\log_2 n},$$

which is perhaps not what one would have first conjectured.

4. B-trees. All of the results in §§ 1–3 generalize from 2,3-trees to their more practical relatives B-trees [2, § 6.2.3]. Although these generalized results are often harder to prove than their 2,3-relatives, the added difficulty is technical rather than conceptual in nature. Accordingly, we shall discuss the generalizations in only a cursory fashion, pointing out the slight differences in formulation as we go.

(4.1) A *B-tree of order m* ($\geqq 3$) is a rooted, oriented tree whose root has $2 \leqq s \leqq m$ successors, whose nonroot interior nodes have $\lceil m/2 \rceil \leqq s \leqq m$ successors each, and all of whose root-to-leaf paths have the same length.

(4.2) The *detailed profile* of an order m B-tree T is a sequence of $(m-1)$-tuples

$$\Delta = \langle \sigma_0^2, \sigma_0^3, \cdots, \sigma_0^m \rangle \cdots \langle \sigma_d^2, \sigma_d^3, \cdots, \sigma_d^m \rangle$$

where σ_l^s is the number of s-successor nodes at level l of T.

A 2,3-tree is an order 3 B-tree; the quantities earlier denoted β_l and τ_l are now denoted σ_l^2 and σ_l^3, respectively. Obviously, if $l > 1$, all $\sigma_l^s = 0$ for $s < \lceil m/2 \rceil$.

(4.3) The *cost* of the B-tree T with detailed profile Δ as in (4.2) is

$$\mathrm{COST}(T) = \sum_{l=0}^{d-1} (l+1)\left(\sum_{k=2}^{m} k\sigma_l^k \right).$$

Section 1. The results of § 1 and their proofs translate verbatim to our new setting.

Section 2. Lemma 2.1 and its proof translate verbatim. Lemma 2.3 requires some translation, as follows.

LEMMA 2.3′. *If an order m B-tree has an "unsaturated" node at level i (i.e., $\sigma_i^s > 0$ for some $s < m$), and if it has (at least) $\lceil m/2 \rceil - 1$ "available" keys at level $j > i$*

$$\left(i.e., \quad \sum_{\lceil m/2 \rceil < k \leqq m} (k - \lceil m/2 \rceil + 1)\sigma_j^k \geqq \lceil m/2 \rceil - 1 \right),$$

then it is not bushy.

The proof of Lemma 2.3′, as well as that of Theorem 2.4 carry over in a transparent way to B-trees, once one has translated definition (2.2) by replacing 2 and 3 by $\lceil m/2 \rceil$ and m, respectively.

The linear-time algorithm for constructing bushy trees requires only two emendations of any substance in order to accommodate general B-trees. First, in Phase 2 of the algorithm, one replaces the equations (2.4) by the equations

$$\nu_i = \sum_s \sigma_i^s, \qquad \nu_{i+1} = \sum_s s\sigma_i^s; \tag{4.4}$$

hence, the detailed profile of the tree is no longer uniquely specified by the profile. However, one can still produce a detailed profile for the tree from the equations (4.4) in time $O(\log K)$, as the reader can easily verify. The second required change is to the definition (2.7) of FILLORDER; the needed change is obvious.

Finally, the duality between the optimal and pessimal B-tree is almost as striking as that between the corresponding 2,3-trees. The major distinction results from the fact that the "2" in 2,3-trees plays the dual role of $\lceil m/2 \rceil$ and the minimal degree of the root. Thus the scrawny order m B-tree profile satisfies the equations

$$\begin{aligned} d &= \lfloor \log_{\lceil m/2 \rceil} (\nu_d/2) \rfloor + 1, \\ \nu_l &= \max (\lceil m/2 \rceil^l, \lceil \nu_{l+1}/m \rceil), \\ \nu_0 &= 1, \qquad \nu_1 = 2. \end{aligned}$$

The proof of the characterization theorem, however, mirrors that of the characterization of bushy trees, as is the case with 2,3-trees.

Section 3. The B-tree generalization of § 3 can be done, with much more labor but no more insight. The major observable change is that the golden ratio ϕ is replaced by a less familiar algebraic number whose degree depends on the order of the B-trees studied.

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. E. KNUTH, *The Art of Computer Programming III: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[3] A. L. ROSENBERG AND L. SNYDER, *Minimal-Comparison* 2,3-*Trees*, this Journal, 7 (1978), pp. 465–480.

[4] A. C.-C. YAO, *Random* 3-2 *Trees*, Acta Inform, 9 (1978), pp. 159–170.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0005 $01.00/0

A ROUND-OFF ERROR MODEL WITH APPLICATIONS TO ARITHMETIC EXPRESSIONS*

VIJAY B. AGGARWAL† AND JAMES W. BURGMEIER†

Abstract. An arithmetic expression is evaluated in a form most suitable to a given computing structure. To select this "suitable form" restructuring algorithms using laws of associativity, commutativity, and distributivity have been proposed. This raises the question of how different ways of evaluating an expression influence the propagation of errors due to round-off.

An error model consisting of "error vectors" is developed to obtain the absolute error bound satisfied by the computation of a given expression. An error vector algebra is presented that vastly simplifies the calculation of error bounds; yet this model yields the same bounds as other models. The model is used to analyze the error accumulation for different evaluations of division-free arithmetic expressions. With error complexity defined to be the minimum error bound incurred under all modes of evaluation of an expression, a restructuring algorithm is given that minimizes error complexity.

Key words. Arithmetic expression, restructuring, parallel evaluation, round-off error model, error complexity, error vectors, arithmetic laws, minimal error

1. Introduction. During the past few years many articles have appeared concerning algorithms for the computation of arithmetic expressions, especially with regard to parallel evaluation. Using associative, commutative, and distributive laws, these algorithms restructure an arithmetic expression into an equivalent form suitable for parallel evaluation. The predominant theme of these papers has been to compare these restructuring algorithms on the basis of the computational complexity of the resultant form, namely the number of parallel steps required for evaluation. One important aspect of restructuring algorithms is an analysis of the round-off error incurred in the evaluation of the restructured form. This is an open problem mentioned in Brent [1], Muller and Preparata [8], and Stone [10]. This paper is a step toward addressing that shortcoming.

Examples may be cited for which different techniques of evaluating a given arithmetic expression can result in significantly different error bounds. In this paper the error accumulation for different evaluations of division-free arithmetic expressions is analyzed. Our approach will be to develop an error model that facilitates the computation of error bounds satisfied by a given arithmetic expression. This model involves writing an expression in a vector form and computing two associated error vectors by means of some rules established in the Appendix. The desired error bound will then be the inner product of the expression vector with the error vectors. This model possesses at least three important features: (i) the error vector arithmetic rules are fairly simple and through their use the error bound can be found quickly and easily; (ii) the bounds produced in this way agree completely with those obtained by process graphs [2] or Wilkinson's error analysis [11]; and (iii) this model affords an analytic approach for comparing the error in various restructured forms of an arithmetic expression. The last feature provided the original stimulus for the development of the model: we wanted to assess the effects of restructuring on round-off error. The model presented here has enabled us to prove some important results concerning the use of the distributive, commutative, and associative laws in restructuring processes. We show that round-off error bounds are unaffected by such restructuring except for the use of additive associativity.

* Received by the editors September 20, 1976, and in final revised form May 26, 1978.
† Department of Mathematics, University of Vermont, Burlington, Vermont 05401.

2. Assumptions and definitions. When using floating point arithmetic with fixed word length each operation results in an error, referred to as round-off error. It is important to recognize that this error is not due to errors in the quantities used in the computation; rather, it is an independent source of error. Now let E and F be two expressions with absolute errors ΔE and ΔF, respectively. Then the relative error in the sum $E+F$ is given by

$$\frac{\Delta(E+F)}{E+F}=\frac{\Delta E}{E+F}+\frac{\Delta F}{E+F}+r_A$$

where r_A is the round-off error introduced by the addition itself (typically in a binary computer with a t digit mantissa, $|r_A|\leqq 2^{-t+1}$). Thus

$$\Delta(E+F)=\Delta E+\Delta F+(E+F)r_A \tag{2.1}$$

similarly

$$\Delta(E-F)=\Delta E-\Delta F+(E-F)r_S \tag{2.2}$$

and

$$\Delta(E*F)=F\Delta E+E\Delta F+(E*F)r_M. \tag{2.3}$$

Here r_S and r_M are round-off errors introduced due to the operations of subtraction and multiplication, respectively. In (2.3) the term $\Delta E\Delta F$ should also appear, but we shall ignore such products of errors.

Since we are analyzing round-off errors, we shall assume no errors in the input data. Throughout the paper all arithmetic expressions will be division-free. Such an expression E can be written as a sum of products, called terms, $T_1, \cdots, T_k$. Let $\tilde{E}$ be a computed value of E. Repeated use of (2.1)–(2.3) results in an expression for the error consisting of a sum of the terms in E multiplied by round-off errors r_i and by positive integers p_i (due to accumulation):

$$\text{Error}=E-\tilde{E}=T_1p_1r_1+T_2p_2r_2+\cdots+T_kp_kr_k.$$

Let r_A be a bound on all the r_i's due to addition or subtraction, and let r_M be a bound on all the r_i's due to multiplication. Then

$$\begin{aligned}|E-\tilde{E}|\leqq&[|T_{i_1}|p_{i_1}+|T_{i_2}|p_{i_2}+\cdots+|T_{i_s}|p_{i_s}]r_A\\&+[|T_{j_1}|p_{j_1}+|T_{j_2}|p_{j_2}+\cdots+|T_{j_t}|p_{j_t}]r_M.\end{aligned}$$

Now this bound is the same bound we would have obtained by using r_A for the round-off error in all additions and subtractions and r_M for the error in all multiplications. Furthermore all the terms in E may be included in the coefficients of r_A and r_M by writing the above inequality as

$$|E-\tilde{E}|\leqq\Big(\sum_{i=1}^{k}|T_i|a_i\Big)r_A+\Big(\sum_{i=1}^{k}|T_i|m_i\Big)r_M \tag{2.4}$$

where a_i is either zero or the appropriate p_i and m_i is either zero or the appropriate p_j. With the inequality written this way it is clear that the two strings of nonnegative integers, $(a_1, a_2, \cdots, a_k)$ and $(m_1, \cdots, m_k)$, completely describe the error bound. We shall call these strings, or vectors, the r_A and r_M error vectors, respectively.

These two vectors are determined by the mode of evaluation of the arithmetic expression, except that the order of their components is not uniquely determined. To remedy this situation we shall define an "expression vector", whose components are essentially the terms of the arithmetic expression, and which fixes the order in the

error vectors $(a_1, \cdots, a_k)$ and $(m_1, \cdots, m_k)$. To this end let E be an arithmetic expression and let $\bar{E}$ denote its expression vector. Then $\bar{E}$ can be defined recursively as follows:

(1) If $E = a$, and indeterminate, then $\bar{E} = (|a|)$.

(2) If $E = F \pm G$, and $\bar{F} = (f_1, \cdots, f_l)$, $\bar{G} = (g_1, \cdots, g_k)$, then $\bar{E} = (f_1, \cdots, f_l, g_1, \cdots, g_k)$. We shall write $\bar{E} = \bar{F} + \bar{G}$.

(3) If $E = F * G$ and $\bar{F} = (f_1, f_2, \cdots, f_l)$, then $\bar{E} = (f_1\bar{G}, f_2\bar{G}, \cdots, f_l\bar{G})$, where $f_i\bar{G}$ is usual scalar multiplication of a vector. We shall write $\bar{E} = \bar{F} * \bar{G}$.

For example, if $E = a + b$, $F = c(d - e) + f$, then $\bar{E} = (|a|, |b|)$,

$$\bar{F} = (|cd|, |ce|, |f|), \qquad \bar{E} + \bar{F} = (|a|, |b|, |cd|, |ce|, |f|),$$

$$\bar{E} * \bar{F} = (|acd|, |ace|, |af|, |bcd|, |bce|, |bf|)$$

and

$$\bar{E} * \bar{E} = (|a|^2, |ab|, |ba|, |b|^2).$$

With this notation, we can write some previous equations in a form that will be more useful for our purposes. Since we are interested in analyzing algorithms themselves and not how they perform on data of a specific character, we shall regard the input data as indeterminates. Because of this, we must really do a "worst case" error analysis and our bounds will generally be pessimistic. Now, let $A = (a_1, \cdots, a_k)$ and $M = (m_1, \cdots, m_k)$ be the r_A and r_M error vectors for an expression E. Then the (worst case) absolute error bound, $AE(E)$, is given by the right side of (2.4). In the notation just introduced we have

$$AE(E) = (A \cdot \bar{E})r_A + (M \cdot \bar{E})r_M. \tag{2.5}$$

The worst case error bound in eq. (2.5) is obtained by modifying eqs. (2.1)–(2.3) to:

$$AE(F \pm G) = AE(F) + AE(G) + [\bar{1} \cdot (\bar{F} + \bar{G})]r_A \tag{2.6}$$

$$AE(F * G) = (\bar{1} \cdot \bar{G})AE(F) + (\bar{1} \cdot \bar{F})AE(G) + [\bar{1} \cdot (\bar{F} * \bar{G})]r_M \tag{2.7}$$

where, for scalar c, $\bar{c} = (c, c, \cdots, c)$, a vector of appropriate length.

At this point there is no justification for separating r_A and r_M, but our reasons for this will become clear in the sequel. Also, we will no longer use the bar to distinguish an expression vector from an expression; whether we are considering E to be an expression or its expression vector will be apparent from the context. There are several instances where we refer to (2.5)–(2.7); we will refer to the "unbarred" versions of these equations as (2.5′), (2.6′), and (2.7′).

The error vectors introduced above satisfy several laws of arithmetic which facilitate their computation. To state these rules, let A, M be the r_A, r_M error vectors for the expression F and B, N be those for the expression G. Then the r_A, r_M error vectors for the expression $F \pm G$ will be denoted by $A +_a B$ and $M +_m N$, and are given by

$$A +_a B = (\bar{1} + A, \bar{1} + B) \tag{2.8}$$

$$M +_m N = (M, N). \tag{2.9}$$

Let $A = (a_1, \cdots, a_k)$ and $M = (m_1, \cdots, m_k)$. The r_A, r_M error vectors for $F * G$ will be denoted by $A *_a B$ and $M *_m N$, and are given by

$$A *_a B = (\bar{a}_1 + B, \bar{a}_2 + B, \cdots, \bar{a}_k + B) \tag{2.10}$$

$$M *_m N = (\bar{1} + \bar{m}_1 + N, \bar{1} + \bar{m}_2 + N, \cdots, \bar{1} + \bar{m}_k + N). \tag{2.11}$$

These rules are derived in the Appendix and form the basis of our error model. They provide a simple means to quickly obtain error bounds for arithmetic expressions. The procedure involved is much simpler than either process graphs [2] or Wilkinson's forward error analysis [11], and yet produces the same error bounds for division-free expressions. For example, consider the expression

$$E = (x_1^2 + x_2^2 + x_3^2 + x_4^2)(y_1 z_1 + y_2 z_2). \tag{2.12}$$

For convenience, we will assume y_1, y_2, z_1, z_2 are positive. A computational tree will be convenient to display the vectors and is shown in Fig. 1. The r_M vector is shown in brackets [ ], and the r_A vector in parentheses ( ). The expression vector $\bar{E}$ is

$$\bar{E} = (x_1^2 y_1 z_1, x_1^2 y_2 z_2, x_2^2 y_1 z_1, x_2^2 y_2 z_2, x_3^2 y_1 z_1, x_3^2 y_2 z_2, x_4^2 y_1 z_1, x_4^2 y_2 z_2).$$

From Fig. 1,

$$A = (4, 4, 4, 4, 3, 3, 2, 2), \qquad M = \bar{3}.$$

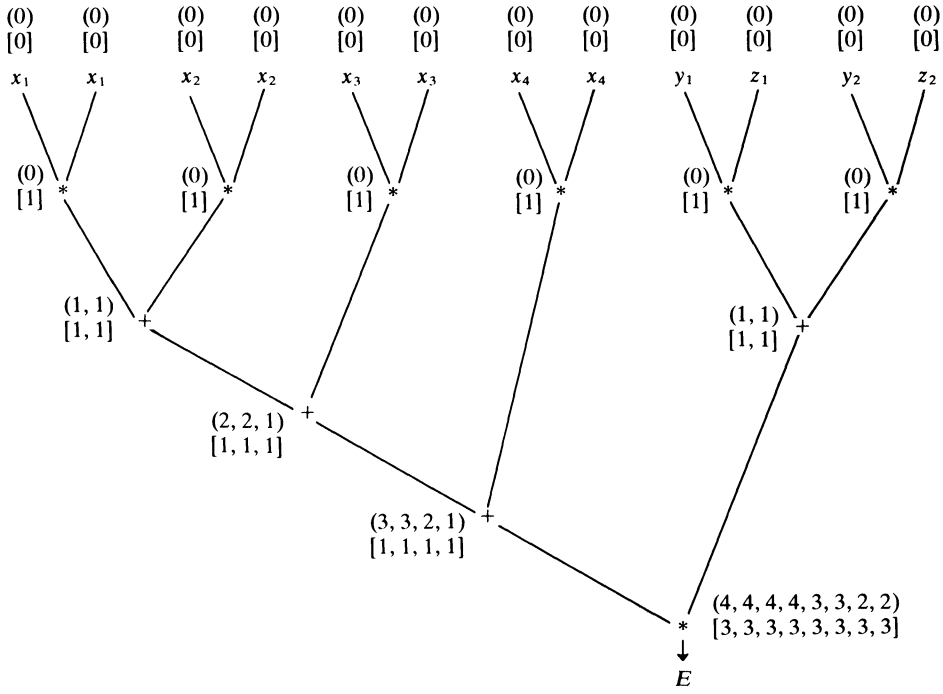


FIG. 1. *A computational tree for expression* (2.12) *showing* r_A *vectors,* ( ), *and* r_M *vectors* [ ].

This example was chosen since the left factor in E is examined in [2] using process graphs and the right factor is done in [11]. Both analyses are tedious and involved, requiring several pages of equations. With the present approach even the combined expression is handled easily and quickly.

3. Sum and product of *n* numbers. We now briefly study the error accumulated in the evaluation of a sum of numbers. Since any expression is equivalent to an expression consisting of a sum of terms, we will use these results in § 5. Let $S = \sum a_i$, where each a_i is an indeterminate. Since the r_M error vector for S is clearly zero, the absolute error bound, $AE(S)$, is given by

$$|S - \tilde{S}| \leqq AE(S) = (A \cdot S) r_A$$

where $\tilde{S}$ is the computed value for S and A is the r_A error vector depending on the evaluation mode for S. For the sequential-sum algorithm

$$S = (\cdots((a_1 + a_2) + a_3) + \cdots + a_n)$$

the r_A error vector is $A_S = (n-1, n-1, n-2, \cdots, 2, 1)$. For the parallel-sum algorithm

$$S = (\cdots((a_1 + a_2) + (a_3 + a_4)) + \cdots + (a_{n-1} + a_n)\cdots)$$

with $n = 2^k$, the r_A error vector is $A_P = \bar{k}$. These r_A error vectors can be obtained easily from the error vector arithmetic. To compare the different evaluation modes for S, we shall use the "sum norm" on the respective r_A error vectors. Hence the best computational scheme is the one with the least sum norm for the error vector A, where the sum norm of $A = (\sigma_1, \sigma_2, \cdots, \sigma_n)$ is defined by

$$\|A\| = \sum_{i=1}^{n} \sigma_i.$$

The next theorem relates the vector A to the associated binary tree given by the particular evaluation mode for the sum.

THEOREM 1. *Let S be the sum of n variables $a_1, \cdots, a_n$. Then, for any evaluation mode of S (given by an associated binary tree, T), the absolute error bound is*

$$AE(S) = r_A \sum_{i=1}^{n} |a_i| w_i(a_i)$$

where $w(a_i)$ is the number of edges in the unique path from the leaf a_i to the root.

Proof. It suffices to show that the r_A error vector E_T is given by $(w(a_1), w(a_2), \cdots, w(a_n))$. When $n = 2$, the error vector is $(1, 1)$. For $n \geqq 2$, let $T = L \wedge R$ where L and R are respectively left and right binary subtrees with E_L and E_R as the r_A error vectors associated with the roots of L and R, respectively. Then using (2.8), we have

$$E_T = (\bar{1} + E_L, \bar{1} + E_R).$$

By induction, in the error vector E_L, the coefficient corresponding to the variable a_i in the subtree L is the index of the node a_i in L. Clearly the index of the node a_i in T equals one plus the index of a_i of L. Similar results hold true for the right subtree R. Since $\{a_1, \cdots, a_n\}$ is the disjoint union of the variables in L and those in R, we get

$$E_T = (w(a_1), \cdots, w(a_n)).$$

It is easily shown that for a binary tree with n leaves, the sum $\sum_{j=1}^{n} w(a_j)$ is minimized by an almost balanced tree. Thus consider the *extended parallel sum algorithm* for $n = 2^k + i$, $0 \leqq i < 2^k$; 1) add $a_1, a_2, \cdots, a_{2i}$ in parallel yielding i sums; 2) add these in parallel with the remaining $2^k - i$ terms. For this algorithm the r_A error vector is $(k+1, k+1, \cdots, k+1, k, k, \cdots, k)$ and this is the best computational scheme to add n numbers.

The extended parallel sum algorithm or any scheme to minimize the norm of the r_A error vector is closely related to Huffman coding [3]. In fact given $a_1, a_2, \cdots, a_n$ all positive and with potentially different magnitudes we can form the sum with minimal error bound by adding quantities as if the variables were leaves on a Huffman tree whose path lengths are obtained from the magnitude of the variables. For positive and negative variables, the positive and negative items may be summed separately with the use of two such Huffman trees and then these results summed.

Since an arithmetic expression is equivalent to an expression involving a sum of terms and a term is the product of variables, we consider ways to compute products. The following theorem is easily established using binary trees and the rule for $*_m$ given in (2.11).

THEOREM 2. *Let P be the product of n indeterminates $a_1, \cdots, a_n$. Then for any evaluation mode for P requiring $n-1$ multiplications, the absolute error bound is $AE(P) = (n-1)|P|r_M$.*

This result shows that all evaluation modes to compute the product of n numbers are errorwise equivalent.

4. Error complexity. The sum and product of n numbers indicate that there is a certain minimum error incurred in every computational scheme. The minimum error inherent in a given arithmetic expression is defined in the next paragraph. For an arithmetic expression E, let Γ consist of all arithmetic expressions obtained from E by using the associative, commutative, and distributive laws. Furthermore, with each member F of Γ let there be associated a well defined mode of evaluation. For example, for the expression $E = a+b+cd$, the expressions $F_1 = (a+b)+cd$, $F_2 = a+(b+dc)$, and $F_3 = (a+cd)+b$ are equivalent to E and there is an unambiguous mode of evaluation for each F_i. Thus F_1, F_2, F_3 belong to Γ but $E \notin \Gamma$. It will be convenient to have E belong to Γ. Without loss of generality, we assume that a left to right evaluation is made for any ambiguous subexpression in E. Thus $E = F_1$ in the above example.

The *error complexity* $EC(E)$ of an arithmetic expression E is defined as the minimum over Γ of the error bounds yielded by members of Γ:

$$EC(E) = \min_{F \in \Gamma} \{(A_F \cdot \bar{1})r_A + (M_F \cdot \bar{1})r_M\}. \tag{4.1}$$

The subscripts on the error vectors A_F and M_F are to emphasize that they may differ for the different members of Γ depending on the mode of evaluation. The results of § 3 show that

$$EC\left(\sum_{j=1}^{n} a_j\right) = (nk+2i)r_A, \qquad n = 2^k + i,$$

and

$$EC\left(\prod_{j=1}^{n} a_j\right) = (n-1)r_M.$$

This notion of error complexity is used to compare restructuring algorithms for arithmetic expressions. A restructuring algorithm is said to be a *minimal error* algorithm α if for every arithmetic expression E, the resultant equivalent form $\alpha(E)$ satisfies the error complexity of E. In the next section we prove some properties of r_A and r_M error vectors that result in the discovery of a minimal error algorithm.

5. Invariance of errors under restructuring. We now turn our attention to possible changes in error bounds when restructuring algorithms are used. Typically restructuring of an arithmetic expression is done by using the associative, commutative, and distributive laws. The next four theorems indicate the extent to which restructuring can alter the error bounds for arithmetic expressions.

THEOREM 3. *The use of the distributive law in a restructuring algorithm for an arithmetic expression does not change the round-off error bounds for the resultant expression.*

Proof. At any stage of a restructuring algorithm the distributive law changes the mode of evaluation of an arithmetic expression by introducing

$$E * (F+G) = (E * F)+(E * G)$$

where E, F, G are the subexpressions at that stage. We shall show that

$$AE(E * (F+G)) = AE(E * F + E * G) \tag{5.1}$$

to prove that the error bounds do not change. We first verify that the r_A contribution to both sides of (5.1) is the same. Let A, B, C be the r_A error vectors for E, F, G. Then we must show that

$$[A *_a (B +_a C)] \cdot [E * (F+G)] = [(A *_a B)+_a(A *_a C)] \cdot [(E * F)+(E * G)]. \tag{5.2}$$

Some identities will shorten the calculations:

$$(A +_a B)\cdot(E+F) = (\bar{1}\cdot E)+(\bar{1}\cdot F)+(A\cdot E)+(B\cdot F),$$

$$(A *_a B)\cdot(E * F) = (A\cdot E)(\bar{1}\cdot F)+(B\cdot F)(\bar{1}\cdot E),$$

and

$$\bar{1}\cdot(E * F) = (\bar{1}\cdot E)(\bar{1}\cdot F).$$

The first follows from (2.8) and the definition of the inner product, the second follows from the lemma in the Appendix and the definition of $*_a$, and the third is a vector version of multiplication of expressions. Let LHS and RHS denote the left and right side of (5.2). Also let $H = F+G$ and $D = B +_a C$. Then using the three identities,

$$\begin{aligned}
\text{LHS} &= (A *_a D)\cdot(E * H)\\
&= (A\cdot E)(\bar{1}\cdot H)+(D\cdot H)(\bar{1}\cdot E)\\
&= (A\cdot E)(\bar{1}\cdot(F+G))+[(B +_a C)\cdot(F+G)](\bar{1}\cdot E)\\
&= (A\cdot E)(\bar{1}\cdot F)+(A\cdot E)(\bar{1}\cdot G)\\
&\quad +[(\bar{1}\cdot F)+(\bar{1}\cdot G)+(B\cdot F)+(C\cdot G)](\bar{1}\cdot E).
\end{aligned}$$

Next let $R = A *_a B$, $S = A *_a C$, $Y = E * F$, $Z = E * G$. Then again by the identities,

$$\begin{aligned}
\text{RHS} &= (R +_a S)\cdot(Y+Z)\\
&= (\bar{1}\cdot Y)+(\bar{1}\cdot Z)+(R\cdot Y)+(S\cdot Z)\\
&= \bar{1}\cdot(E * F)+\bar{1}\cdot(E * G)+(A *_a B)\cdot(E * F)+(A *_a C)\cdot(E * G)\\
&= (\bar{1}\cdot E)(\bar{1}\cdot F)+(\bar{1}\cdot E)(\bar{1}\cdot G)+(A\cdot E)(\bar{1}\cdot F)\\
&\quad +(B\cdot F)(\bar{1}\cdot E)+(A\cdot E)(\bar{1}\cdot G)+(C\cdot G)(\bar{1}\cdot E).
\end{aligned}$$

This is just a rearrangement of the terms appearing in LHS; hence LHS = RHS, proving the invariance of the r_A contribution to (5.1). The proof for the contribution of r_M is done in a similar way, using the relations

$$(M +_m N)\cdot(E+F) = (M\cdot E)+(N\cdot F)$$

$$(M *_m N)\cdot(E * F) = (\bar{1}\cdot E)(\bar{1}\cdot F)+(\bar{1}\cdot E)(N\cdot F)+(\bar{1}\cdot F)(M\cdot E).$$

These results follow from the definitions of $+_m$ and $*_m$. This completes the proof.

Using the identities given above the following results may be proved.

THEOREM 4. *The use of the commutative law in a restructuring algorithm for an arithmetic expression does not change the round-off error bounds for the expression.*

This may be phrased in terms of error vectors as follows.

$$(A +_a B)\cdot(E+F) = (B +_a A)\cdot(F+E),$$

$$(M +_m N)\cdot(E+F) = (N +_m M)\cdot(F+E),$$

$$(A *_a B)\cdot(E*F) = (B *_a A)\cdot(F*E),$$

$$(M *_m N)\cdot(E*F) = (N *_m M)\cdot(F*E).$$

THEOREM 5. *The use of the multiplicative associative law in a restructuring algorithm for an arithmetic expression does not change the round-off error bounds for the expression.*

In terms of error vectors this may be stated in the following way.

$$[A *_a (B *_a C)]\cdot[E*(F*G)] = [(A *_a B)*_a C]\cdot[(E*F)*G],$$

$$[M *_m (N *_m P)]\cdot[E*(F*G)] = [(M *_m N)*_m P]\cdot[(E*F)*G].$$

THEOREM 6. *The use of the additive associative law in a restructuring algorithm for an arithmetic expression does not change the contribution of multiplications to the round-off error bounds for the expression.*

In error vector notation:

$$[M +_m (N +_m P)]\cdot[E+(F+G)] = [(M +_m N)+_m P]\cdot[(E+F)+G].$$

Theorems 3 to 6 lead to the following corollaries.

COROLLARY 1. *Let* Γ *be the class of equivalent forms of an arithmetic expression* E. *Then, for any member* F *of* Γ

$$M_F\cdot F = M_E\cdot E$$

where M_F *and* M_E *are the* r_M *error vectors of* F *and* E, *respectively.*

COROLLARY 2. *Let* F *be an equivalent form of an arithmetic expression* E *obtained by using any laws except additive associativity. Then*

$$A_F\cdot F = A_E\cdot E$$

where A_F *and* A_E *are the* r_A *error vectors of* F *and* E, *respectively.*

A restructuring algorithm uses the five laws of additive and multiplicative commutativity, additive and multiplicative associativity and distributivity to output an equivalent form of an expression suitable for a given computing architecture. The comparison of the absolute error bound for these equivalent forms requires the derivation of their respective r_A and r_M error vectors. The r_M error contribution, $(M\cdot E)r_M$, remains invariant under the use of all five laws; the r_A error contribution, $(A\cdot E)r_A$, is changed only by the use of additive associativity.

These two corollaries lead to a simplification of error complexity EC. Applying Corollary 1 to (4.1) we get

$$EC(E) = r_M(M_E\cdot\bar{1}) + r_A \min_{F\in\Gamma}(A_F\cdot\bar{1}).$$

The second term is proportional to the minimum norm of the r_A error vector. This allows us to seek a restructuring algorithm α which minimizes $\|A_{\alpha(E)}\|$ for all arithmetic expressions E. Such an algorithm is

ALGORITHM α. Let E be any division-free arithmetic expression.

1. Use the distributive law repeatedly to obtain an expression E' equivalent to E and consisting of a sum of products.

2. Use the extended parallel sum algorithm on E', yielding the expression $\alpha(E)$. For example, if $E = a(b+c+d)+e(f+g)$, then $\alpha(E) = ((ab+ac)+ad)+(ef+eg)$.

THEOREM 7. *For any division-free expression E, let Γ be the class of expressions equivalent to E. Then*

$$\|A_{\alpha(E)}\| = \min_{F \in \Gamma} \|A_F\|.$$

Proof. Among all possible modes of evaluating E there is one, say F, for which $\|A_F\|$ is a minimum. In applying step 1 of α to F, assume that parentheses are inserted so that no use is made of the additive associative law. If F' denotes this altered form of F at this stage, then Theorem 3 says that $\|A_F\| = \|A_{F'}\|$. Step 2 of α is an application of the results of § 3, and there we obtained (in present notation) that $\|A_{\alpha(F)}\| \leqq \|A_{F'}\|$. Since $\|A_{F'}\|$ is minimal, $\|A_{\alpha(F)}\|$ is also minimal. Now the terms of $\alpha(E)$ are just a permutation of the terms of $\alpha(F)$ and hence the same is true of $A_{\alpha(E)}$ and $A_{\alpha(F)}$. Thus $\|A_{\alpha(E)}\| = \|A_{\alpha(F)}\| = \|A_F\|$, yielding that $\|A_{\alpha(E)}\|$ is minimal.

6. Two examples

A. We give an example to show that there are arithmetic expressions for which the equivalent form that minimizes the tree height does not yield the minimum error bound. Let

$$E = a_1 + a_2 + a_3a_4 + a_5a_6a_7a_8 + a_9 \cdots a_{16} + \cdots + a_{65} \cdots a_{128}.$$

There are 8 terms in the expression E and a minimum tree height computation of E, with height 7, is given by the equivalent form

$$E' = (\cdots [\{(a_1+a_2)+(a_3a_4)\}+((a_5a_6)(a_7a_8))] + \cdots).$$

Another equivalent form E'' of E is

$$E'' = (\cdots \{(a_1+a_2)+(a_3a_4+(a_5a_6)(a_7a_8))\} + \cdots)$$

where the 8 terms are calculated first and then added in a parallel manner.

The corresponding r_A error vectors A' and A'' are

$$A' = (7, 7, 6, 5, 4, 3, 2, 1) \quad \text{and} \quad A'' = \bar{3}.$$

(The r_M error vectors for E' and E'' are the same.) Thus A' does *not* yield the minimum error bound since $\|A'\|$ grows as $O(n^2)$ ($n = 8$ here) whereas $\|A''\|$ grows only as $O(n \log_2 n)$.

B. The second example concerns the expression

$$E_1 = \left(\sum_{i=1}^{n} U_i\right)\left(\sum_{j=1}^{m} V_j\right).$$

We evaluate E_1 by using the extended parallel sum algorithm (EPSA) on each factor and then multiply. This requires $(n+m-2)$ additions and one multiplication. An equivalent form using *only* the distributive law is

$$E_2 = \sum_{i=1}^{n} \left(\sum_{j=1}^{m} U_iV_j\right),$$

where EPSA is used on each inner sum and then again on the outer sum. This version uses $mn-1$ additions and mn multiplications. A third equivalent form, using additive associativity on E_2, is

$$E_3 = U_1V_1 + U_1V_2 + U_1V_3 + \cdots,$$

in which the terms are added in a left to right order. E_3 requires the same number of operations as E_2.

By our results E_1 and E_2 satisfy the same error bound, but the error bound for E_3 is considerably larger. This was tested with a variety of U_i, V_j, m and n. In all cases E_1 and E_2 agreed to six significant digits, while E_3 differed in the third place. (Computations were done on a 6 digit base 16 computer: $r_A \approx 9 \times 10^{-7}$.) For example, $m = n = 100$, $|U_i|$ chosen randomly in $[0, 2]$, $|V_j|$ chosen randomly in $[0.5, 1.5]$ and both with random signs produced:

$$E_1 = -.317816, \quad E_2 = -.317813, \quad E_3 = -.322394.$$

Note that E_1 has 198 adds and 1 multiply, while E_2 and E_3 have 9,999 adds and 10,000 multiplies!

7. Concluding remarks. Restructuring algorithms produce different equivalent forms of an arithmetic expression and we have raised the question of round-off error propagation and accumulation in the evaluation of these equivalent forms. In order to compare the absolute error bound for different evaluating modes, we have assumed that no error exists in data or inputs. Furthermore, in order to avoid considering any specific characteristic of the data or making assumptions about the relative magnitudes of inputs, we have done a worst case error analysis of restructuring algorithms.

We have shown that for division-free arithmetic expressions the absolute error bound can be broken up into two parts, one due to additions and subtractions and the other due to multiplications. These two parts are completely determined by the r_A and r_M error vectors and the expression vector. Different modes of evaluating an arithmetic expression change the r_A and r_M error vectors. Since associative, commutative, and distributive laws form the essential part of a restructuring algorithm, we examined the invariance properties of r_A and r_M error vectors under these transformations. This enabled us to determine the extent to which these error vectors are changed by a different evaluation mode.

For any numerical problem there are different ways of computing the results. We defined the error complexity to be the minimum error inherent in the problem, irrespective of the evaluation mode, thus leading to the concept of minimal error restructuring algorithms. Such an algorithm has been presented; it transforms every division-free arithmetic expression into a form whose error bound is the error complexity of the original expression.

In view of the invariance of the error bound under most transformations used in a restructuring algorithm, it is clear there are many modes of evaluation which will yield the error complexity. It is an open problem to find a restructuring algorithm whose error bound is the error complexity for the expression, and which minimizes computation time, or storage or some other quantity.

Appendix. We now outline the proof of the addition and multiplication rules for r_A and r_M error vectors given in (2.8)–(2.11). The following lemma is used in the proof.

LEMMA. *Let* $V = (v_1, v_2, \cdots, v_k)$, $E = (e_1, \cdots, e_k)$, $W = (w_1, \cdots, w_l)$ *and* $F = (f_1, \cdots, f_l)$ *be arbitrary vectors. Then*

$$\begin{aligned}(\bar{1} \cdot F)(V \cdot E) + (\bar{1} \cdot E)(W \cdot F) &= (\bar{v}_1 + W, \bar{v}_2 + W, \cdots, \bar{v}_k + W) \cdot (e_1 F, \cdots, e_k F) \\ &= (\bar{w}_1 + V, \bar{w}_2 + V, \cdots, \bar{w}_l + V) \cdot (f_1 E, f_2 E, \cdots, f_l E).\end{aligned}$$

Proof.

$$(\bar{1}\cdot F)(V\cdot E)+(\bar{1}\cdot E)(W\cdot F)=(f_1+f_2+\cdots+f_l)(v_1e_1+\cdots+v_ke_k)$$
$$+(e_1+e_2+\cdots+e_k)(W\cdot F)$$
$$=e_1(v_1f_1+v_1f_2+\cdots+v_1f_l)+e_1(W\cdot F)+\cdots+e_k(v_kf_1+v_kf_2+\cdots+v_kf_l)$$
$$+e_k(W\cdot F)$$
$$=e_1(\bar{v}_1\cdot F)+e_1(W\cdot F)+\cdots+e_k(\bar{v}_k\cdot F)+e_k(W\cdot F)$$
$$=(\bar{v}_1+W)\cdot e_1F+(\bar{v}_2+W)\cdot e_2F+\cdots+(\bar{v}_k+W)\cdot e_kF$$
$$=(\bar{v}_1+W,\bar{v}_2+W,\cdots,\bar{v}_k+W)\cdot(e_1F,e_2F,\cdots,e_kF)$$

and similarly the second part of the equality in the lemma can be proved.

THEOREM A.1. *Let E, F and G be expressions such that $G=E\pm F$. If A, B and C are the r_A error vectors for E, F and G respectively, then*

$$C=(\bar{1}+A,\bar{1}+B).$$

Similarly if M, N and P are the r_M error vectors for E, F and G respectively, then

$$P=(M,N).$$

Proof. Rewriting equation (2.6) for $G=E\pm F$, we have

$$AE(G)=AE(F)+AE(F)+[\bar{1}\cdot(\bar{E}+\bar{F})]r_A.$$

Substituting the expression for the absolute error bounds $AE(E)$, $AE(F)$, and $AE(G)$ given by (2.5′), we get

$$(C\cdot\bar{G})r_A+(P\cdot\bar{G})r_M=(A\cdot\bar{E})r_A+(M\cdot\bar{E})r_M+(B\cdot\bar{F})r_A+(N\cdot\bar{F})r_M+(\bar{1}\cdot\bar{G})r_A.$$

Equating the coefficients of r_A yields

$$(C\cdot\bar{G})=(A\cdot\bar{E})+(B\cdot\bar{F})+\bar{1}\cdot(\bar{E},\bar{F})$$

and hence $C=(\bar{1}+A,\bar{1}+B)$. Equating the coefficients of r_M yields

$$P\cdot\bar{G}=(M\cdot\bar{E})+(N\cdot\bar{F})$$

so $P=(M,N)$.

THEOREM A.2. *Let E and F be arithmetic expressions with A and B as r_A error vectors and M and N as r_M error vectors, respectively. Then*

$$A*_aB=(\bar{a}_1+B,\bar{a}_2+B,\cdots,\bar{a}_k+B)$$
$$M*_mN=(\bar{1}+\bar{m}_1+N,\bar{1}+\bar{m}_2+N,\cdots,\bar{1}+\bar{m}_k+N).$$

Proof. Let $H=E*F$ and $AE(H)=(D\cdot\bar{H})r_A+(Q\cdot\bar{H})r_M$. We wish to relate D to A, B and Q to M, N. Since

$$AE(H)=(\bar{1}\cdot\bar{F})AE(E)+(\bar{1}\cdot\bar{E})AE(F)+(\bar{1}\cdot\bar{H})r_M,$$

using (2.7) for $H=E*F$, we get

$$(D\cdot\bar{H})r_A+(Q\cdot\bar{H})r_M=(\bar{1}\cdot\bar{F})[(A\cdot\bar{E})r_A+(M\cdot\bar{E})r_M]$$
$$+(\bar{1}\cdot\bar{E})[(B\cdot\bar{F})r_A+(N\cdot\bar{F})r_M]+\bar{1}\cdot(e_1\bar{F},e_2\bar{F},\cdots,e_k\bar{F})r_M \tag{A.1}$$

where $\bar{E}=(e_1,\cdots,e_k)$ as usual.

Since $H=(e_1\bar{F},e_2\bar{F},\cdots,e_k\bar{F})$ write $D=(D_1,D_2,\cdots,D_k)$, where each D_i is a

vector of length l—the length of F. Equating the coefficients of r_A in (A.1) and using the lemma, we obtain

$$(D_1, D_2, \cdots, D_k)\cdot(e_1F, e_2F, \cdots, e_kF) = (\bar{a}_1+B, \bar{a}_2+B, \cdots, \bar{a}_k+B)\cdot(e_1F, e_2F, \cdots, e_kF).$$

Consequently, $D_i = \bar{a}_i + B$, or $D = (\bar{a}_1+B, \bar{a}_2+B, \cdots, \bar{a}_k+B)$. Equating the coefficients of r_A in (A.1) yields

$$Q\cdot H = (\bar{1}\cdot F)(M\cdot E)+(\bar{1}\cdot E)(N\cdot F)+\bar{1}\cdot(e_1F, e_2F, \cdots, e_kF).$$

Again using the lemma and writing $Q = (Q_1, Q_2, \cdots, Q_k)$, each Q_i a vector of length l, produces

$$(Q_1, \cdots, Q_k)\cdot(e_1F, \cdots, e_kF) = (\bar{m}_1+N, \cdots, \bar{m}_k+N)\cdot(e_1F, \cdots, e_kF) + \bar{1}\cdot(e_1F, \cdots, e_kF).$$

Hence $Q_i = \bar{1}+\bar{m}_i+N$, or

$$Q = (\bar{1}+\bar{m}_1+N, \cdots, \bar{1}+\bar{m}_k+N).$$

We now give some properties of the r_A and r_M error vectors which are easily established.

While the operations $+_a$, $+_m$, $*_a$, $*_m$ are not commutative since the order of the components is important, the following results are true.

THEOREM A.3: (a) *For r_A error vectors A, B, C we have*

(i) $(A *_a B)*_a C = A *_a (B *_a C)$

(ii) $A *_a (B +_a C) = (A *_a B) +_a (A *_a C)$.

(b) *For r_M error vectors M, N, P we have*

(iii) $(M *_m N)*_m P = M *_m (N *_m P)$

(iv) $(M +_m N)+_m P = M +_m (N +_m P)$

(v) $M *_m (N +_m P) = (M *_m N)+_m (M *_m P)$.

Note that $+_a$ is *not* associative.

Acknowledgment. We would like to thank the referee for many valuable comments including the connection with Huffman coding.

REFERENCES

[1] R. P. BRENT, *The parallel evaluation of arithmetic expressions in logarithmic time*, Complexity of Sequential and Parallel Numerical Algorithms, J. F. Traub, ed., Academic Press, New York, 1973, pp. 83–102.

[2] W. S. DORN AND D. MCCRAKEN, *Numerical Methods with Fortran IV Case Studies*, John Wiley, New York, 1972.

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

[4] P. M. KOGGE, *The numerical stability of parallel algorithms for solving recurrence problems*, Digital Systems Lab., Stanford Univ., Stanford, CA, Sept. 1972.

[5] ———, *Parallel solution of recurrence problems*, IBM J. Res. Develop., 18 (1974), pp. 138–148.

[6] D. J. KUCK AND K. MARUYAMA, *Time bounds on the parallel evaluation of arithmetic expressions*, this Journal, 4 (1975), pp. 147–162.

[7] P. LINZ, *Accurate floating-point summation*, Comm. ACM, 13 (1970), pp. 361–362.

[8] D. E. MULLER AND F. P. PREPARATA, *Restructuring of arithmetic expressions for parallel evaluation*, TR 676, Coordinated Science Lab., Univ. of Illinois, Urbana, 1975.

[9] H. S. STONE, *An efficient parallel algorithm for the solution of a tridiagonal linear system of equations*, J. Assoc. Comput. Mach., 20 (1973), pp. 27–38.

[10] ———, *Problems of parallel computation, Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub, ed., Academic Press, New York, 1973, pp. 1–16.
[11] J. H. WILKINSON, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1963.
[12] S. WINOGRAD, *On the parallel evaluation of certain arithmetic expressions*, J. Assoc. Comput. Mach., 22 (1975), pp. 477–492.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0006 $01.00/0

GENERATING TREES AND OTHER COMBINATORIAL OBJECTS LEXICOGRAPHICALLY*

S. ZAKS† AND D. RICHARDS†

Abstract. We show a one-to-one correspondence between all the ordered trees that have n_0+1 leaves and n_i internal nodes with k_i sons each, for $i=1,\cdots,t$, (hence $n_0=\sum_1^t (k_i-1)n_i$) and all the lattice paths in the $(t+1)$-dimensional space, from the point $(n_0, n_1, \cdots, n_t)$ to the origin, which do not go below the hyperplane $x_0=\sum_1^t (k_i-1)x_i$. Procedures for generating these paths (and thus the ordered trees) are presented and the ranking and unranking procedures are derived.

Key words. ordered tree, lattice path, lexicographic order, ranking, unranking

1. Introduction. Motivated by the problem of generating, ranking and unranking all k-ary trees with n nodes, we solved the analogous problem for all trees with n_i nodes having k_i sons each, $i=1, 2, \cdots, t$, and n_0+1 leaves (hence $n_0=\sum_1^t (k_i-1)n_i$).

We establish a 1–1 correspondence between those trees and the integer sequences $a_1\, a_2 \cdots a_n$, $n=\sum_0^t n_i$, which have n_i occurrences of k_i for $i=1, 2, \cdots, t$, and n_0 0's, such that in each prefix $a_1\, a_2 \cdots a_l$, $1 \leqq l \leqq n$, the number of 0's is not greater than $\sum_1^t (k_i-1)\cdot$(number of k_i's in the prefix).

It turns out that there is also a 1–1 correspondence between these sequences and the lattice paths $L=L_0\, L_1 \cdots L_n$ in the $(t+1)$-dimensional space, from the point $(n_0, n_1, \cdots, n_t)$ to the origin $(0, 0, \cdots, 0)$, which do not go below the hyperplane $x_0=\sum_1^t (k_i-1)x_i$.[1] These correspondences will be shown in § 2.

The algorithm, which lexicographically generates a modified version of the above sequences, is discussed in § 3. The ranking and unranking procedures are the subject of § 4.

2. Trees, sequences and paths.

2.1. Ordering of trees. We will deal solely with *ordered trees* (or *planted planar trees*). We will follow the conventions in [5]. Let $K=(k_0, \cdots, k_t)$ and $N=(n_0, \cdots, n_t)$ be $(t+1)$-tuples of nonnegative integers, such that $k_t > k_{t-1} > \cdots > k_0 = 0$ and $n_0 = \sum_1^t (k_i-1)n_i$. We are concerned with the set of trees $T(K, N)$ where each tree has n_i nodes with k_i sons each for $1 \leqq i \leqq t$, and for convenience, there are n_0+1 nodes with 0 sons, i.e. leaves. If $t=1$, then we have regular[2] k_1-ary trees with n_1 internal nodes.

There are two common ways, found in the current literature, to order k-ary trees which we generalize to the set $T(K, N)$. Let $|T|$ be the number of nodes in tree T, r_T be the degree of its root, and T_i be the subtree rooted at the ith son of the root of T.

The ordering given in [4], [5] for binary trees and in [9] for k-ary trees can be generalized to arbitrary trees as A-order as follows:

* Received by the editors January 18, 1978, and in revised form May 8, 1978.

† Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. This work was supported in part by the National Science Foundation under Grant NSF MCS 73-03408.

[1] We assume that each step in the path is directed towards the origin, and we use this assumption throughout this paper.

[2] In a regular k-ary tree, each internal node has exactly k sons.

Two trees, T and T', are in *A-order*, $T < T'$, if
1) $|T| < |T'|$ or
2) $|T| = |T'|$ and for some i, $1 \leqq i \leqq r_{T'}$ we have
   a) $T_j = T'_j$ for $j = 1, 2, \cdots, i-1$ and
   b) $T_i < T'_i$.

Let p_T be the sequence formed by consecutively numbering the nodes (by traversing T) in post-order and reading them in pre-order. Two trees, T and T', are in A-order if p_T is lexicographically less than $p_{T'}$. For binary trees in-order is interchangeable with post-order and is used in [4]. The proof of this correspondence is analogous to the proof used in the binary case.

A second ordering is given in [15], [16] for k-ary trees which we generalize to arbitrary trees as B-order as follows:

Two trees, T and T', are in *B-order*, $T < T'$, if
1) $r_T < r_{T'}$ or
2) $r_T = r_{T'}$ and for some i, $1 \leqq i \leqq r_T$ we have
   a) $T_j = T'_j$ for $j = 1, 2, \cdots, i-1$ and
   b) $T_i < T'_i$.

Later we give our interpretation of B-order using sequences. The best known algorithms for ranking and unranking trees are considerably more efficient when the trees are in B-order. Therefore, in this paper, we will only be concerned with generation in B-order.

2.2. Tree sequences and lexicographic ordering. Define $A(K, N)$ to be the set of integer sequences $a = a_1 a_2 \cdots a_n$ that have n_i occurrences of the integer k_i and possesses the dominating property. A sequence, a, has the dominating property if the number of 0's is not greater than $\sum_1^l (k_i - 1) \cdot$ (number of k_i's) for every prefix $a_1 a_2 \cdots a_l$, $1 \leqq l \leqq n$. The following theorem was proved in [1] (using the reverses of our sequences).

THEOREM 1. *There is a* 1–1 *correspondence between* $T(K, N)$ *and* $A(K, N)$.

This correspondence is simple to understand and use. Given a tree, T, construct the sequence a_T by labeling each node with its number of sons and reading the labels in pre-order. The last node is not read since it is always a leaf, and its omission simplifies matters. This maps a tree to a sequence. The inverse mapping is accomplished by building a tree node by node from the sequence a_T. Begin by creating a root with degree a_1 and position a pointer there. In general, process a_i by creating a new son of the node v currently pointed to and move the pointer from v to it. If v has its requisite number of sons, backtrack to v's father.

The dominating criterion arises naturally since a tree in $T(K, N)$ has $\sum_1^t (k_i - 1)n_i + 1$ leaves. If the criterion were violated, it would indicate the existence of a completed tree which does not arise since the final leaf is omitted. The property can be written as $\sum_1^l (a_i - 1) \geqq 0$, since the sum of the negative terms is the number of 0's, or more succinctly as

$$\sum_1^l a_i \geqq l.$$

This has a simple interpretation: there are not more nodes than the collective number of sons.

Pre-order search of trees shares with other search methods the property that it inspects all of a node's ancestors before inspecting that node. This is enough to insure

the sequences read are in $A(K, N)$. Breadth-first search is such a search method and is used in [1], [3]. We will use pre-order exclusively because it simplifies the generation procedure.

Similar sequences have been studied extensively in relationship to the "ballot problem" (for example [14]). They have been related to binary trees in [2] and were independently given for k-ary trees in [3], [15], [16]. An overview of such sequences is found in [6].

It is vital to note that $T < T'$ i.e. in B-order if a_T is lexicographically less than $a_{T'}$. This follows from the definitions of a_T and B-order. Note that r_T and $r_{T'}$ are equal to a_1 and a_1 respectively. Therefore, if $r_T < r_{T'}$, then a_T precedes $a_{T'}$. If $r_T = r_{T'}$ and the first $i-1$ subtrees are equal, then the corresponding prefixes of a_T and $a_{T'}$ are equal, since a_T is formed in a pre-order fashion. Then the argument recurs on T_i and T_i'. Therefore, if we generate the sequences of $A(K, N)$ lexicographically, we will generate the trees of $T(K, N)$ in B-order.

In arranging the sequences of $A(K, N)$ in lexicographic order, as normally defined, only the relative values of the k_i's are needed. Therefore, since $k_{i+1} > k_i$ and $k_0 = 0$, we find it convenient to map the sequences of $A(K, N)$ to sequences $b = b_1\, b_2 \cdots b_n$, where $b_i = j$ if $a_i = k_j$. More formally, b is an element of $B(K, N)$ if it contains n_i occurrences of the integer i and for every subsequence $b_1\, b_2 \cdots b_l$, $\sum_1^l k_{b_j} \geqq l$. There is obviously a one-to-one correspondence between $A(K, N)$ and $B(K, N)$ that preserves the lexicographic ordering.

We note that there is also a correspondence between ordered forests and such sequences. An ordered forest consists of ordered trees which are in turn ordered. Define $F(K, N)$ in the same way as $T(K, N)$ except that $n_0 = \sum_1^t (k_i - 1)n_i + (f-1)$, i.e., $F(K, N) = T(K, N)$ if $f = 1$. $A(K, N)$ is defined analogously. If we introduce a new node v of degree f and connect it to the roots of the f ordered trees, we create one ordered tree. The correspondence is easily seen if we prefix the sequence a, $a \in A(K, N)$, with the integer f to get a sequence corresponding to the tree with root v and note that this sequence now has the dominating property. Our generation and ranking procedures will work identically on sequences from $F(K, N)$ as from $T(K, N)$, but we use $T(K, N)$ in our discussion for clarity.

2.3. Lattice paths. Corresponding to these sequences are lattice paths within a bounded region of $(t+1)$-dimensional space from the point $(n_0, n_1, \cdots, n_t)$ to the origin. Each step of the path is one unit towards the origin parallel to some ith dimensional axis, and the path may not go below the hyperplane $x_0 = \sum_1^t (k_i - 1)x_i$. Let $P(K, N)$ be this set of paths.

In [1] it is shown that there is one-to-one correspondence between $B(K, N)$ and $P(K, N)$. To map a path to a sequence, let b_j be i if the jth step of the path is parallel to the ith dimensional axis. The inverse mapping follows immediately. More formally, an element of $P(K, N)$ is a sequence of lattice points $L_0\, L_1 \cdots L_n$ where $L_0 = (n_0, n_1, \cdots, n_t)$, $L_N = (0, 0, \cdots, 0)$ and if $b_i = j$ and $L_{i-1} = (x_0, x_1, \cdots, x_t)$ then $L_i = (x_0, \cdots, x_{j-1}, x_j - 1, x_{j+1}, \cdots, x_t)$. The relation of ranking and unranking to lattice paths was first discussed in [12] and [13].

Example. As an example, consider the tree in Fig. 1 which is an element of $T(K, N)$, where $K = (0, 2, 3)$ and $N = (4, 2, 1)$. The corresponding elements from $A(K, N)$, $B(K, N)$ and $P(K, N)$ are given.

3. Generation of trees. In the preceding section we described a mapping from $B(K, N)$ to $A(K, N)$ and from $A(K, N)$ to $T(K, N)$. We also showed that the lexicographic order in $B(K, N)$ corresponds to B-order in $T(K, N)$. To generate the next

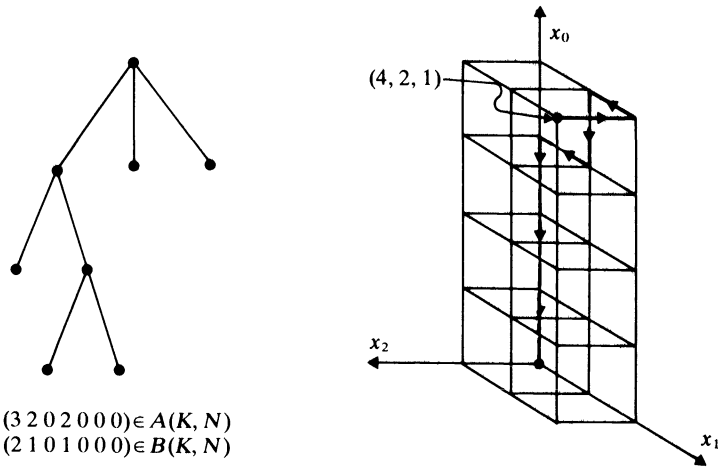


FIG. 1.

tree after a given tree T we produce its corresponding sequence $b \in B(K, N)$ and generate the lexicographically next sequence b' and map it to T', the next tree.

All the sequences of $B(K, N)$ are permutations of each other. If it were not for the condition that each sequence should have the dominating property, it would be a simple matter of generating permutations in lexicographic order which has been well studied [8]. However, we have chosen one such algorithm and adapted it. In its original form, it can be described as follows: scan the permutation $c_1 c_2 \cdots c_n$ of $\{1, 2, \cdots, n\}$, from the right until the first occurrence of $c_i < c_{i+1}$. Substitute c_i with the least c_j such that $c_j > c_i$ and $j > i$ and append after it the first permutation of $\{c_i, c_{i+1}, \cdots, c_n\} - \{c_j\}$ in the ordering. (This is done efficiently by a single exchange and subsequence reversal.)

Similarly, we scan from the right for the first $b_i < b_{i+1}$ and substitute b_i with the appropriate b_j. And again, we append the first permutation of $b^* = \{b_i, b_{i+1}, \cdots, b_n\} - \{b_j\}$. If it were not for the dominating property, the first permutation of b^* would be $0^{m_0} 1^{m_1} 2^{m_2} \cdots t^{m_t}$ where S^x indicates x repetitions of the sequence S, and there are m_i occurrences of i in b^*. Let $d = m_0 - \sum_1^t m_i(k_i - 1)$. The first permutation of b^* is

$$0^d (1\, 0^{k_1 - 1})^{m_1} (2\, 0^{k_2 - 1})^{m_2} \cdots (t\, 0^{k_t - 1})^{m_t}.$$

To show this, note that the dominating property can be rewritten as $\sum_{l+1}^{n} k_{b_i} \leqq n - l$. Therefore if the first permutation of b^* began with $d + 1$ 0's, the property would not hold. The next character must be the smallest nonzero character of b^*; otherwise, some other sequence would precede it. Say it was a 1; then at most $k_1 - 1$ 0's may follow before the property is again violated. The argument recurs, establishing the above permutation. Note that $d > 0$, since the original sequence had the dominating property and $b_i < b_j$, where b_i and b_j were interchanged.

We now state the preceding discussion as an algorithm. Note that **for** termination is checked before loop entry. It is easily seen to have time complexity $O(n)$ where $n = \sum_0^t n_i$.

ALGORITHM GENERATE (b). (This algorithm generates the lexicographically next sequence after the input sequence b.)

```
begin
    for j ← 1 to t do m_j ← 0;
    i ← n; sum ← 0;
    while b_{i-1} ≧ b_i do
        begin m_{b_i} ← m_{b_i} + 1;
            if b_i > 0 then sum ← sum + k_{b_i} − 1;
            i ← i − 1
        end;
    j ← 0; l ← b_{i-1} + 1;
    while j = 0 do
        if m_l > 0 then j ← l
        else l ← l + 1;
    m_{b_{i-1}} ← m_{b_{i-1}} + 1;
    b_{i-1} ← j;
    m_j ← m_j − 1;
    sum ← sum − k_j + 1 + (if b_{i-1} > 0 then k_{b_{i-1}} − 1 else 0);
    m_0 ← m_0 − sum;
    for j ← 0 to t do
        while m_j > 0 do
            begin b_i ← j; m_j ← m_j − 1; i ← i + 1;
                for l ← 1 to k_i − 1 do
                    begin b_i ← 0; i ← i + 1 end
            end
end.
```

This algorithm can be stated more succinctly in the case of k-ary trees, i.e., $t = 1$. In [16] this is done, but the more convenient reverse B-order was used, where the precedence relations are merely reversed. Generation and ranking are both done differently but use sequences from $B(K, N)$.

In [9], [10], k-ary trees are ranked and generated in B-order by use of a mapping between k-ary trees and binary trees. In [7] binary trees are generated and ranked in B-order, and the sequence of the level numbers (i.e. the heights) of the leaves read in in-order is used. The correspondence of these two methods to B-order was established in [15]. Applying the generating algorithm to $B(K, N)$ as in the example in the end of the second section, we get the following sequences:

Index	Sequence	Index	Sequence
1	1010200	12	1210000
2	1012000	13	2001010
3	1020010	14	2001100
4	1020100	15	2010010
5	1021000	16	2010100
6	1100200	17	2011000
7	1102000	18	2100010
8	1120000	19	2100100
9	1200010	20	2101000
10	1200100	21	2110000
11	1201000		

4. Ranking and unranking. In this section we compute the function Index (L) that, given a path $L \in P(K, N)$, will compute its corresponding position in the lexicographic ordering of $P(K, N)$; also, given an integer w, we construct the path $L \in P(K, N)$ such that Index $(L) = w$. As discussed earlier, the paths $L = L_0 L_1 \cdots L_n$ in $P(K, N)$ are those lattice paths from the point $(n_0, n_1, \cdots, n_t)$ to the origin which do not go below the hyperplane $x_0 = \sum_1^t (k_i - 1)x_i$.

We make use of the *multinomial coefficients*

$$\binom{d}{d_1, d_2, \cdots, d_l} = \begin{cases} 0 & \text{if any } d_i \text{ is } <0 \\ \dfrac{d!}{d_1!\, d_2! \cdot \cdots \cdot d_l!} & \text{otherwise} \end{cases}$$

where $d = \sum_1^l d_i$. The multinomial coefficient has a familiar interpretation as the number of lattice paths from the point $(d_1, d_2, \cdots, d_l)$ to the origin. This interpretation gives a combinatorial proof of the following lemma, which is also easily proved directly from the above definition.

LEMMA 1. *If $d = \sum_1^l d_i$ and all d_i are integers, then*

$$\binom{d}{d_1, d_2, \cdots, d_l} = \sum_{i=1}^{l} \binom{d-1}{d_1, d_2, \cdots, d_{i-1}, d_i - 1, d_{i+1}, \cdots, d_l}.$$

Let $C(n_0, n_1, n_2, \cdots, n_t)$ denote the number of lattice paths from the point $(n_0, n_1, \cdots, n_t)$ to the origin which do not go below the hyperplane $x_0 = \sum_1^t (k_i - 1)x_i$. The following theorem defines these entries recursively, and solves the recurrence relation.

THEOREM 2. *The solution to the recurrence relation*

$$C(n_0, n_1, n_2, \cdots, n_t) = \begin{cases} 0 & \text{if } n_i < 0 \\ & \quad \text{for } i = 1, 2, \cdots, \text{or } t, \\ 1 & \text{if } n_0 = n_1 = \cdots = n_t = 0, \\ 0 & \text{if } n_0 = \sum_1^t (k_i - 1)n_i - 1, \\ \sum_{j=0}^{t} C(n_0, n_1, \cdots, n_j - 1, \cdots, n_t) & \text{otherwise} \end{cases}$$

is given by

$$(*)\quad C(n_0, n_1, n_2, \cdots, n_t) = \binom{n}{n_0, n_1, \cdots, n_t} - \sum_{i=1}^{t} (k_i - 1)\binom{n}{n_0 + 1, n_1, \cdots, n_i - 1, \cdots, n_t}$$

where $n = n_0 + n_1 + \cdots + n_t$.

Proof. We show that $C(n_0, n_1, \cdots, n_t)$, as given by $(*)$, satisfies the recurrence relation and the boundary conditions. When $n_i < 0$ for $i = 1, 2, \cdots,$ or t, $(*)$ gives the value 0 by definition. The case $n_0 = n_1 = \cdots = n_t = 0$ is taken care of in the same way. When $n_0 = \sum_1^t (k_i - 1)n_i - 1$ and no n_i is <0, $(*)$ can be rewritten as

$$C(n_0, n_1, n_2, \cdots, n_t) = \frac{n!}{(n_0 + 1)!\, n_1! \cdots n_t!}\left[n_0 + 1 - \sum_1^t (k_i - 1)n_i\right]$$

from which it is clear that $C(n_0, n_1, \cdots, n_t)$ is 0 for this case. If $n_0, n_1, \cdots, n_t$ are none of the above, we prove the recurrence by induction on n. For $n = 1$, $(*)$ is correct. We assume that it holds for any $m < n$, and take $n = n_0 + n_1 + \cdots + n_t$. By the recursive definition of $C(n_0, n_1, \cdots, n_t)$ we have

$$C(n_0, n_1, \cdots, n_t) = \sum_{j=0}^{t} C(n_0, n_1, \cdots, n_j - 1, \cdots, n_t).$$

For each of the terms on the right, we use (*), by the induction hypothesis, and get

$$C(n_0, n_1, \cdots, n_t) = \sum_{j=0}^{t} \binom{n-1}{n_0, n_1, \cdots, n_j - 1, \cdots, n_t}$$

$$- \sum_{j=0}^{t} \sum_{i=1}^{t} (k_i - 1) \binom{n-1}{n_0 + 1, n_1, \cdots, n_i - 1, \cdots, n_j - 1, \cdots, n_t}$$

$$= \sum_{j=0}^{t} \binom{n-1}{n_0, n_1, \cdots, n_j - 1, \cdots, n_t}$$

$$- \sum_{i=1}^{t} (k_i - 1) \sum_{j=0}^{t} \binom{n-1}{n_0 + 1, n_1, \cdots, n_i - 1, \cdots, n_j - 1, \cdots, n_t}$$

which, by the previous lemma, gives

$$C(n_0, n_1, \cdots, n_t) = \binom{n}{n_0, n_1, \cdots, n_t} - \sum_{i=1}^{t} (k_i - 1) \binom{n}{n_0 + 1, n_1, \cdots, n_i - 1, \cdots, n_t}$$

as desired. □

This theorem has been previously solved for the $t = 1$ case: for $k_1 = 2$ it was solved in [11] and a solution for arbitrary k_1 is found in [14] and [10]. A solution for the general problem for points *on* the hyperplane $x_0 = \sum_1^t (k_i - 1)x_i$ is given in [1] using an involved generating function argument.

Given a path $L \in P(K, N)$, we find its position Index (L) in the lexicographic ordering of $P(K, N)$, as follows:

THEOREM 3. *Let* $b = b_1\, b_2 \cdots b_n \in B(K, N)$ *and the corresponding lattice path* $L = L_0\, L_1 \cdots L_n \in P(K, N)$, *where* $L_i = (y_{i0}, y_{i1}, \cdots, y_{it})$. *Then*

$$\text{Index } (L) = 1 + \sum_{i=0}^{n-1} \sum_{j=0}^{b_{i+1}-1} C(y_{i0}, y_{i1}, \cdots, y_{ij} - 1, \cdots, y_{it})$$

where the $C(\cdot, \cdot, \cdots, \cdot)$*'s are given by Theorem* 2.

Proof. By definition we know that all the sequences that begin with either of $0, 1, \cdots$ or $b_1 - 1$ will come before this sequence b, and their number is indicated by the inner summation $\sum_{j=0}^{b_1-1}$. This follows from the definitions of b_i, y_{ij} and Theorem 2.

Next, we know that all the sequences which begin with $b_1 0$, $b_1 1, \cdots$, or $b_1(b_2 - 1)$ will come before b, and their number is indicated by the summation $\sum_{j=0}^{b_2-1}$. The rest follows immediately by induction, following this line of argument.

The constant 1 is added so that the indexing will begin with 1 rather than 0. □

The time complexity depends on how the $C(x_0, \cdots, x_t)$ are calculated. If storage is inexpensive or if ranking is done frequently all the values could be stored in $(t+1)$-dimensional array of space complexity $O(n^*)$ in time proportional to $(t+1)n^*$ where $n^* = \prod_1^t n_i$. Using this array, however, allows each ranking to be done with time complexity $O((t+1)n)$. If ranking is done infrequently the values of the $C(x_0, \cdots, x_t)$ can be calculated as needed in time $O((t+1)n)$. This leads to an $O((t+1)^2 n^2)$ ranking procedure. Note that for most applications $(t+1)$ will be small and independent of n. In [9] a ranking procedure for k-ary trees is given which is $O((nk)^2)$ time-bounded, which is improved to $O(nk)$ in [10] with preprocessing. The best known ranking procedures [9] for k-ary trees in A-order are $O(n^k)$.

To illustrate this procedure refer to the tree and path in our previous example. In Fig. 2 the lattice points have been labeled with $C(x_0, \cdots, x_t)$. The path L corresponding to that tree gets the following rank:

$$\text{Index } (L) \doteq 1 + 0 + 12 + 5 + 0 + 2 + 0 + 0 + 0 = 20.$$

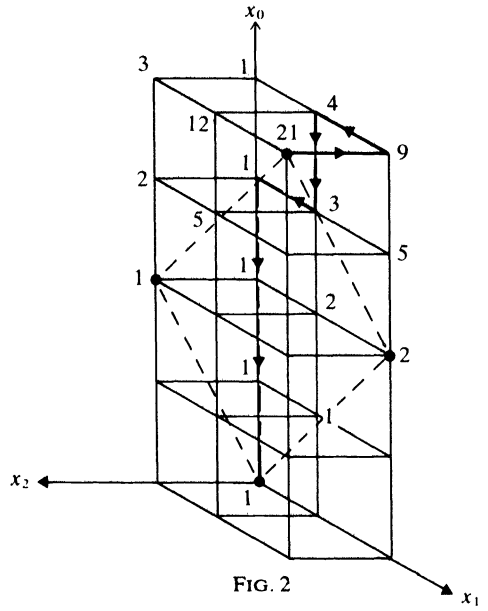


FIG. 2

As for the unranking procedure, we will follow Theorem 3 in a reverse order. We are given a number w, and look for a path L such that Index $(L) = w$. The idea is best explained by an example: suppose we want to find the 20th sequence in $P(K, N)$ where $K = (0, 2, 3)$ and $N = (4, 2, 1)$. Starting at the point $(4, 2, 1)$ we sum up the entries in direction $0, 1, \cdots$ (see Fig. 2) in that order as long as we do not exceed $20 - 1 = 19$. Here we take 12, which corresponds to making the first move from $(4, 3, 1)$ to $(4, 3, 0)$, or $b_1 = 2$. Starting from this point $(4, 3, 0)$, we can sum up the entries in the directions $0, 1, \cdots$ (in that order) as long as we do not exceed $19 - 12 = 7$. Here we take 5, which corresponds to making the next move from $(4, 3, 0)$ to $(3, 3, 0)$, etc. This is given more formally as algorithm UNRANK.

ALGORITHM UNRANK (w). (This algorithm returns the b sequence corresponding to the lattice path L having rank w. The rank of the sequence beginning at L_i is always u.)

```
begin
    u ← w − 1;
    (y0, y1, ···, yt) ← (n0, n1, ···, nt);
    for i ← 1 to n do
        begin
            (Find the largest j such that sum of entries in
            the first j directions does not exceed u)
            j ← 0;
            sum ← 0;
            S ← C(y0 − 1, y1, ···, yt);
            while sum + S ≦ u do
                begin sum ← sum + S; j ← j + 1;
                    S ← C(y0, ···, yj − 1, ···, yt)
                end;
            bi ← j;
            yj ← yj − 1;
            u ← u − sum;
        end
end
```

The proof follows directly from the previous theorem. The space and time complexity considerations are the same as for the ranking procedure.

Acknowledgment. The authors wish to thank Professor C. L. Liu for his valuable assistance and encouragement during this research.

REFERENCES

[1] I. Z. CHORNEYKO AND S. G. MOHANTY, *On the enumeration of certain sets of planted plane trees*, J. Combinatorial Theory Ser. B, 18 (1975), pp. 209–221.

[2] N. G. DEBRUIJN AND B. J. M. MORSELT, *A note on plane trees*, Ibid., 2 (1967), pp. 27–34.

[3] D. A. KLARNER, *Correspondences between plane trees and binary sequences*, Ibid., 9 (1970), pp. 401–411.

[4] G. D. KNOTT, *A numbering system for binary trees*, Comm. ACM, 20 (1977), no. 2.

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

[6] R. C. READ, *The coding of various kinds of unlabeled trees*, Graph Theory and Computing, R. C. Read, ed., Academic Press, 1972, pp. 153–182.

[7] F. RUSKEY AND T. C. HU, *Generating binary trees lexicographically*, this Journal, 6 (1977), pp. 745–758.

[8] R. SEDGEWICK, *Permutation generation methods*, Comput. Surveys, 9 (1977), pp. 152–154.

[9] A. E. TROJANOWSKI, *On the ordering, enumeration and ranking of k-ary trees*, Tech. rep. UIUCDCS-R-77-850, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, February 1977.

[10] ———, *Ranking and listing algorithms for k-ary trees*, this Journal, to appear.

[11] W. A. WHITWORTH, *Arrangements of m things of one sort and m things of another sort under certain conditions of priority*, Messenger of Math, 8 (1878), pp. 105–114.

[12] H. S. WILF, *A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects*, preprint, Univ. of Pennsylvania, Philadelphia.

[13] S. G. WILLIAMSON, *On the ordering, ranking and random generation of basic combinatorial sets*, Proceedings of the Table Rondé, Combinatoire et Représentation du Groupe Symmetrique, Strasbourg, France, 26–30 April 1976, Springer-Verlag Lecture Notes in Mathematics No. 579, Springer-Verlag, Berlin, pp. 309–339.

[14] A. M. YAGLOM AND I. M. YAGLOM, *Challenging Mathematical Problems with Elementary Solutions*, vol. 1: *Combinatorial Analysis and Probability Theory*, Holden-Day, San Francisco, 1964.

[15] S. ZAKS, *Lexicographic generation of ordered trees*, Theor. Comput. Sci., to appear.

[16] ———, *Generating k-ary trees lexicographically*, Tech. rep UIUCDCS-R-77-901, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, November 1977.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0007 $01.00/0

HEURISTICS THAT DYNAMICALLY ORGANIZE DATA STRUCTURES*

JAMES R. BITNER†

Abstract. We first consider heuristics that dynamically alter linked lists, causing more frequently accessed keys to move nearer the "top" of the list. We show that the move to front rule reduces the access time much more quickly than the transposition rule, then give a "hybrid" of these two rules which decreases the access time quickly *and* has low asymptotic cost. We also discuss rules that assume a counter is associated with each key. Second, we consider rules for binary search trees. The monotonic tree rule performs well only when the entropy of the probability distribution for key requests is low; otherwise, it does not reduce the access time. A final class of rules using rotations give nearly optimal performance.

Key words. Data structure, heuristics, self-organizing files, move to front rule, transposition rule, monotonic trees, heaps

1. Introduction. We consider heuristics that reduce access time in linked lists and binary search trees by "orgainizing" the data structure, that is, by moving more frequently accessed keys nearer the "top" of the structure (where searching begins). We assume that keys are requested according to a fixed but *unknown* probability distribution. Thus, the heuristic must *dynamically* alter the data structure as requests are made.

We define the *expected access time* (or *cost*) of a data structure to be $\sum_{i=1}^{n} p_i c_i$ where c_i is the number of comparisons required to locate key k_i, which is requested with probability p_i. The cost can be significantly affected by location of the keys in the data structure. For example, consider a linked list of n elements in which key k_i is requested with probability $p_i = 1/(iH_n)$ where $H_n = \sum_{i=1}^{n} 1/i$ (Zipf's Law). If the order of the keys in the list is random, the expected access time is $(n+1)/2$. However, if the keys are optimally arranged (in order of decreasing probability), the expected access time is only $n/H_n \approx n/(\ln n)$. In fact, the heuristics we consider will perform close to the optimum, so substantial savings will be realized.

Previous work with linked lists [1]–[7] has been concerned with permutation rules. A *permutation rule* [1] for a list of n elements is defined by $\{\tau_i : 1 \leqq i \leqq n\}$, a set of permutations over n elements. When the ith element in the list is requested, permutation τ_i is used to reorder the list. One example is the *move to front rule*, which moves the requested key to the top of the list. All keys which it passes over move down one position. Another example is the *transposition rule*, which transposes the requested key with the one above it. If the requested key is already on top, both rules leave the list unchanged.

Previous results are solely concerned with the asymptotic behavior of these rules. The formula for the asymptotic cost[1] of the move to front rule is given in [2], [5], [6] and has been analyzed by Knuth [6, p. 399] when the request probabilities are given by Zipf's Law. No simple form for the asymptotic cost of the transposition rule has been found. Rivest [1] proved the transposition rule has *lower* (or equal) asymptotic cost than

* Received by the editors April 22, 1977. The major portion of this work was done at the University of Illinois at Urbana-Champaign as the author's Ph.D thesis and was supported in part by the National Science Foundation under Grants GJ-41538 and MCS 77-02705.

† Department of Computer Science, University of Texas, Austin, Texas 78712.

[1] We use the phrase "cost of rule x" to mean "the cost of accessing a data structure being modified by rule x".

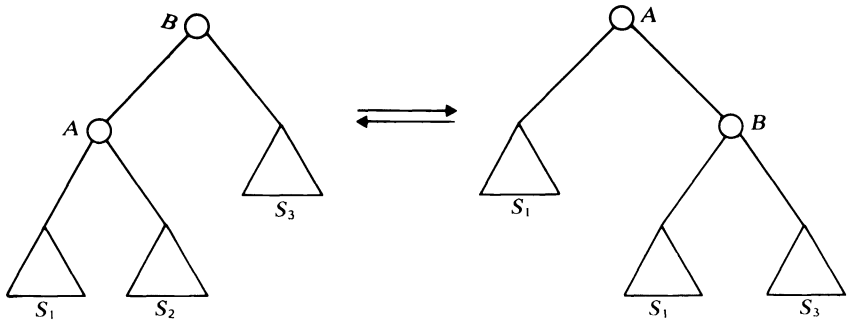


FIG. 1.1. *The two rotations. Here, A and B are nodes.* S_1, S_2 *and* S_3 *are subtrees.*

the move to front rule for *any* probability distribution and conjectured the transposition rule to have lowest asymptotic cost of all the permutation rules for every distribution. Yao (see [7]) has shown that if such an optimal permutation rule *does* exist, it must be the transposition rule.

In previous work concerning binary search trees, rotations (see Fig. 1.1) have been used [7], [10] to define analogues to the transposition and move to front rules for linked lists. The *move up one* rule uses a rotation to move the requested key up one level, and the *move up root rule* successively applies rotations, promoting the requested key until it becomes the root.

Formulas for the asymptotic cost of the move to root rule have been determined [7], [10], the former for an arbitrary initial distribution. Allen and Munro [10] have bounded the cost of the move to root rule, for *any* probability distribution, by essentially (2 ln 2) times the optimal cost. In contrast, they give a distribution for which the move up one rule has cost $\approx \pi\sqrt{n}$. Therefore, the asymptotic cost of the move up one rule *is not* always less than or equal to that of the move to root rule (as opposed to the result [1] that the cost of the transposition rule is always less than or equal to that of the move to front rule).

In our analysis of these heuristics, we assume that any two requests are independent and that the request probabilities do not vary with respect to time. Further, we will not consider the cost of performing the dynamic alteration because the heuristics are simple and cheaply executed. Therefore, this cost should be small compared to the time spent accessing the data structure.

Section 2 will compare how quickly the transposition and move to front rules approach their asymptotic cost and show that for two different measures that the move to front rule converges much more rapidly. Section 3 will discuss the first request rule, which has the same behavior as the move to front rule, and § 4 will use it to form a "hybrid rule" that combines the rapid convergence of the move to front rule with the low asymptotic cost of the transposition rule. Section 5 will consider three rules that use extra storage associated with each key as a counter. The performance of one of these rules, the limited difference rule, is nearly optimal.

The final two sections consider binary search trees where each node has storage associated with it that can be used as a counter. Section 6 considers an intuitively appealing rule, the monotonic tree rule. However, its cost is shown to be asymptotically equal to that of a randomly built tree. Section 7 considers a group of rules that give nearly optimal performance.

We use ln x to denote the natural logarithm of x and log x to denote the base 2 log. Also, the following standard notations are used: $f(n)=O(g(n))$ means $\lim_{n\to\infty} f(n)/g(n)$ is bounded, $f(n)=o(g(n))$ means this limit equals zero, and $f(n)=\Omega(g(n))$ means this limit is bounded, but *not* equal to zero. Finally, a data structure will always have n keys $k_1, \cdots, k_n$, with request probabilities $p_1, \cdots, p_n$ respectively.

2. Rate of convergence of permutation rules. Rivest [1] considered the asymptotic cost of permutation rules and found that the cost of the transposition rule is less than or equal to that of the move to front rule for every probability distribution. This section takes a different point of view; we consider how quickly permutation rules approach their asymptotic cost. This is an important problem, because a rule may have very low asymptotic cost, yet converge so slowly that it is not practically useful. We find that for two different measures of convergence, the move to front rule converges much more quickly than the transposition rule. Initially, it will have lower cost, making it the superior rule if a relatively small number of requests will be made.

The intuitive reason for this is clear. In the initial random ordering, many high probability keys are far down in the list. Using the move to the front rule, these keys rise quickly to the top. In contrast, the transposition rule allows keys to rise only one position per request, and the cost decreases slowly.

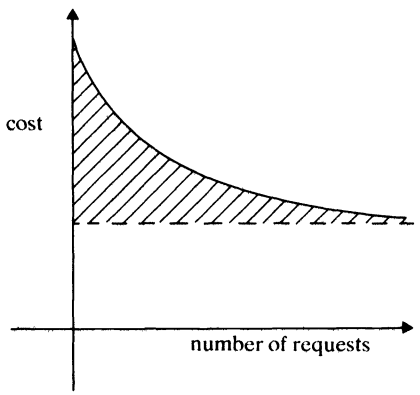


FIG. 2.1. *The overwork is the area between the cost curve and its asymptote (shaded above.)*

The first measure of convergence we consider is the *overwork*, defined as the area between the cost curve and its asymptote (see Fig. 2.1). Note the "steeper" the cost curve is, the smaller the overwork will be. Let OV_{MTF} and OV_{TR} be the overwork of the move to front and transposition rules respectively. To begin the analysis of the overwork, we determine the expected cost of the move to the front rule as a function of time.

THEOREM 2.1 *If each initial list is equally likely, the expected cost of the move to front rule after t requests is*

$$1+2\sum_{1\le i<j\le n}\frac{p_ip_j}{p_i+p_j}+\sum_{1\le i<j\le n}\frac{(p_i-p_j)^2}{2(p_i+p_j)}(1-p_i-p_j)^t.$$

Proof. We first determine a useful form for $E(\text{Cost})$. If l_i is a random variable denoting the level of k_i (equal to the number of comparison required to locate it), we

have

$$E(\text{Cost}) = \sum_{i=1}^{n} p_i E(l_i). \tag{2.1}$$

Introduce random variables A_{ij} for $i \neq j$ where

$$A_{ij} = \begin{cases} 1 & \text{if } k_i \text{ is ahead of } k_j \text{ in the list,} \\ 0 & \text{if not.} \end{cases}$$

Since the level of a key is simply one more than the number of keys ahead of it in the list, $l_i = 1 + \sum_{j \neq i} A_{ji}$. Substituting this into (2.1) and noting $E(A_{ji}) = \text{Prob}\,(k_j \text{ ahead of } k_i)$ gives $\text{E(Cost)} = 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \text{Prob}\,(k_j \text{ ahead of } k_i)$. This formula will be frequently used in the proofs of subsequent theorems.

We now calculate Prob (k_i is ahead of k_j at time t). Two different situations can cause k_i to be ahead of k_j. First, neither k_i nor k_j was requested in t requests and k_i was initially ahead of k_j. The probability of this is $\frac{1}{2}(1-p_i-p_j)^t$. Second, k_i's most recent request was at time $m \geqq 1$, and k_j was not requested *after* time m. The probability for this is

$$\sum_{m=1}^{t} (1-p_i-p_j)^{t-m} p_i = \left(\frac{p_i}{p_i+p_j}\right) - (1-p_i-p_j)^t \left(\frac{p_i}{p_i+p_j}\right).$$

Adding these gives Prob (k_i ahead of k_j at time t), and substituting into the expression for E(Cost) proves the theorem. □

As $t \to \infty$ the last term vanishes, and the first terms give the steady state cost (see [2], [5], [6]). The last term then measures the speed of convergence. From this formula, we can calculate the overwork for the move to front rule.

THEOREM 2.2.

$$\text{OV}_{\text{MTF}} = \sum_{1 \leqq i < j \leqq n} \frac{(p_i - p_j)^2}{2(p_i + p_j)^2}.$$

Proof. The difference between the cost after t requests and the asymptotic cost is given by the last term in Theorem 2.1. Summing this over $0 \leqq t \leqq \infty$ gives the overwork. □

COROLLARY 2.1. $\text{OV}_{\text{MTF}} < n(n-1)/4$ *for every probability distribution with* $n > 2$.

Proof. Since $(p_i - p_j)^2/(p_i + p_j)^2 \leqq 1$ and cannot equal 1 for all pairs, we have $\text{OV}_{\text{MTF}} < \sum_{1 \leqq i < j \leqq n} \frac{1}{2} = n(n-1)/4$. Further, this bound is the best possible. Consider the distribution with $p_i = \varepsilon^{i-1}/K$, $K = (1-\varepsilon^{n+1})/1-\varepsilon$. By choosing ε sufficiently small we can make each term arbitrarily close to $\frac{1}{2}$ and the sum arbitrarily close to $n(n-1)/4$. □

The overwork of a rule tells us how significant the initial transient behavior of the rule is. For example, if we make $n(n-1)$ requests to the move to front rule, the expected number of comparisons *per request* will differ from the asymptotic value by at most $\frac{1}{4}$. The transient behavior of the rule can be safely ignored, and the chain can be assumed to start in steady state.

No general form has been found for the time-varying cost and overwork of the transposition rule. These can, however, be calculated for several simple distributions.

THEOREM 2.3. *If* $p_1 = 1$ *and* $p_i = 0$, $2 \leqq i \leqq n$, *then*

$$\text{OV}_{\text{MTF}} = \frac{n-1}{2} \quad \textit{and} \quad \text{OV}_{\text{TR}} = \frac{n^2-1}{6}.$$

Proof. OV_{MTF} is determined by substituting the p_i's into the formula in Theorem 2.2. To determine OV_{TR} note that if k_1 starts in position i, the overwork is $\sum_{j=1}^{i} (j-1) =$

$(i-1)i/2$ (the cost for accessing k_1 as it moves up one position at a time minus 1, the asymptotic cost). Since k_1 has probability $1/n$ for starting in any given position

$$\mathrm{OV}_{\mathrm{TR}} = \sum_{i=1}^{n} \frac{1}{n} \frac{(i-1)i}{2} = \frac{n^2-1}{6}. \quad \square$$

THEOREM 2.4. *If* $p_1 = 0$ *and* $p_i = 1/(n-1)$, $2 \leqq i \leqq n$, *then*

$$\mathrm{OV}_{\mathrm{MTF}} = \frac{n-1}{2} \quad \textit{and} \quad \mathrm{OV}_{\mathrm{TR}} = \frac{n^2-1}{6}.$$

Proof. $\mathrm{OV}_{\mathrm{MTF}}$ is determined by substituting the p_i into Theorem 2.2. To determine $\mathrm{OV}_{\mathrm{TR}}$, let A_i for $1 \leqq i \leqq n$ be a random variable equal to the number of requests that are made while k_1 is in position i. The cost of accessing the list when k_1 is in position i is $n/2 + n - i/(n-1)$. Since $n/2$ is the asymptotic cost, each request adds a contribution of $n - i/(n-1)$ into the overwork while k_1 is in position i. The overwork is then the expected sum of all these contributions, therefore

$$\mathrm{OV}_{\mathrm{TR}} = \sum_{i=1}^{n-1} \frac{n-i}{n-1} \cdot E(A_i).$$

To calculate $E(A_i)$, note that with probability $(n-i)/n$, k_1 is initially below position i, giving $A_i = 0$. With probability i/n, k_1 is in or above position i. It stays in position i until the key below it is requested (probability $1/(n-1)$). The expected time spent in position i is then $n-1$. Therefore $E(A_i) = (i/n) \cdot (n-1)$ and

$$\mathrm{OV}_{\mathrm{TR}} = \sum_{i=1}^{n-1} \frac{n-i}{n-1} \cdot \frac{i}{n} \cdot (n-1) = \frac{n^2-1}{6}. \quad \square$$

For both distributions, the difference in overwork is substantial: $\Omega(n)$ compared with $\Omega(n^2)$. We now consider a more complicated distribution.

THEOREM 2.5. *If* $p_i = 1/(iH_n)$, $1 \leqq i \leqq n$ *and* $H_n = \sum_{i=1}^{n} 1/i$ *(Zipf's Law), then*

$$\mathrm{OV}_{\mathrm{MTF}} = \frac{5n^2}{12} - (n^2+n+\tfrac{1}{6})(H_{2n} - H_n) + \tfrac{1}{6} H_n + \frac{n(n+1)(2n+1)}{3} [H_{2n}^{(2)} - H_n^{(2)}]$$

where $H_n = \sum_{i=1}^{n} 1/i$ *and* $H_n^{(2)} = \sum_{i=1}^{n} 1/i^2$. *For large* n, $\mathrm{OV}_{\mathrm{MTF}} \approx (\frac{3}{4} - \ln 2)n^2 \approx .057n^2$.

Proof. Substituting the p_i's into Theorem 2.2 gives

$$\frac{1}{2} \sum_{1 \leqq i < j \leqq n} \frac{(i-j)^2}{(i+j)^2} = \frac{1}{4} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{(i-j)^2}{(i+j)^2}.$$

Making the substitution $k = i + j$ gives

$$\frac{1}{4} \sum_{k=2}^{n} \sum_{j=1}^{k-1} \frac{(k-2j)^2}{k^2} + \frac{1}{4} \sum_{k=n+1}^{2n} \sum_{j=k-n}^{n} \frac{(k-2j)^2}{k^2}.$$

By expanding the numerators, the result simplifies to that given by the theorem in a straightforward but tedious manner. (See [7] for details.) To prove the asymptotic result, note $H_n \approx \ln n$, so $H_{2n} - H_n \approx \ln 2n - \ln n = \ln 2$, and $H_{2n}^{(2)} - H_n^{(2)} = 1/(2n) + O(n^{-2})$. $\square$

A comparison of $\mathrm{OV}_{\mathrm{MTF}}$ and $\mathrm{OV}_{\mathrm{TR}}$ for Zipf's Law is shown in Table 2.1. The results indicate $\mathrm{OV}_{\mathrm{TR}} = \Omega(n^3)$, again, much larger than $\mathrm{OV}_{\mathrm{MTF}}$. The values for $\mathrm{OV}_{\mathrm{TR}}$ are approximate and were obtained by successively calculating for $t = 0, 1, \cdots, T$ the

TABLE 2.1

The overwork for a list of n elements whose probabilities are given by Zipf's law.

N	Move to front rule	Transposition rule
3	0.2006	0.4579
4	0.4463	1.6503
5	0.7978	3.9793
6	1.2567	7.7514
7	1.8272	13.3005
8	2.5076	
9	3.2994	
10	4.2031	
11	5.2189	
12	6.3473	
13	7.5882	
14	8.9420	
15	10.4087	
16	11.9884	
17	13.6812	
18	15.4871	
19	17.4063	
20	19.4387	

vector consisting of the probabilities for each of the $n!$ orderings of the list after t requests. Initially, every component of this vector is $1/(n!)$. It is then successively multiplied by the $n! \times n!$ transition matrix. The cost at time t is easily determined from these vectors. The asymptotic cost is subtracted from each of these costs, and the resulting differences are summed, giving a good approximation of the overwork.

The second measure of convergence is the number of requests, r, required for the expected *total cost* of the transposition rule to become smaller than that of the move to front rule. The total cost is merely the sum of the costs for the first r requests and tells us which rule will be cheaper to use for retrieving r requests. Table 2.2 gives some values for r, assuming Zipf's Law. Since r increases quadratically with n, this measure also indicates that the move to front rule converges much more rapidly.

The conclusion is clear: although the transposition rule has lower asymptotic cost than the move to front rule, it decreases the cost more slowly, making the move to front rule the better choice if few requests ($O(n^2)$) will be made to the list.

TABLE 2.2

The number of requests, r, required for the transposition rule to have lower total cost than the move to front rule. We assume a list of n elements whose probabilities satisfy Zipf's law.

n	r
3	6
4	10
5	14
6	20
7	27
10	50
20	212

3. The first request rule. The *first request rule* is defined as follows: the *first* time a key is requested, it is moved up in the list until it comes to the top or a previously requested key. After that, it is not moved. Note that the keys occur in the list in order of their first request, and that after all keys have been requested, the ordering obtained is the same as if the keys had not been known a priori, and the list had been built by inserting a "new" key (one requested for the first time) at the end of the list.

The following theorem characterizes the performance of this rule.

THEOREM 3.1. *For any probability distribution over the initial lists, the probability of obtaining a given final list after any number of requests is the same for the move to front and first request rules.*

Proof. For any sequence of requests $r_1, \cdots, r_K$ to the move to front rule the sequence of requests $r_K, \cdots, r_1$ to the first request rule produces the same final list. Since every sequence that produces a given list using the move to front rule has a corresponding sequence of equal probability using the first request rule, the probability for obtaining any given list is equal for both rules. □

An interesting consequence of Theorem 3.1 is a reasonable situation where the move to front rule will not reduce the cost. Suppose the keys are not known a priori and the list is constructed by inserting a "new" key at the end of the list. Clearly, the resulting steady state distribution will be that of the first request rule, and the move to the front rule will *not* decrease the cost (since, by Theorem 3.1, its Markov chain is already in its steady state).

4. A hybrid rule. A "hybrid" rule that initially uses the move to front rule then switches to the transposition rule combines the best features of both rules: Initially it behaves like the move to front rule and has rapid convergence. Asymptotically, it behaves like the transposition rule and has low asymptotic cost. Determining when to switch rules is difficult (see [7]) and is the major disadvantage of this rule.

A better hybrid can be obtained by using the first request rule on a key's first request, then using the transposition rule on all of its subsequent requests. Because it does a cost reducing transposition on second and subsequent requests for a key, while the first request rule alone does nothing, this hybrid converges more quickly than the first request rule and hence, by Theorem 3.1, more quickly than the move to front/transposition hybrid.

The first request hybrid rule gives both rapid convergence and lower asymptotic cost while only slightly increasing the complexity of the algorithm to modify the data structure. The hybrid rule should be used in the region where both these features are important (say, $\Omega(n)$ to $\Omega(n^2)$ requests). Outside this region, the additional overhead of this rule is not compensated for.

5. Rules that use counters. In this section, we consider rules that use "extra" storage associated with each key as a counter. This allows improved performance over the permutation rules. The obvious rule is the *frequency count rule* which keeps the keys sorted by the number of times each has been requested. Asymptotically, this rule gives the optimal ordering; because of the law of large numbers, if $p_1 > p_2$, the probability k_1 has been requested more times than k_2 (and hence is ahead of k_2 in the list) approaches 1. In addition, if we have no a priori knowledge about the request probabilities, the rule always produces the ordering with the lowest expected cost. If k_1 has been requested more times than k_2, then Prob $(p_1 > p_2)$ is larger than Prob $(p_1 < p_2)$, and k_1 must be ahead of k_2 in the list with lowest expected cost, and, of course, the frequency count rule produces this ordering.

The disadvantage of this rule is that the counters grow in proportion to the p_i's and overflow whatever field is allotted them. When a counter overflows, some method of reducing the fields must be applied (perhaps subtracting a constant or dividing by 2). If the fields are small, the cost of this would be prohibitive.

The rate of growth of the count fields can be decreased by storing with the ith key the difference between the count of the $i-1$st key and the ith key. These differences contain enough information to keep the list sorted by frequency count. Also, since they grow in proportion to the *difference* between successive probabilities, they will be smaller than the frequency counts, and the frequency count rule will take longer to overflow its fields. Note that this rule can be cheaply executed because each request will alter only two difference fields.

However, even with this improvement, the rule still requires an unbounded amount of storage. Our purpose in this section is to define, analyze and compare various rules that use a limited (and hopefully very small) amount of storage.

5.1. The limited difference rule. The first of these rules is the *limited difference rule*, which stores the differences between successive counts but imposes an upper bound (a parameter of the rule) on their size. A field that has reached this limit is *not* increased by subsequent requests, but may, of course, later be decreased. The limited difference rule is not optimal, but its performance rapidly approaches the optimum as the maximum difference increases; even small maximum differences give nearly optimal performance (see Table 5.1). For a list of two elements, this approach is exponential.

TABLE 5.1
Comparison of rules that use counters.

	$c=0$	$c=1$	$c=2$	$c=3$	$c=4$	$c=5$
Limited difference rule (maximum difference = c)	3.9739	3.4162	3.3026	3.2545	3.2288	3.2113
Wait c, move to front and clear (exact)	3.9739	3.6230	3.4668	3.3811	3.3285	
Wait c, transpose and clear (exact)	3.4646	3.3399	3.2929	3.2670	3.2501	
Wait c and move to front (exact)	3.9739	3.8996	3.8591	3.8338	3.8165	3.8040
Wait c and transpose	3.4646	3.3824	3.3576	3.3473	3.3312	3.3272

Asymptotic costs for various rules assuming a nine element list whose probabilities are given by Zipf's law. Compare these with the optimal cost which is 3.1814. Cost for the limited difference rule and the wait c and transpose rule were estimated by simulations consisting of 1,000 requests. The average of 200 trials is shown.

THEOREM 5.1.1. *For a list of two elements with probabilities a and b (assume $b>a$), the difference between the asymptotic cost of the limited difference rule and the optimal cost is*

$$\frac{2b(b-a)(b/a)^c-(b-a)}{[(b/a)^{2c+1}-1]}\approx\frac{2b(b-a)}{(b/a)^{c+1}}$$

where c is the maximum difference.

Proof. The behavior of the list can be modeled by a Markov chain with $2c+2$ states:

A_i, $0\leqq i\leqq c$ where the key with probability a is first in the list and the difference is i.

B_i, $0\leqq i\leqq c$ where the key with probability b is first in the list with difference i.

It is easy to verify that the steady state equations are:

$$\begin{aligned}
&A_c = aA_{c-1}+aA_c, && B_c = bB_{c-1}+bB_c,\\
&A_i = aA_{i-1}+bA_{i+1}, && B_i = bB_{i-1}+aB_{i+1}, \qquad 2\leqq i\leqq c-1,\\
&A_1 = bA_2+aA_0+aB_0, && B_1 = aB_2+bB_0+bA_0,\\
&A_0 = bA_1, && B_0 = aB_1,
\end{aligned}$$

and, in addition, $\sum_{i=0}^{c} A_i + \sum_{i=0}^{c} B_i = 1$. We solve this system of equations to get

$$A_0 = bA_c(b/a)^{c-1}, \qquad B_0 = aA_c(b/a)^{c+1},$$
$$A_i = A_c(b/a)^{c-i}, \qquad B_i = A_c(b/a)^{c+i}, \qquad 1 \leqq i \leqq n,$$

and

$$A_c = \frac{a-b}{a[1-(b/a)^{2c+1}]}.$$

The cost of the list is

$$\begin{aligned}
&(a+2b)\,\text{Prob (key with probability } a \text{ is first in list)} \\
&\quad +(b+2a)\,\text{Prob (key with probability } b \text{ is first)} \\
&\quad =(a+2b)\sum_{i=0}^{c} A_i + (b+2a)\sum_{i=0}^{c} B_i \\
&\quad =(b+2a)+\frac{2b(b-a)(b/a)^c-(b-a)}{[(b/a)^{2c+1}-1]}.
\end{aligned}$$

Now, if $b > a$, the optimal cost is the first term in this expression. The difference from the optimum is then as given in the theorem. □

Hence the difference from the optimum decreases exponentially with base b/a. Two observations result from this fact. First, the "flatness" of the distribution (determined by how close b/a is to one in this case) determines the number of bits required to distinguish the more probable elements. The flatter the distribution, the more bits will be required.

A second observation is that since the decrease is exponential, we would expect the cost to be nearly optimal even for small values of c. The results of a simulation, shown in Table 5.1, support this conclusion. In addition to nearly optimal asymptotic cost, the convergence of this rule is also very rapid. Until a difference field reaches the upper limit, the limited difference rule behaves exactly like the version of the frequency count rule that uses difference fields. Thus, for this initial segment, the limited difference rule converges as rapidly as possible. Therefore, both in terms of asymptotic cost and convergence, the rule is nearly optimal.

5.2. Wait *c*, move and clear rules. The second class of rules we will study is the *wait c, move and clear rules*. These rules associate a field, initially zero, with each key. When a key is accessed, the corresponding field is incremented. If it exceeds c, the maximum value, the key is "moved" using the corresponding permutation rule (for example, the wait c, *transpose* and clear rule uses the *transposition* rule), and the field of *every* key is reset to zero. The cost of resetting the fields may be very significant. However, if all fields are stored in the same area (instead of being directly associated with the key), they can be efficiently reset by zeroing a contiguous area of core. The performance of these rules is analyzed by the following theorem.

THEOREM 5.2.1. *The asymptotic cost of a wait c, move and clear rule with request probabilities* $p_1, \cdots, p_n$ *equals the asymptotic cost of the corresponding permutation rule with modified request probabilities* $\hat{p}_1(c), \cdots, \hat{p}_n(c)$, *where* $\hat{p}_i(c)$ *equals*

$$\sum_{a_1=0}^{c} \cdots \sum_{a_{i-1}=0}^{c} \sum_{a_{i+1}=0}^{c} \cdots \sum_{a_n=0}^{c} \frac{(c+a_1+\cdots+a_{i-1}+a_{i+1}\cdots+a_n)!}{c!a_1!\cdots a_{i-1}!a_{i+1}!\cdots a_n!} \cdot p_i^{c+1} p_1^{a_1} \cdots p_{i-1}^{a_{i-1}} p_{i+1}^{a_{i+1}} \cdots p_n^{a_n},$$

(*the probability that* k_i *is requested* $c+1$ *times before any other key is requested* $c+1$ *times.*)

Proof. Consider the sequence of keys moved by a wait c move and clear rule. Since all fields are cleared after every move, successive moves are independent, and the move

probabilities do not vary with respect to time. Therefore, the move sequence satisfies the same assumptions as a sequence of requests. Using the *move* sequence as inputs (requests) to the corresponding permutation rule results in the list given by the wait c, move and clear rule. However, since the probability that a key is an input to the permutation rule equals the probability that it is in the move sequence, the input (request) probabilities the permutation rule "sees" are exactly the $\hat{p}_i(c)$ given by the theorem.

This would complete the proof if every request to the wait c, move and clear rule caused a move. This is not the case since we must wait after each move while the counts build up. If this waiting time were dependent on the current state, states with longer waiting times would have proportionally greater probabilities. Fortunately, this is not the case. After each move, the counts are reset and hence each state will have the same expected waiting time. □

COROLLARY 5.2.1. *For any c, the asymptotic cost of the wait c, transpose and clear rule is less than or equal to that of the wait c, move to front and clear rule for every distribution.*

Intuitively, a wait c, move and clear rule has lower cost than the corresponding permutation rule because it has "shifted" the move probabilities to favor the high probability keys. These now have a proportionally greater chance of being moved. We prove this for the wait c, move to front and clear rule, then show both rules approach the optimum as $c \to \infty$, as was the case for the limited difference rule.

THEOREM 5.2.2. *For $c \geqq 1$ the asymptotic cost of the wait c, move to front and clear rule is less than that of the move to front rule.*

Proof. Renumber the p_i so that $p_1 \geqq p_2 \geqq \cdots \geqq p_n$.

$$\begin{aligned} \mathrm{E(Cost)} &= 1 + \sum_{i=1}^{n} p_i \sum_{j \neq i} \mathrm{Prob}\,(k_j \text{ ahead of } k_i) \\ &= 1 + \sum_{i>j} [p_i - (p_i - p_j)\,\mathrm{Prob}\,(k_i \text{ ahead of } k_j)]. \end{aligned}$$

The probability that k_i is ahead of k_j using the wait c, move to front and clear rule is Prob (k_i is requested $c+1$ times before k_j is requested $c+1$ times). If $p_i > p_j$, this is greater than Prob (k_i is requested once before k_j is requested once), the probability of k_i being ahead using the move to front rule. Since the wait c, move to front and clear rule increases Prob (k_i ahead of k_j), it decreases the cost. □

THEOREM 5.2.3. *As $c \to \infty$ the asymptotic cost of a wait c, move and clear rule approaches the optimum.*

Proof. For the wait c, move to front and clear rule, Prob (k_i is ahead of k_j) = Prob (k_i is requested $c+1$ times before k_j is requested $c+1$ times). As $c \to \infty$, this approaches 1 if $p_i > p_j$ and 0 if $p_i < p_j$, hence the optimal ordering is approached. Since the cost of the wait c, transpose and clear rule is smaller than that of the wait c, move to front and clear rule by Corollary 5.2.1 it also approaches the optimum. □

Table 5.1 compares the wait c move and clear rules with the limited difference rule. Their performance is good (by Theorem 5.2.3 it approaches the optimum), but it is surpassed by that of the limited difference rule.

Comparing the convergence gives a decisive advantage to the limited difference rule. The wait c, move and clear rules decrease the cost much more slowly than the corresponding permutation rule with modified probabilities since a counter must exceed c for a move to be made. In the worst case, where every key is requested c times before any key is requested $c+1$ times, a move will be done every $cn+1$ requests, and

the convergence will be a factor of $\Omega(n)$ slower than the corresponding permutation rule. On the other hand, the *best* case occurs when the same key is requested $c+1$ times, and a move will be made every $c+1$ requests. Hence, the convergence must be slowed by *at least* a factor of $c+1$.

For an idea of the average decrease in convergence, consider n equally likely keys and $c=1$. Note that the expected number of requests before a move is made is maximized by this distribution. The expected number of requests before a key is requested for a second time equals

$$\sum_{i=0}^{n} \text{Prob (no key has been requested twice after } i \text{ requests).}$$

This probability equals the number of sequences of length i of distinct keys $n!/((n-i)!)$ divided by the total number of sequences (n^i)

$$= \sum_{i=0}^{n} \frac{n!}{(n-i)!}\frac{1}{n^i} = \sqrt{\frac{\pi n}{2}} - \frac{1}{3} + O(n^{-1/2})$$

by [9, eq. 1.2.11 (25)]. Thus, even if $c=1$, the convergence is slowed by a factor of $\Omega(\sqrt{n})$ over the permutation rules, resulting in extremely slow convergence.

Clearly then, the wait c, move and clear rules are outperformed by the limited difference rule. Though their asymptotic cost approaches the optimum as $c \to \infty$, it is still higher than that of the limited difference rule. In addition, these rules converge much more slowly than the limited difference rule.

5.3. Wait c and move rules. The third class of rules that use a fixed amount of storage is the *wait c and move rules*. These behave like the wait c, move and clear rules except that when a key is moved, only *its* field is reset. The wait c and move to front rule is analyzed in the following theorem.

THEOREM 5.3.1. *The asymptotic cost of the wait c and move to front rule is*

$$1 + \sum_{i \neq j} p_i D_{ji}$$

where $D_{ji} = \text{Prob}\,(k_j \textit{ ahead of } k_i)$

$$= \frac{p_j}{(p_i+p_j)(c+1)^2} \sum_{k=0}^{c} (c-k+1)\left(\frac{p_i}{p_i+p_j}\right)^k \sum_{m=0}^{c} \binom{m+k}{k}\left(\frac{p_j}{p_i+p_j}\right)^m.$$

Proof. Neither the count fields nor the relative ordering of two given keys (k_x and k_y) is effected by requests for other keys. To determine $D_{yx} = \text{Prob}\,(k_y \text{ is ahead of } k_x)$, it suffices to ignore requests to all other keys and consider a list consisting only of k_x and k_y, requested with probabilities $(p_x/(p_x+p_y))(=a)$ and $(p_y/(p_x+p_y))(=b)$. The list is modeled by a Markov chain with $2(c+1)^2$ states:

A_{ij}, $0 \leqq i, j \leqq c$ where k_x is first in the list with count i and k_y has count j.

B_{ij}, $0 \leqq i, j \leqq c$ where k_y is first in the list with count j and k_x has count i.

(Note that the first subscript is always k_x's count.)
The steady state equations are:

$$A_{ij} = aA_{i-1,j} + bA_{i,j-1}, \qquad B_{ij} = aB_{i-1,j} + bB_{i,j-1},$$

$$A_{0j} = bA_{0,j-1} + aA_{cj} + aB_{cj}, \qquad B_{0j} = bB_{0,j-1},$$

$$A_{i0} = aA_{i-1,0}, \qquad B_{i0} = aB_{i-1,0} + bA_{ic} + bB_{ic},$$

for $0 < i, j \leqq c$ and

$$A_{00} = aA_{c0} + aB_{c0}, \qquad B_{00} = bA_{0c} + bB_{0c}.$$

Before solving for the steady state probabilities, we make two observations: First, it is easily seen that if this chain leaves a given state, then later returns, the number of transitions made in the meantime must be a multiple of $c+1$. Such a chain is said to be *periodic* with period $c+1$. (All previous chains have been *aperiodic*, i.e. their period was one.)

A chain which is periodic does not converge to its steady state distribution in the sense that $\lim_{t\to\infty} P_t(x) = p(x)$, where $P_t(x)$ is the probability of being in state x after t transitions and $p(x)$ is the steady state probability of state x. However, since this chain is irreducible, the ergodic theorem holds (see [8]). This states that $\lim_{t\to\infty} (1/t)\sum_{i=0}^{t} P_t(x) = p(x)$. Thus, the chain asymptotically spends a fraction equal to $p(x)$ of its time in state x, and if we observe the chain at a random time it has probability $p(x)$ of being in state x. If c_t is the expected cost at time t, $\lim_{t\to\infty} (1/t)\sum_{i=0}^{t} c_i = \sum_x p(x)c(x)$ where $c(x)$ is the cost of state x. The cost converges to the asymptotic cost in this sense. Note that the asymptotic cost is still the steady state probability of a state times its cost summed over all states; only the strength of convergence has been changed.

The second observation is that we must wait in each state of the two element chain while keys other than A and B are being requested. However, since key requests do not depend on whether A is ahead of B, or the count of either key, these requests are independent of the state, and hence the expected waiting time is the same for each state. Therefore, the steady state distribution will give Prob (A ahead of B).

We now solve the steady state equations. By adding pairs of equations, we can verify $A_{ij} + B_{ij} = 1/(c+1)^2$. This corresponds to the intuitive fact that asymptotically, every pair of count fields (without regard to the order of the list) is equally likely. Substituting this relation gives

$$A_{ij} = aA_{i-1,j} + bA_{i,j-1} \qquad \text{for } 0 < i, j \leqq c,$$

$$A_{0j} = bA_{0,j-1} + \frac{a}{(c+1)^2} \qquad \text{for } 0 < j \leqq c,$$

$$A_{i0} = aA_{i-1,0} \qquad \text{for } 0 < i \leqq c,$$

$$A_{00} = \frac{a}{(c+1)^2}.$$

Or, equivalently,

$$A_{ij} = aA_{i-1,j} + bA_{i,j-1} \qquad \text{for } 0 \leqq i, j \leqq c,$$

$$A_{-1,j} = \frac{1}{(c+1)^2} \qquad \text{for } 0 \leqq j \leqq c,$$

$$A_{i,-1} = 0 \qquad \text{for } 0 \leqq i \leqq c.$$

Extending these recurrences to hold for all $i, j \leqq 0$ will not affect the A_{ij} we are interested in. The recurrence can then be solved using generating functions. Define

$$F(x, y) = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} A_{ij} x^i y^i.$$

Substituting for A_{ij} gives

$$F(x, y) = axF(x, y) + \frac{a}{(c+1)^2(1-y)} + by\,F(x, y)$$

and

$$F(x, y) = \left(\frac{a}{(c+1)^2(1-y)}\right)\left(\frac{1}{1-ax-by}\right) = \frac{a}{(c+1)^2}\left(\sum_{i=0}^{\infty} y^i\right)\left(\sum_{j=0}^{\infty} (ax+by)^j\right).$$

Using the binomial theorem gives

$$F(x, y) = \frac{a}{(c+1)^2}\left(\sum_{i=0}^{\infty} y^i\right)\left(\sum_{j=0}^{\infty}\sum_{k=0}^{j}\binom{j}{k}(ax)^{j-k}(by)^k\right)$$

$$= \frac{a}{(c+1)^2}\sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\sum_{k=0}^{j}\binom{j}{k}a^{j-k}b^k x^{j-k} y^{i+k}.$$

Now substitute i' for $j-k$ and j' for $i+k$ and then drop the primes;

$$F(x, y) = \frac{a}{(c+1)^2}\sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\sum_{k=0}^{j}\binom{i+k}{k}a^i b^k x^i y^j.$$

Therefore

$$A_{ij} = \frac{a}{(c+1)^2}\sum_{k=0}^{j}\binom{i+k}{k}a^i b^k.$$

Prob (A ahead of B) is then

$$\sum_{i=0}^{c}\sum_{j=0}^{c} A_{ij} = \frac{a}{(c+1)^2}\sum_{k=0}^{c}(c-k+1)b^k\sum_{i=0}^{c}\binom{i+k}{k}a^i.$$

Substituting this into the formula $E(\text{Cost}) = 1 + \sum_{i \neq j} p_i$ Prob (k_j is ahead of k_i) and recalling that a and b were originally $p_x/(p_x+p_y)$ and $p_y/(p_x+p_y)$ finishes the proof. □

The following two theorems help to compare the wait c and move to front rule with the previous rules. First, we show it does have lower asymptotic cost than the move to front rule. Then we show its cost *does not* approach the optimum as $c \to \infty$.

THEOREM 5.3.2. *For $c \geqq 1$ and any set of request probabilities except the uniform distribution or a distribution with a key of probability one, the wait c and move to front rule has strictly lower cost than the move to front rule.*

Proof. (Note that for the distributions cited above, the rules have equal cost.) Let $a = p_i/(p_i+p_j)$ and $b = p_j/(p_i+p_j)$. Consider

$$\text{Prob } (k_i \text{ ahead of } k_j) - \frac{a}{b} \cdot \text{Prob } (k_j \text{ ahead of } k_i) \tag{5.3.1}$$

for the wait c and move to front rule.

We show that if $p_i > p_j$ and $p_i \neq 1$, then (5.3.1) is positive.
By Theorem 5.3.1, (5.3.1) equals

$$\frac{a}{(c+1)^2}\sum_{k=0}^{c}\sum_{m=0}^{c}(c-k+1)\binom{m+k}{k}[a^m b^k - a^k b^m].$$

The terms where $k = m$ vanish. Consider the remaining terms in pairs where the (k, m)

term is paired with (m, k). These equal

$$(c-k+1)\binom{m+k}{k}[a^m b^k - a^k b^m] + (c-m+1)\binom{m+k}{m}[a^k b^m - a^m b^k]$$

$$= (m-k)\binom{m+k}{m}[a^m b^k - a^k b^m] > 0.$$

Adding all the pairs together and multiplying by $a/(c+1)^2$ gives (5.3.1). Hence (5.3.1) is positive.

Since (5.3.1) is positive, Prob (k_i ahead of k_j) $> a$, which is Prob (k_i ahead of k_j) for the move to front rule. Using the argument from Theorem 5.2.2 then shows the wait c and move to front rule has lower cost. □

An intuitive explanation of the theorem is as follows: Previously, the wait c, move and clear rules decreased the cost by altering the probability that a key is moved from the request probabilities to a more favorable distribution. The wait c and move to front rule does not do this; since a key is moved after every $(c+1)$st request for it, the move probabilities remain unchanged in the sense that a key requested with probability p_i will account for a fraction of the total number of moves equal to p_i. If successive moves were independent, the cost would be the same as the move to front rule. However, this is not the case. Consider any two keys, k_i and k_j. After k_i has been moved (assume $p_i > p_j$), its count is set to zero. Asymptotically, k_j's count is uniformly distributed over $0, 1, \cdots, c$. After k_j has been moved, its count is zero, and k_i's count ranges uniformly from zero to c. These two cases are obviously symmetric with the roles of k_i and k_j reversed. Clearly, in the case where k_j has been moved, the next move will occur sooner because the count of k_i (the more probable key) is closer to causing a move. Therefore, the probability that k_i is ahead of k_j is increased because we must wait longer for the next move in states where k_i is ahead of k_j.

Before proving the next theorem, we provide an intuitive interpretation of the formula given in Theorem 5.3.1. Rewriting, we get

$$\frac{a}{(c+1)^2} \sum_{i=0}^{c} \sum_{j=0}^{c} \sum_{k=0}^{j} \binom{i+k}{k} a^i b^k$$

where we consider k_x and k_y and $a = p_x/(p_x + p_y)$, $b = p_y/(p_x + p_y)$.

At a random time, any count pair (i', j') is equally likely (probability $1/(c+1)^2$). Let $i = c - i'$ and $j = c - j'$ (the number of requests until a given key is moved). For the count pair (i', j'), we calculate the probability that k_x is moved before k_y. Clearly *any* sequence of requests where k_x is requested i times and k_y is requested k times (for any $k < j$), followed by a request for k_x will cause k_x to be moved first. Thus the probability for moving k_x is $a \cdot \sum_{k=0}^{j} \binom{i+k}{k} a^i b^k$. Multiplying by $1/(c+1)^2$, the probability of this point, and summing over all points gives the probability k_x is moved, which equals the probability it is ahead of k_y.

THEOREM 5.3.3. *Except for the uniform distribution and distributions having a key of probability one, the wait c and move to front rule does not approach the optimum as* $c \to \infty$.

Proof. Choose any two keys k_x and k_y with $p_x > p_y$ and $p_x \neq 1$. For some large c, consider a grid of points $\{(i/c, j/c)$ for $0 \leqq i, j \leqq c\}$ where the pair $(i/c, j/c)$ corresponds to the states where the chain has counts i and j. As requests are made, the point representing the chain's state has probability $p_x/(p_x + p_y) = a$ of moving $1/c$ in the x-direction and probability $p_y/(p_x + p_y) = b$ of moving $1/c$ in the y-direction. As $c \to \infty$, these transitions become smaller and smaller, and the point moves in the x-direction with velocity a and in the y-direction with velocity b. (The time scale must

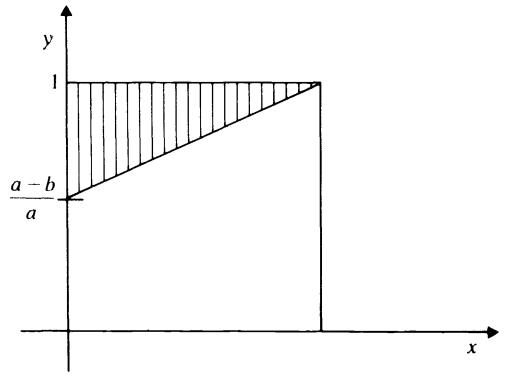


FIG. 5.3.1. *Points in the shaded region reach the* $y = 1$ *boundary first.*

be readjusted so that c requests are made per unit time.) If the chain begins in the shaded region in Fig. 5.3.1, it reaches the $y = 1$ boundary before the $x = 1$ boundary and causes k_y to be ahead of k_x.

If we begin observing the chain at a random time, every point of the grid is equally likely. Since the shaded region has area $b/(2a)$, Prob (k_y ahead of k_x) $= b/(2a) = p_y/(2p_x) > 0$, as $c \to \infty$. Therefore, the cost does not approach the optimum. □

COROLLARY 5.3.1. *Renumber the keys such that* $p_1 \geqq p_2 \cdots \geqq p_n$; *then as* $c \to \infty$, *the cost of the wait c and move to front rule approaches*

$$1 + \sum_{i<j} \left[\frac{3}{2} p_j - \frac{p_j^2}{2p_i} \right].$$

Proof. From the proof of Theorem 5.3.3, Prob (k_j ahead of k_i) $= p_j/(2p_i)$ if $j > i$. Substituting this into

$$E(\text{Cost}) = 1 + \sum_{i<j} \left[p_j + (p_i - p_j) \frac{p_j}{2p_i} \right]$$

(see Theorem 5.2.2) proves the corollary. □

The wait c and transpose rule has not yet been exactly analyzed. The results of a simulation of this rule are shown in Table 5.1 along with exact values for the wait c and move to front rule. The cost of these rules is higher than both the wait c, move and clear rules and the limited difference rule. (Of course, from Theorem 5.3.3, we know it cannot approach the optimum as the other rules do.) Also, since the wait c and move rules make a move every $c + 1$ requests, on the average, they have faster convergence than the wait c, move and clear rules, but slower than the limited difference rule. Again, the limited difference rule is superior in both convergence and asymptotic cost.

6. Monotonic tree rule. The *monotonic tree rule* keeps the tree ordered so that the most frequently requested key is the root of the tree. Both subtrees are recursively ordered in the same manner. (This property is the same as that required for a heap, see Williams [11].) This property can be easily maintained as requests are made; rotations are used to successively promote the requested key until it becomes the root or until its count is less than or equal to that of its father.

Asymptotically, the most probable key will be the root (by the law of large numbers, it will be requested the most times), and each subtree will be recursively ordered by the probabilities. The worst case cost for this asymptotic tree (called a

monotonic tree) can be very high. Suppose key k_i has probability p_i and the lexicographic ordering of the keys is $k_1 < k_2 < \cdots < k_n$. If $p_1 > p_2 > p_3 > \cdots > p_n$, the skewed tree shown in Fig. 6.1 will result. If the p_i are approximately equal, this tree has much higher cost than the optimal tree, which is more balanced. The following theorem shows how large this difference can be.

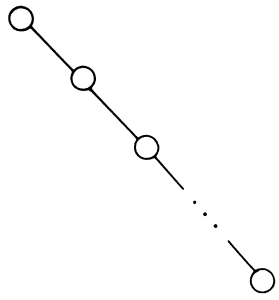


FIG. 6.1. *A worst case monotonic tree.*

THEOREM 6.1 (Melhorne [12]). *The ratio between the cost of a monotonic tree and the optimal tree may be as high as* $n/(4 \log n)$ *for trees with n nodes.*

This theorem depends on a very unfavorable choice for the probabilities and only gives an idea of the worst case performance of monotonic trees. We consider how these trees perform "on the average" by assuming the probabilities are randomly chosen in some way. The first method of randomly choosing the probabilities assumes we have a fixed set of probabilities $\{p_1, \cdots, p_n\}$ and that they are randomly assigned to the n keys with each of the $n!$ assignments being equally likely. The following theorem gives the expected cost for a monotonic tree in this case.

THEOREM 6.2 (Knuth [6, p. 432]). *Given n keys and n probabilities* $(p_1 \geqq p_2 \geqq \cdots \geqq p_n)$, *if each of the* $n!$ *assignments of probabilities to keys is equally likely, the expected cost of a monotonic tree is* $[2 \sum_{i=1}^{n} H_i p_i] - 1$, *where* $H_n = \sum_{i=1}^{n} 1/i$.

We now determine upper and lower bounds for this cost, showing it is largely determined by the entropy, $H \equiv -\sum_{i=1}^{n} p_i \log p_i$ of the distribution.

THEOREM 6.3. *Given n keys and n probabilities, if each of the* $n!$ *assignments of probabilities to keys is equally likely, the expected cost of a monotonic tree* (COST_{MON}) *satisfies*:

$$(2 \ln 2)H - f(n) \leqq \text{COST}_{\text{MON}} \leqq (2 \ln 2)H + 1$$

where $f(n) = 2 \ln A_n + 1 - (2/A_n) \sum_{i=1}^{n} e^{1/i!} [1/(i!) + H_i]$ *and* $A_n = \sum_{i=1}^{n} e^{1/i!}$. *Further,*

$$f(n) < 3 - 2\gamma + \frac{2(\alpha - 1)[\gamma + \ln n]}{n+1}$$

where $\gamma \approx 0.577$ *is Euler's constant and* $\alpha = (e-1)^2 \approx 2.953$. *For large n,* $f(n)$ *is bounded by* $3 - 2\gamma \approx 1.846$.

Proof. (*Upper bound*). Consider

$$(2 \ln 2)H + 1 - \text{COST}_{\text{MON}} = 2 \sum_{i=1}^{n} p_i (\ln \frac{1}{p_i} + 1 - H_i).$$

Since $p_1 \geqq p_2 \geqq \cdots \geqq p_n$, we have $p_i \leqq 1/i$ and $\ln 1/p_i \geqq \ln i$. Therefore

$$(2 \ln 2)H + 1 - \text{COST}_{\text{MON}} \geqq 2 \sum_{i=1}^{n} p_i(\ln i + 1 - H_i).$$

Since $\ln i + 1 \geqq H_i$, the sum is nonnegative, proving the upper bound.

Further, this is the best possible upper bound of the form $aH + b$. For the distribution $p_1 = 1$ and $p_i = 0$ for $i > 1$, we have $H = 0$ and $\text{COST}_{\text{MON}} = 1$. Hence $b \geqq 1$. For the uniform distribution, $H = \log n$ and COST_{MON} is $(2(n+1)/n)H_n - 3$ (see Cor. 6.1). Asymptotically $\text{COST}_{\text{MON}} = 2 \ln n$, forcing $a \geqq 2 \ln 2$. Thus $(2 \ln 2)\, H + 1$ is the best possible bound.

(*Lower bound*). Consider the function

$$(2 \ln 2)H - \text{COST}_{\text{MON}}. \tag{6.1}$$

We first find the distribution that maximizes it. Consider any k and let $p_k = x$ and $p_{k+1} = c - x$. Given that these two probabilities must sum to c and $p_k \geqq p_{k+1}$, we find the choice of x that maximizes (6.1). Function (6.1) is equal to

$$\begin{aligned}(-2 \ln 2)[x \log x + (c-x) \log (c-x)] - H_k x - H_{k+1}(c-x) \\ + \text{terms independent of } x \text{ and } c.\end{aligned} \tag{6.2}$$

Differentiating with respect to x and simplifying gives

$$2 \ln \left(\frac{c-x}{x}\right) + \frac{2}{k+1},$$

which has a zero at $x_0 = ce^{1/k+1}/(1 + e^{1/k+1})$, which is the maximum. (Note to the left of x_0, (6.2) is strictly increasing, to the right, strictly decreasing.)

Therefore the choice

$$p_k = \frac{ce^{1/k+1}}{1 + e^{1/k+1}} \quad \text{and} \quad p_{k+1} = \frac{c}{1 + e^{1/k+1}} \tag{6.3}$$

maximizes (6.2). The distribution in which all the p_i satisfy (6.3) must maximize (6.1). Consider any other distribution. By increasing or decreasing some p_i (whichever moves it closer to x_0) we increase (6.2) and hence (6.1). Therefore this distribution cannot be the maximum.

To determine the distribution satisfying (6.3), note

$$p_{k+1} = e^{1/k+1} p_k.$$

Therefore

$$p_i = e^{1/i!} p_1 \quad \text{for } i = 1, \cdots, n.$$

To determine p_1, we use $\sum_{i=1}^{n} p_i = 1$, giving

$$p_1 = 1/\sum_{i=1}^{n} e^{1/i!} \equiv 1/A_n.$$

Therefore $p_i = e^{1/i!}/A_n$. We now determine $(2 \ln 2)H - \text{COST}_{\text{MON}}$ for this distribution. Substituting for the p_i's and simplifying gives

$$2 \ln A_n + 1 - \frac{2}{A_n} \sum_{i=1}^{n} e^{1/i!}\left[\frac{1}{i!} + H_i\right], \tag{6.4}$$

the $f(n)$ given in the statement of the theorem. To derive an upper bound, we first use

the Taylor's series expansion for e^x to give

$$A_n = \sum_{i=1}^{n} e^{1/i!} = \sum_{i=1}^{n} \sum_{j=0}^{\infty} \frac{1}{(i!)^j j!}$$

$$= n + \sum_{j=1}^{\infty} \frac{1}{j!} \sum_{i=1}^{n} \frac{1}{(i!)^j}$$

$$\leqq n + \sum_{j=1}^{\infty} \frac{1}{j!} \sum_{i=1}^{n} \frac{1}{i!}$$

$$\leqq n + \sum_{j=1}^{\infty} \frac{e-1}{j!} = n + (e-1)^2.$$

Let $\alpha = (e-1)^2 \approx 2.953$, The expression in (6.4) is less than

$$2 \ln (n+\alpha) + 1 - \frac{2}{n+\alpha} \sum_{i=1}^{n} e^{1/i!} \left[\frac{1}{i!} + H_i \right]$$

$$\leqq 2 \ln (n+\alpha) + 1 - \frac{2}{n+\alpha} \sum_{i=1}^{n} H_i$$

$$= 2 \ln (n+\alpha) + 1 - \frac{2}{n+\alpha} [(n+1)H_n - n].$$

We then use the inequalities $\ln (n+\alpha) < \ln n + \alpha/n$, $1/(n+\alpha) > 1/n - \alpha/n^2$, $1/(n+\alpha) > 1/(n+1) - (\alpha-1)/(n+1)^2$, and $H_n > \ln n + \gamma$ to give that the expression in (6.4)

$$\leqq 2 \ln n + \frac{2\alpha}{n} + 1 - 2\left(\frac{1}{n+1} - \frac{\alpha-1}{(n+1)^2} \right)(n+1)(\ln n + \gamma) + 2\left(\frac{1}{n} - \frac{\alpha}{n^2} \right) n$$

$$= 3 - 2\gamma + \frac{2(\alpha-1)[\gamma + \ln n]}{n+1}. \qquad \square$$

For purposes of comparison, we give the cost of the optimal tree.

THEOREM 6.4 (Bayer[19]). *Given n keys with probabilities* $p_1, \cdots, p_n$, *the cost of the optimal tree* (COST_{OPT}) *satisfies*

$$H - \log H - \log e + 1 \leqq \text{COST}_{\text{OPT}} \leqq H + 1,$$

where $H = -\sum_{i=1}^{n} p_i \log p_i$.

Comparing Theorems 6.3 and 6.4 shows that the monotonic tree rule will perform nearly optimally when the entropy is small. In this case, there will be several high probability keys, and these will be correctly placed near the root of the tree. For high entropy distributions, this crude heuristic does poorly; the keys have nearly equal probability, and now the important concern is the *shape* of the tree, not the location of the keys with highest probability. In this case, the monotonic tree differs significantly from the optimum.

We are also interested in the expected cost of a *random tree*, which is built by successively inserting the n keys, with each of the $n!$ insertion sequences being equally likely.

THEOREM 6.5. *Given n keys and n probabilities, if each of the* $n!$ *assignments of probabilities to keys is equally likely, the expected cost of a random tree is* $2((n+1)/n)H_n - 3$ *for any set of probabilities.*

Proof. Let $p(k_i)$ for $1 \leqq i \leqq n$ be random variables denoting the probability chosen for k_i and let l_i denote the level of k_i. We have

$$E(\text{Cost}) = E\Big(\sum_{i=1}^{n} p(k_i)l_i\Big) = \sum_{i=1}^{n} E(p(k_i)l_i).$$

Since the insertion sequence (and hence l_i) does not depend on $p(k_i)$, these two random variables are independent and

$$E(\text{Cost}) = \sum_{i=1}^{n} E(p(k_i))E(l_i) = \frac{1}{n}\sum_{i=0}^{n} E(l_i).$$

Now k_j will be an ancestor of k_i if and only if k_j occurs in the insertion sequence before k_i and before any key that is between k_i and k_j in the ordering on the keys. The probability of this is $1/(|i-j|+1)$. Therefore since $E(l_i) = 1 + \sum_{j \neq i} \text{Prob}\ (k_j \text{ is an ancestor of } k_i)$ we have

$$\begin{aligned} E(l_i) &= 1 + \sum_{j \neq i} \frac{1}{|i-j|+1} = 1 + \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1} \\ &= 1 + [H_i - 1] + [H_{n-i+1} - 1] \end{aligned}$$

and

$$\frac{1}{n}\sum_{i=1}^{n} E(l_i) = \frac{1}{n}\Big[\sum_{i=1}^{n} H_i + H_{n-i+1}\Big] - 1 = \Big(\frac{2}{n}\sum_{i=1}^{n} H_i\Big) - 1 = \frac{2(n+1)}{n} H_n - 3. \qquad \square$$

This quantity is the same as that derived by Hibbard [13]. However, he assumed that the keys were equally probable, and our result holds for any set of probabilities as long as they are randomly assigned to the keys.

COROLLARY 6.1. *For any set of n probabilities $p_1, \cdots, p_n$ that are randomly assigned to n keys, the expected cost of the monotonic tree is less than or equal to the expected cost of a random tree.*

Proof. Since the monotonic tree cost is $2\sum_{i=1}^{n} H_i p_i - 1$ and $H_1 < H_2 < \cdots < H_n$, clearly the maximum cost for all sets of probabilities with $p_1 \geqq p_2 \geqq \cdots \geqq p_n$ will be given by $p_i = 1/n$. Substituting $p_i = 1/n$ into the formula for the monotonic tree cost gives $(2(n+1)/n)H_n - 3$, the random tree cost. $\square$

We compare these three costs for several distributions. The first is the geometric distribution, $p_i = r^i/R$, $r < 1$, $1 \leqq i \leqq n$, where $R = (r - r^{n+1})/(1-r)$. Substituting into the formula for the monotonic tree cost gives

$$\frac{2}{r - r^{n+1}}\Big[\sum_{j=1}^{n} \frac{r^j}{j} - r^{n+1}\sum_{j=1}^{n} \frac{1}{j}\Big] - 1.$$

If n is large, this is approximately $(2/r)\ln(1/(1-r)) - 1$, a constant, as compared with $(2(n+1)/n)H_n - 3 \approx 2 \ln n$ for a random tree. Thus, the rule provides substantial savings over a random tree.

The entropy for this distribution is

$$H = \log \frac{1}{1-r} - \frac{r}{1-r} \log r.$$

Table 6.1 compares COST_{OPT} with COST_{MON}. Note that for small r, the distribution approaches $p_1 = 1, p_i = 0$ for $i > 1$, and $\text{COST}_{\text{OPT}} \approx \text{COST}_{\text{MON}}$. As $r \to 1$, the dis-

TABLE 6.1
A Comparison of COST_{OPT} *and* COST_{MON}.

r	COST_{OPT}	COST_{MON}	% Increase
0.3	1.376	1.378	0.12%
0.4	1.540	1.554	0.94%
0.5	1.766	1.773	0.39%
0.6	2.020	2.054	1.72%
0.7	2.331	2.440	4.67%
0.8	2.746	3.024	10.10%
0.9	3.577	4.117	15.10%

Trees with 50 nodes were considered. COST_{OPT} was approximated by constructing the optimal tree for each of twenty randomly chosen assignments of probabilities to keys and averaging the costs.

tribution approaches the uniform distribution and $\text{COST}_{\text{OPT}}/\text{COST}_{\text{MON}} \approx 2 \ln 2$. Note, however, that even if $r = .9$, COST_{MON} is still within 15% of the optimum.

The second distribution we consider is Zipf's Law. Here,

$$\text{COST}_{\text{MON}} = H_n - \frac{H_n^{(2)}}{H_n} - 1 \approx \ln n$$

where

$$H_n^{(2)} = \sum_{i=1}^{n} \frac{1}{i^2} < \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}.$$

To calculate COST_{OPT}, we first find H. We have

$$H = \log H_n + \frac{1}{H_n} \sum_{i=1}^{n} \frac{\ln i}{i(\ln 2)}.$$

Using the approximations $\sum_{i=1}^{n} \ln i/i \approx (\ln n)^2/2$ and $H_n \approx \ln n$ gives

$$\text{COST}_{\text{OPT}} \approx \log \ln n + \frac{(\ln n)^2}{2(\ln 2)(\ln n)} \approx \frac{\ln n}{2(\ln 2)}.$$

Therefore $\text{COST}_{\text{MON}} \approx (2 \ln 2)\ \text{COST}_{\text{OPT}}$ for large n, which is to be expected since $H \to \infty$ as $n \to \infty$. However, COST_{MON} is approximately half of $(2(n+1)/n)H_n - 3$, the cost of a random tree. Table 6.2 compares these costs for Zipf's Law and $n = 100$. Here, the expected cost of the monotonic tree is only 15% greater than the optimum. Thus,

TABLE 6.2
The performance of monotonic trees.

	English letters	Zipf's law					
		#1	#2	#3	#4	#5	Average
Random cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
Monotonic tree cost (Exact)	3.77	4.91	4.18	5.32	4.68	4.20	4.66
Increase over optimal	13.6%	19.7%	6.5%	27.9%	15.1%	6.0%	15.1%

The distributions considered were the probabilities for the English letters and five others that were generated by choosing a random ordering of 100 keys whose probability were given by Zipf's Law.

for both distributions, the expected cost of the monotonic tree is significantly smaller than that of the random tree.

We now consider a second method of selecting the key probabilities which has been studied by Nievergelt and Wong [14]. Here we are given a probability density, $f(x)$, and the key probabilities are chosen with respect to that density. We drop the requirement that our choices must sum to one, so instead of probabilities, we consider key *weights*. The cost of a tree is now $\sum_{i=1}^{n} w_i l_i$ where w_i is the weight of k_i. A theorem by Nievergelt and Wong [14] computes the expected costs of the optimal and random trees for this case.

THEOREM 6.6 (Nievergelt and Wong [14]). *If n key weights are independently selected from a density function $f(x)$ with finite mean μ, the expected cost of the optimal tree equals $\mu n \log n + O(n)$, and the expected cost of a random tree equals $(2 \ln 2)\mu n \log n + O(n)$.*

Nievergelt and Wong found the cost of the monotonic tree to be $(2 \ln 2)\mu n \log n + O(n)$ for the uniform distribution and conjectured this to be the cost for all distributions. We first derive a general form for the cost, then prove this conjecture.

THEOREM 6.7. *If n key weights are independently chosen from a density function $f(x)$ with finite mean μ, the expected cost of the monotonic tree is*

$$2\mu\left[nH_{n-1} - \left(\frac{n}{2} - 1\right)\right] - 2\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)F(y)^i \, dy$$

where $F(x) = \int_{-\infty}^{x} f(y)\, dy$.

Proof. Let w_i be the weight chosen for k_i and

$$A_{ji} = \begin{cases} 1 & \text{if } k_j \text{ is an ancestor of } k_i, \\ 0 & \text{if not.} \end{cases}$$

(Note that w_i and A_{ji} are *not* independent.) Since the level of k_i equals $1 + \sum_{j \neq i} A_{ji}$,

$$E(\text{Cost}) = \sum_{i=1}^{n} E\left(w_i\left(1 + \sum_{j \neq i} A_{ji}\right)\right) = n\mu + \sum_{i=1}^{n} \sum_{j \neq i} E(w_i A_{ji}).$$

To determine $E(w_i A_{ji}) = \int_{-\infty}^{\infty} y \text{ Prob } (w_i A_{ji} = y)\, dy$, note that A_{ji} can only be 0 or 1, and if $A_{ji} = 0$ the only y having nonzero probability is $y = 0$. Since this will be multiplied by $y = 0$, the case with $A_{ji} = 0$ can be ignored and

$$E(w_i A_{ji}) = \int_{-\infty}^{\infty} y \text{ Prob } (w_i = y \text{ and } A_{ji} = 1)\, dy.$$

To determine Prob $(w_i = y$ and $A_{ji} = 1)$ note that k_j will be an ancestor of k_i if and only if $w_j > w_i$ and w_j is greater than the weight of each of the $|i-j|-1 = m$ keys between k_i and k_j in the ordering on the keys. The probability that $w_i = y$ is $f(y)\, dy$. We then chose an $x \geqq y$ for w_j. Any specific x is chosen with probability $f(x)\, dx$. For this x, we must chose the $|i-j|-1 = m$ keys between k_i and k_j to have weight less than or equal to x. The probability for this is $F(x)^m$. The product of these is then integrated over $x \geqq y$, giving

$$\begin{aligned} \text{Prob } (w_i = y \text{ and } A_{ji} = 1) &= \int_{y}^{\infty} f(y)f(x)F(x)^m \, dx\, dy \\ &= f(y)\frac{1 - F(y)^{m+1}}{m+1}, \end{aligned}$$

since $dF(x)/dx = f(x)$ and $F(\infty) = 1$.

Then

$$E(w_i A_{ji}) = \int_{-\infty}^{\infty} yf(y)\left[\frac{1-F(y)^{m+1}}{m+1}\right] dy.$$

Note that this quantity depends only on m, and not the values of i and j. Since there are $2(n-m-1)$ distinct ordered (i, j) pairs having a given value of m,

$$E(\text{Cost}) = n\mu + \sum_{m=0}^{n-2} 2(n-m-1) \int_{-\infty}^{\infty} yf(y)\left[\frac{1-F(y)^{m+1}}{m+1}\right] dy$$

$$= n\mu + 2\sum_{m=0}^{n-2} \frac{n-(m+1)}{m+1} \int_{-\infty}^{\infty} yf(y)\, dy - 2\sum_{m=0}^{n-2} \frac{n-m-1}{m+1} \int_{-\infty}^{\infty} yf(y)F(y)^{m+1}\, dy$$

$$= 2\mu\left[nH_{n-1} - \left(\frac{n}{2}-1\right)\right] - 2\sum_{m=1}^{n-1} \frac{n-m}{m} \int_{-\infty}^{\infty} yf(y)F(y)^m\, dy. \quad \square$$

To prove this cost is asymptotically $(2 \ln 2)\, \mu n \log n$, we need the following lemma.

LEMMA 6.1. *For any density function f with finite mean μ,*

$$\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)F^i(y)\, dy = o(n \log n).$$

Proof. First note that for any $i \geqq 1$,

$$|yf(y)F^i(y)| \leqq |yf(y)|.$$

Hence Lebesque's Dominated Convergence Theorem (see [15]) applies and we have

$$\lim_{i\to\infty} \int_{-\infty}^{\infty} yf(y)F^i(y)\, dy = \int_{-\infty}^{\infty} yf(y) \lim_{i\to\infty} F^i(y)\, dy = 0$$

since

$$\lim_{i\to\infty} F^i(y) = 0 \text{ if } F(y) < 1 \quad \text{and} \quad f(y) = 0 \text{ if } F(y) = 1.$$

Now,

$$\sum_{i=1}^{n-1} \frac{n-i}{i} \int_{-\infty}^{\infty} yf(y)F^i(y)\, dy < n \sum_{i=1}^{n-1} \frac{1}{i} \int_{-\infty}^{\infty} yf(y)F^i(y)\, dy. \tag{6.5}$$

We now choose N such

$$\int_{-\infty}^{\infty} yf(y)F^i(y)\, dy < \varepsilon \quad \text{for } i \geqq N.$$

Putting this in (6.5) gives that the right-hand side of (6.5)

$$< n \sum_{i=1}^{N-1} \frac{1}{i} \int_{-\infty}^{\infty} yf(y)F^i(y)\, dy + n \sum_{i=N}^{n-1} \frac{\varepsilon}{i} < n \sum_{i=1}^{N-1} \frac{\mu}{i} + n \sum_{i=N}^{n-1} \frac{\varepsilon}{i}.$$

Since $H_x < \ln x + 1$, we have the above expression

$$< n(\ln (N-1) + 1)\mu + n\varepsilon(\ln (n-1) + 1).$$

Therefore

$$\frac{n(\ln(N-1)+1)\mu + n\varepsilon(\ln(n-1)+1)}{n\log n}$$
$$= \frac{(\ln(N-1)+1)\mu}{\log n} + \frac{\varepsilon(\ln(n-1)+1)}{(\log e)(\ln n)}$$
$$= \frac{(\ln(N-1)+1)\mu}{\log n} + \frac{\varepsilon}{\log e} + \frac{\varepsilon}{(\log e)(\ln n)}.$$

We can make the first and third terms arbitrarily small (say, less than ε) by choosing n sufficiently large. Therefore,

$$\frac{\sum_{i=1}^{n-1}((n-i)/i)\int_{-\infty}^{\infty} yf(y)F^i(y)\,dy}{n\log n} < \left(2+\frac{1}{\log e}\right)\varepsilon$$

for $n > N'$. Therefore the limit of this ratio is zero as $n \to \infty$ and the lemma is proved. □

THEOREM 6.8. *If n keys have their weights chosen according to* any *density function with finite mean* μ, *the expected cost of a monotonic tree is* $(2\ln 2)\mu n\log n + O(n)$, *asymptotically equal to the cost of a random tree.*

Proof. The cost of a monotonic tree is

$$2\mu\left[nH_{n-1} - \left(\frac{n}{2}-1\right)\right] - 2\sum_{i=1}^{n-i}\frac{n-i}{i}\int_{-\infty}^{\infty} yf(y)F^i(y)\,dy.$$

Since the first term is asymptotically equal to $(2\ln 2)\mu n\log n$ and the second is $o(n\log n)$, the leading term in the asymptotic cost is $(2\ln 2)\mu n\log n$.

We now show that the cost of a monotonic tree is less than or equal to that of a random tree, proving that the cost of a monotonic tree equals $(2\ln 2)\mu n\log n + O(n)$. The method we are using to select key weights choses n weights independently from a density function. An equivalent method first selects a *set* of n weights from an n-dimensional density function. (This function can be constructed so that the probability of choosing a given set equals the probability of obtaining it (in any order) from n selections from the original function.) We then choose a permutation of the set. By Corollary 6.1, the expected cost of a monotonic tree is less than or equal to that of a random tree for any such set, proving the theorem. □

The majority of the results concerning monotonic trees are discouraging. This method performs well only when we are guaranteed that the key probabilities will differ significantly from the uniform distribution (i.e., have low entropy). This is not the case in the situation described by Nievergelt and Wong (Bayer [19] has shown that the expected entropy of a randomly chosen probability distribution is $\log n - \ln 2$, which is nearly $\log n$, the maximum entropy) and the performance is asymptotically the same as randomly built trees. This conclusion is substantiated by simulations run by Walker and Gottlieb [16] which showed the monotonic tree rule to perform very poorly.

7. Single and double rotation rules. The previous methods have used the fact that a rotation moves a certain node up in the tree, ignoring the fact that it also moves two (possibly large) subtrees. The following rules achieve superior performance by considering these subtrees in deciding whether to apply a notation. The *limited single rotation rule* applies a rotation at a node on the search path for the requested key if the number of accesses to nodes which will be moved up by the rotation exceeds the number to those which will be moved down. Note that this will reduce our estimate (based on the

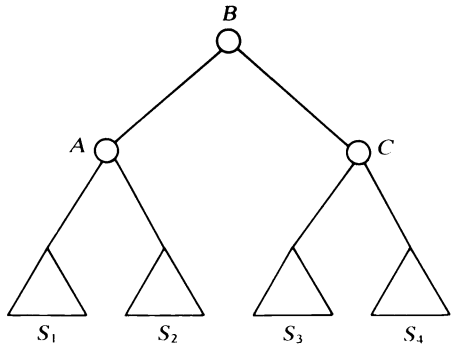


FIG. 7.1

accesses we have seen) of the expected cost of the tree. For example in Fig. 7.1, we perform a rotation to promote A if $w(A)+w(S_1)>w(B)+w(C)+w(S_3)+w(S_4)$. We promote C if $w(C)+w(S_4)>w(B)+w(A)+w(S_1)+w(S_2)$. Here, $w(A)$, $w(B)$ and $w(C)$ are the number of times A, B and C respectively, have been requested, and $w(S_i)$ is the number of times any node in S_i have been requested. Since all this information is available at node B, the rotations can be efficiently done during the search for the requested key.

Note that a rotation at one node may cause other nodes to become "unbalanced"; after promoting node B in Fig. 7.1, rotations may be possible at both A and C. The *total single rotation rule* applies additional rotations to correct all "imbalances" caused by rotations along the search path.

TABLE 7.1
The performance of the limited single rotation (LSR) and total single rotation (TSR) rules.

	English letters	Zipf's Law #1	#2	#3	#4	#5	Average
Random cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
LSR cost	3.44	4.33	4.14	4.46	4.28	4.20	4.28
Increase over optimum	3.55%	5.46%	5.31%	7.29%	5.42%	5.98%	5.89%
Average number of rotations/request	0.111	0.199	0.204	0.199	0.197	0.200	0.200
Average over the last 100 requests		0.033	0.041	0.040	0.039	0.034	0.038
TSR cost	3.41	4.33	4.11	4.41	4.22	4.17	4.25
Increase over optimum	2.93%	5.57%	4.57%	6.02%	3.92%	5.27%	5.07%
Average number of rotations/request	0.118	0.220	0.219	0.217	0.209	0.213	0.215
Average over the last 100 rotations		0.040	0.048	0.044	0.036	0.036	0.041

A simulation was run to determine the cost of various rules. Fifty trees were randomly generated, and the cost and other statistics were recorded after 500 requests. The probability distributions we considered were the English letters and five others that were generated by choosing a random ordering of 100 keys whose probabilities were given by Zipf's Law.

Table 7.1 compares these two rules. Similar simulations have been run by Baer [17]. The total single rotation rule has a slightly lower cost and does surprisingly few more rotations. However, there is much more overhead associated with a rotation in the total rotation rule. Since imbalances can propogate throughout the tree, either a pointer to a node's father must be maintained or we must stack the nodes encountered during the search for the requested key. Table 7.1 also shows how much work the rules require after many requests; after an initial period to "organize" the tree, both rules require very few rotations.

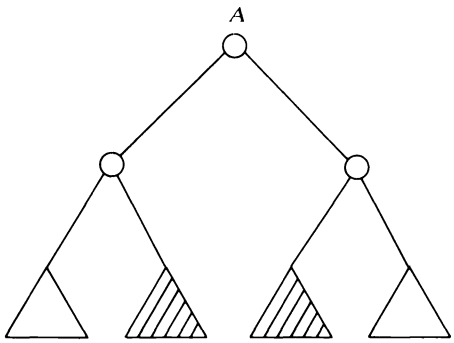


FIG. 7.2. *The inside subtrees of node A are darkened.*

A weakness of these rules is that they do not consider the "inside" subtrees (the right subtree of a node's left son, or the left subtree of its right son, see Fig. 7.2). A rotation can promote either "outside" subtree, but the inside subtrees remain at the same level. This can lead to very poor trees that are still "stable" in the sense that no rotations can be performed. Figure 7.3 shows an example. This tree is stable as long as the weight of a node is less than or equal to that of its father.

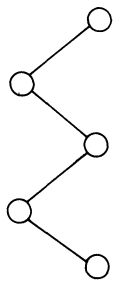


FIG. 7.3

While the worst case performance for such a distribution is quite bad, a simulation suggests the average case is acceptable. Consider probability distribution where $p_1 = \frac{50}{1275}, p_2 = \frac{48}{1275}, \cdots, p_{25} = \frac{2}{1275}, p_{26} = \frac{1}{1275}, p_{27} = \frac{3}{1275}, \cdots, p_{50} = \frac{49}{1275}$. The tree shown in Fig. 7.3 is stable for this probability distribution. Yet, after 500 requests the limited rotation rule reduced the cost to 4.7593, a mere 3.06% increase over the optimal cost of 4.6180.

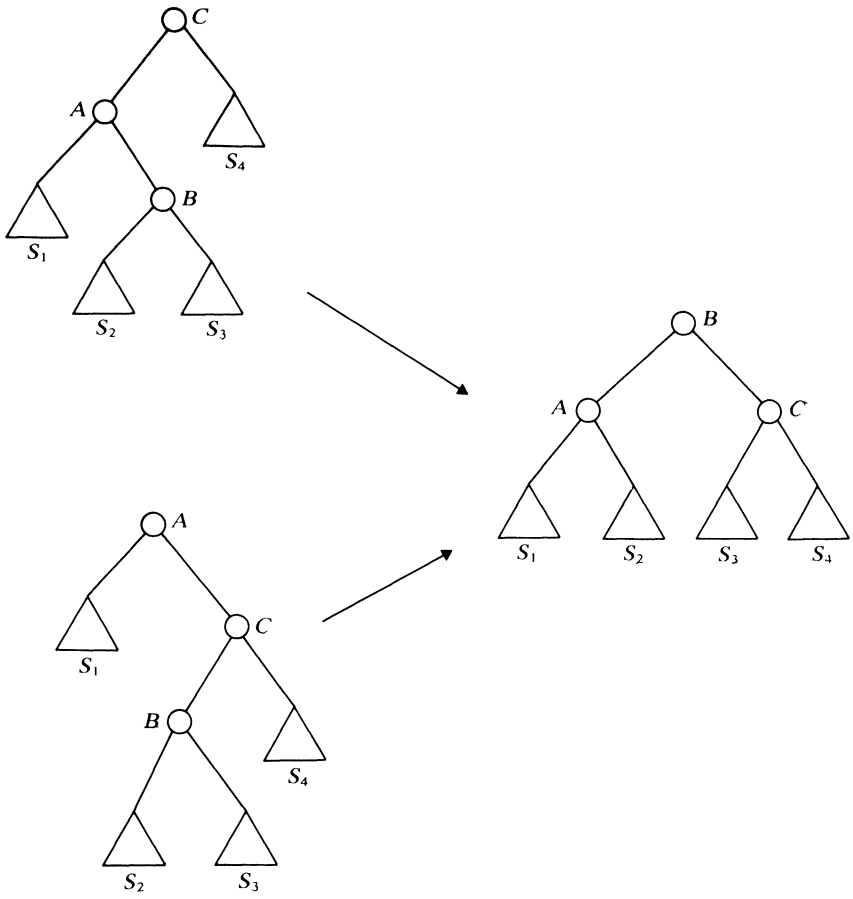


FIG. 7.4. *The two double rotations. In either case, key B has been requested.*

We can get a superior rule by also using the two transformations (called *double rotations*) shown in Fig. 7.4. These allow the promotion of inside subtrees. The *double rotation rule* applies a rotation at any node on the search path if it reduces the cost of the tree. Both single and double rotations are considered. The criterion for applying a single rotation has already been discussed. The condition for applying the "upper" double rotation shown in Fig. 7.4 is $2w(B)+w(S_2)+w(S_3)>w(C)+w(S_4)$ and that for the "lower" double rotation is similar.

Table 7.2 shows the cost of the double rotation rule, which is within 3.84% of the optimum, and the total number of rotation required per request (counting both single and double rotations) is very close to the averages for the limited rotation rule and the total rotation rule.

However, after many requests, fewer rotations are required than for either single rotation rule. In fact, the average over the last 100 requests was .027 single rotation (one every 36 requests) and .008 double rotations (one every 129 requests).

More complicated rules are possible. Bruno and Coffman [18] have considered an extension of the double rotation rule that can promote a node any number of levels by using a sequence of rotations. They, however, were concerned with an algorithm to build a nearly optimal tree from a set of known key probabilities and used this set of transformations to reduce the cost of the initial tree. Every final tree in their simulation was within 5 percent of the optimum, and the average was within 2.6%.

TABLE 7.2
The performance of the double rotation (DR) rule.

	English letters	Zipf's Law #1	#2	#3	#4	#5	Average
Random cost	5.15	7.26	7.50	7.27	7.33	7.63	7.40
Optimal cost	3.32	4.10	3.93	4.16	4.06	3.96	4.04
DR cost	3.40	4.29	4.06	4.32	4.23	4.09	4.20
Increase over optimum	2.35%	4.62%	3.45%	3.80%	4.11%	3.22%	3.84%
Average number of single rotations/request	0.074	0.129	0.129	0.127	0.126	0.126	0.127
Average over last 100 requests		0.029	0.027	0.025	0.028	0.028	0.027
Average numbers of double rotations/request	0.037	0.082	0.081	0.082	0.079	0.080	0.081
Average over last 100 requests		0.007	0.007	0.008	0.007	0.009	0.008

See Table 7.1 for explanation

This suggests further rules, where we consider promoting the requested node i levels for $i = 1, 2, \cdots, k$, where k is a parameter of the rule. Note that the single rotation rules have $k = 1$, and the double rotation rule has $k = 2$. Increasing k will increase the work the rule must do, but will result in decreased retrieval times. The results of Bruno and Coffman suggest that the retrieval time will not be greatly improved by increasing k beyond 2, while the increase in the complexity of the algorithm to execute the rule would be substantial.

The double rotation rule appears to be the best of these three rules. It has better performance than the limited single rotation rule (the cost averaged within 3.84% of the optimum). In addition, fewer rotations per request are required; using the double rotations allows the tree to be altered more efficiently.

8. Conclusion. We first discussed several heuristics for dynamically organizing linked lists. Analysis of the asymptotic behavior of the move to front rule and transposition rule has been done by Rivest [1], who showed the transposition to be always superior. We considered how *quickly* rules organize the data structure and found the move to front rule superior in this respect. A hybrid rule was then defined which had the fast convergence of the move to front rule and the low asymptotic cost of the transposition rule.

If the data structure has space for a count field to be associated with each key, the frequency count rule, which is optimal, can be used. However, since this rule eventually overflows any fixed size field, we considered rules that use a bounded amount of storage. The best of these is the limited difference rule. Its performance is not optimal, but approaches the optimum as the upper bound on the difference fields is increased. This upper bound need not be too large; even for small bounds, the performance is nearly optimal.

Other rules that use counters are the wait c, move and clear rules and the wait c and move rules. Both classes are an asymptotic improvement over the corresponding permutation rule. However, their asymptotic cost is greater than that of the limited difference rule, and their convergence is very slow.

Next, we examined several methods for dynamically organizing binary search trees. The first was the *monotonic tree rule*. The worst case performance of this rule was

already known to be very poor. To evaluate this rule in an average case, we considered two different methods of randomly choosing probabilities for the keys.

The first method assumes we are given a set of n probabilities and randomly assign them to the keys. For this case, the expected cost of a monotonic tree (COST_{MON}) was shown to satisfy

$$(2 \ln 2)H - f(n) \leqq \text{COST}_{\text{MON}} \leqq (2 \ln 2)H + 1$$

where $f(n)$ is bounded by a small constant and H is the entropy of the probability distribution. This shows that the monotonic tree rule performs well only when the probability distribution has low entropy. Zipf's Law and the geometric distribution were considered. For both, the monotonic tree rule obtained significant decreases in cost over a randomly built tree.

The second method assumes that the weights of the keys are chosen according to an arbitrary density function. In this case, the performance was shown to be asymptotically the same as a random tree. This agrees with the results from the first method; entropy of a randomly chosen distribution of n probabilities is very high [19] ($\log n - \log 2$, nearly $\log n$, the maximum).

We then discussed rules with lower cost than the monotonic tree rule. Simulations showed that the cost of the *limited single rotation rule* averaged within 5.89 percent of the optimum. The *total single rotation rule* reduced the average cost to 5.07 percent of the optimum, and the *double rotation rule* averaged approximately 3.84 percent of the optimum. Though the double rotation rule must check for both single and double rotations, it averaged one rotation every 36 requests and one double rotation every 129 after the initial period when the tree is being "organized". Compared with the limited single rotation rule, the double rotation rule does less work after the initial period. It then appears to be the best choice of the counter rules.

Acknowledgment. I wish to thank E. M. Reingold for his help in researching and writing this article and D. L. Burkholder for the proof of Lemma 6.1.

REFERENCES

[1] R. L. RIVEST, *On self organizing sequential search heuristics*, Comm. ACM, 19 (1976), pp. 63–67.
[2] P. J. BURVILLE AND J. F. C. KINGMAN, *On a model for storage and search*, J. Appl. Probability, 10 (1973), pp. 697–701.
[3] W. J. HENDRICKS, *The stationary distribution of an interesting Markov chain*, Ibid., 9 (1972), pp. 231–233.
[4] ———, *An extension of a theorem concerning an interesting Markov chain*, Ibid., 10 (1973), pp. 886–890.
[5] J. MCCABE, *On serial files with relocatable records*, Operations Res., 12 (1965), pp. 609–618.
[6] D. E. KNUTH, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA, 1973.
[7] J. R. BITNER, *Heuristics that dynamically alter data structures to reduce their access time*, University of Illinois Report UIUCDCS-R-76-818, July, 1976 (Ph.D. thesis).
[8] W. FELLER, *An Introduction to Probability Theory and its Application*, John Wiley, New York, 1968.
[9] D. E. KNUTH, *The Art of Computer Programming*, vol. 1, Addison-Wesley, Reading, MA, 1973.
[10] B. ALLEN, AND I. MUNRO, *Self-Organizing binary search trees*, 17th Annual Symposium on Foundations of Computer Science (1976), pp. 166–172.
[11] J. W. J. WILLIAMS, *Algorithm 232—Heapsort.*, Comm. ACM, 7 (1964), pp. 347–348.
[12] K. MELHORNE, *Nearly optimal binary search trees*, Acta Informat., 5 (1975), pp. 287–295.
[13] T. N. HIBBARD, *Some combinatorial properties of certain trees with application to searching and sorting*, J. Assoc. Comput. Mach., 9 (1962), pp. 16–17.
[14] J. NIEVERGELT AND C. K. WONG, *On binary search trees*, Information Processing 71, vol. 1, North-Holland, Amsterdam, (1971), pp. 91–98.
[15] W. RUDIN, *Principles of Mathematical Analysis*, McGraw-Hill, New York, 1976.
[16] W. A. WALKER AND C. C. GOTTLIEB, *A top-down algorithm for constructing nearly optimal*

lexicographic trees, Graph Theory and Computing, R. C. Reid, ed., Academic Press, New York, 1972, pp. 303–323.

[17] J. L. BAER, *Weight balanced trees*, National Computer Conference 1975, pp. 467–472.

[18] J. BRUNO AND E. G. COFFMAN, *Nearly optimal binary search trees*, Proc. IFIP Congress 71 (1971).

[19] P. J. BAYER, *Improved bounds on the costs of optimal and balanced binary search trees*, MAC Technical Memo–69, November, 1975.

SIAM J. COMPUT.
Vol. 8, No. 1, February 1979

0097-5397/79/0801-0008 $01.00/0

TOTAL ORDERING PROBLEM*

J. OPATRNY†

Abstract. The problem of finding a total ordering of a finite set satisfying a given set of in-between restrictions is considered. It is shown that the problem is *NP*-complete.

Key words. algorithms, computational complexity, total ordering, *NP*-completeness

1. Introduction. In the design of circuits it can be desirable to arrange input and output pins in such a way that a particular pin is located between two specific pins. The problem of arranging pins along an edge can be mathematically formulated as follows: given a finite set of elements S and a set of ordered triples $R \subseteq S \times S \times S$ (the set of "in-between" restrictions), does there exist a total ordering of S such that if $(a, b, c) \in R$ then either $a < b$ and $b < c$ or $c < b$ and $b < a$?

In this paper the time complexity of the problem of finding a total ordering of a set S satisfying a given set of in-between restrictions is investigated and it is shown that the problem is *NP*-complete. It implies that, unless $P = NP$, the problem of finding a total ordering is inherently hard. The time complexity of the above total ordering problem has been an open problem proposed by R. Karp.

2. Preliminaries. In this section the basic definitions are presented and the Total Ordering Problem is stated.

DEFINITION. A *partial ordering* of a set S is a relation between elements of S, denoted by $<$, satisfying the following properties for any elements a, b, c in S:

i) If $a < b$ and $b < c$ then $a < c$.

ii) If $a < b$ then $b \not< a$.

iii) $a \not< a$.

A partial ordering of S is called a *total ordering* of S if for any two distinct elements a, b in S either $a < b$ or $b < a$.

DEFINITION. *The Total Ordering Problem* (TOP) is the following: given a finite set S and a set of ordered triples $R \subseteq S \times S \times S$, determine whether there exists a total ordering of S such that for any element (a, b, c) in R either $a < b$, $b < c$ or $c < b$, $b < a$ (we say that such an ordering satisfies R). A solution for TOP would be an algorithm which takes an instance (S, R) of TOP and outputs true if and only if there exists a total ordering of S satisfying R.

DEFINITION. *NP* is the set of all languages for which there is a polynomial time bounded nondeterministic recognition algorithm.

A language L_0 is *NP-complete* if L_0 is in *NP* and existence of a polynomial deterministic algorithm to recognize L_0 implies that for every L in *NP* we can effectively find a polynomial deterministic algorithm to recognize L.

DEFINITION. A language L is *polynomially reducible to language* L_0 if there is a deterministic polynomial algorithm which will convert each string a in the alphabet of L into a string b in the alphabet of L_0 such that $a \in L$ if and only if $b \in L_0$.

DEFINITION. A *hypergraph* is an ordered pair $H = (V, E)$ such that V is a nonempty finite set and E is a nonempty system of subsets of V. The elements of V are called the edges of H. The rank of hypergraph H is $\max \{\|e\|: e \in E\}$ where $\|e\|$ denotes the number of elements in e.

* Received by the editors September 15, 1977.

† Computer Science Department, Concordia University, Montreal. This research was done at the University of Alberta with the support of The National Research Council of Canada.

The 2-*colorability Problem* is the following: given a hypergraph $H = (V, E)$ of rank 3, determine whether there exist sets V_B, V_R of blue, red color vertices respectively such that $V_B \cap V_R = \emptyset$ and $V_B \cap e \neq \emptyset$, $V_R \cap e \neq \emptyset$ for any edge e in E.

DEFINITION. *The 3-satisfiability Problem* is the following:

Given a Boolean expression B in conjunctive normal form with at most 3 literals per clause, determine whether there exists an assignment of 1's and 0's to variables in B such that for any variable x the value of $x \cdot \bar{x} = 0$, $x + \bar{x} = 1$ (where $\bar{x}$ is the complement of x) and the value of B is 1. (We say that B is satisfiable if there exists such an assignment.)

3-satisfiability is *NP*-complete [4]. It is implicit in [6] that 2-colorability of hypergraphs of rank 3 is *NP*-complete.

3. Results. Consider first the following *Simple Total Ordering Problem* (Simple TOP). Given a finite set S and a set of ordered triples $R \subseteq S \times S \times S$, determine whether there exists a total ordering of S such that for any element (a, b, c) in R, $a < b$ and $b < c$.

Therefore, given an instance (S, R) of simple TOP it is known for any (a, b, c) in R that $a < b < c$ while for any (a, b, c) in R in an instance of TOP there are two possibilities: either $a < b < c$ or $c < b < a$. Since an element of R in an instance (S, R) defines precedence requirements on elements of S, Simple Ordering Problem can be reduced to Topological Sorting Problem [5, p. 262].

LEMMA 1. *An instance (S, R) of Simple TOP can be solved in time $O(\|S\| + \|R\|)$.*

Proof. Let (S, R) be an instance of Simple Total Ordering Problem. Let $R' = \{(a, b), (b, c) : (a, b, c) \in R\}$. Then each element of R' defines a partial ordering of S and a total ordering of S satisfying R' (or its existence) can be found using topological sort algorithm in time proportional to $(\|S\| + \|R\|)$ [5, p. 262]. □

Thus, if TOP is hard, the hard part of finding a solution to an instance (S, R) of TOP would be to find for each (a, b, c) in R the "orientation" of it (i.e. whether $a < b < c$ or $c < b < a$) such that the resulting instance of Simple TOP has a solution. It will be shown the problem of 2-colorability of hypergraphs can be polynomially reduced to TOP. In the reduction we will associate with each edge of a hypergraph in-between restrictions such that the problem of assignment of a color to a vertex in the edge will be equivalent to assignment of an orientation to the corresponding in-between restrictions.

LEMMA 2. *Given a hypergraph H of rank 3 with n edges we can construct in $O(n)$ steps S, R with the following property. H is 2-colorable if and only if there exists a total ordering of S satisfying R.*

Proof. Let $H = (V, E)$ be a hypergraph of rank 3,

$$V = \{s_1, s_2, \cdots, s_n\}, \qquad E = E_1 \cup E_2,$$

$$E_1 = \{(a_i, b_i, c_i) : a_i, b_i, c_i \in V, 1 \leqq i \leqq j\},$$

$$E_2 = \{(d_i, e_i) : d_i, e_i \in V, 1 \leqq i \leqq m\}.$$

Let $X, Y_1, Y_2, \cdots, Y_j$ be symbols not in V. Construct set R as follows:

(i) (a_k, Y_k, b_k), (Y_k, X, c_k) are in R for each k, $1 \leqq k \leqq j$.

(ii) (d_k, X, e_k) are in R for each k, $1 \leqq k \leqq m$. Let $S = V \cup \{X, Y_1, Y_2, \cdots, Y_J\}$. It will be shown that there exists a total ordering of S satisfying R if and only if hypergraph H is 2-colorable.

a) Assume that H is 2-colorable. Let $V_B \subseteq V$, $V_R \subseteq V$ be the set of vertices of blue, red color respectively, such that $V_B \cap V_R = \emptyset$, $V_B \cap e \neq \emptyset$, $V_R \cap e \neq \emptyset$ for every edge e in E.

Let f be a function on S defined as follows:

(i) $f(X)=0$.

(ii) if vertex $s_i \in V_R$, where $1 \leqq i \leqq n$, then $f(s_i)=i$; else $f(s_i)=-i$

(iii) for every i, $1 \leqq i \leqq j$, $f(Y_i)$ is defined as follows: if sign $(f(a_i))=$ sign $(f(b_i))$ then

$$f(Y_i)=\min\{f(a_i), f(b_i)\}+1/(i+1)$$

else

$$f(Y_i)=-\text{sign}\,(f(c_i))/(i+1).$$

Clearly, f is a one-to-one function that assigns a rational number to each element of S. If each edge contains vertices of both colors, then the assignment of real numbers of Y_i, $1 \leqq i \leqq j$ satisfies all restrictions in R and, therefore, f defines a total ordering of S satisfying R.

b) Assume that an instance (S, R) of TOP has a solution. Assign colors to vertices in V as follows:

If $s < X$ in a solution of (S, R) then vertex S is in V_B else S is in V_R.

Since it is not possible for any i, $1 \leqq i \leqq j$ that

$$a_i < X \quad \text{and} \quad b_i < X \quad \text{and} \quad c_i < X, \quad \text{or}$$

$$a_i > X \quad \text{and} \quad b_i > X \quad \text{and} \quad c_i > X$$

and similarly it is not possible for any i, $1 \leqq i \leqq m$ that

$$d_i < X \quad \text{and} \quad e_i < X, \quad \text{or}$$

$$d_i > X \quad \text{and} \quad e_i > X$$

no edge of E is a subset of V_R or V_B.

Therefore, H is 2-colorable.

THEOREM. *TOP is NP-complete.*

Proof. Consider the following nondeterministic algorithm to solve a given instance (S, R) of TOP.

(i) Assign nondeterministically an orientation to each element of R and thus transform the problem into an instance (S, R') of Simple TOP.

(ii) Use algorithm from Lemma 1 to solve (S, R').

Clearly, the nondeterministic algorithm above solves (S, R) in polynomial time. Therefore, TOP is in *NP*.

By Lemma 2, 2-colorability of hypergraphs of rank 3 is linearly reducible to TOP. Therefore, *NP*-completeness of 2-colorability problem [6] implies *NP*-completeness of TOP problem.

The reduction of 2-colorability of hypergraphs into TOP is linear and, furthermore, there is a simple correspondence between colors in a hypergraph and orientations of in-between restrictions. Similarly, 2-colorability of hypergraphs is useful in demonstration of *NP*-completeness of other problems [3], [6]. An important advantage of 2-colorability problem is the symmetry of both colors which is not the case of 1's and 0's in the 3-satisfiability problem.

To illustrate the difference we give below a reduction of 3-satisfiability into TOP: Given a Boolean expression

$$B=(a_1+b_1+c_1)\cdot(a_2+b_2+c_2)\cdot\cdots\cdot(a_j+b_j+c_j)$$
$$\cdot(d_1+e_1)\cdot(d_2+e_2)\cdot\cdots\cdot(d_m+e_m)$$

construct set R' as follows:

Let F, X, S_1, $S_2, \cdots, S_m$, V_1, $V_2, \cdots, V_j$, Y_1, $Y_2, \cdots, Y_j$, Z_1, $Z_2, \cdots, Z_j$, U_1, $U_2, \cdots, U_j$ be new symbols.

(i) (a_k, V_k, b_k), (V_k, X, Z_k), (U_k, Y_k, c_k), (Y_k, X, F), $(Z_k, X, U_k) \in R'$ for every k, $1 \leqq k \leqq j$.

(ii) (d_k, S_k, e_k), $(S_k, X, F) \in R'$ for every k, $1 \leqq k \leqq m$.

(iii) $(q, X, \bar{q}) \in R'$ for every variable q in B where $\bar{q}$ is the complement of q.

It can be shown similarly as in the proof of Lemma 2 that there exists a total ordering of symbols in R' satisfying R' if and only if Boolean expression B is satisfiable.

Set R' contains $5j+2m$ restrictions while set R from Lemma 2 contains $2j+m$ restrictions.

Acknowledgment. I would like to thank Dr. V. Chvatal for a simplification of the proof of the main lemma.

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] S. A. COOK, *The complexity of theorem-proving procedures*, Third Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151–158.

[3] V. CHVATAL AND G. THOMASSEN, *Distances in orientations of graphs*, Research report STAN-CS-75-511, Stanford University, Stanford, CA, 1975.

[4] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

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

[6] L. LOVASZ, *Covering and coloring of hypergraphs*, Proc. 4th S-E Conference, Combinationa, Graph Theory, and Computing, 1973, pp. 3–12.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0001 $01.00/0

LOWER BOUNDS ON SYNCHRONOUS COMBINATIONAL COMPLEXITY*

L. H. HARPER† AND J. E. SAVAGE‡

Abstract. Synchronous combinational complexity, a measure of the size of logic circuits without races, is investigated in this paper. The first author has presented a method for obtaining an $O(n \log n)$ lower bound to synchronous combinational complexity and has shown that this bound applies to "almost all" Boolean functions in n variables. However, he could not constructively exhibit functions to which the lower bound applied (although Wolfgang Paul did produce an example). In this paper we weaken and extend the hypothesis of the lower bound so that a larger class of functions satisfies it and apply it to the determinant and marriage functions of $GF(2)$.

Key words. complexity, logic circuits, synchronous circuits, determinant, marriage problem

1. Introduction. Combinational complexity or the circuit size of Boolean functions plays a fundamental role in theoretical computer science. It provides a lower bound to the time to compute functions on Turing machines [1], [2] and on the space-time product on simple general-purpose computers [1] so that a large combinational complexity implies that a function is computationally complex. In fact, we believe combinational complexity to be the most promising tool with which to show that *NP*-complete problems [3], [4] are of exponential complexity. Unfortunately, however, it has not been possible, except under certain special conditions, to derive nonlinear lower bounds to the combinational complexity of functions. Nevertheless, much of general interest has been learned about the subject, as seen in the survey article [5] and in the full account given in [6]. Exponential lower bounds have been derived for a few functions which have the ability to encode all Boolean functions over a slightly smaller set of variables [7], [8]. However, most research on the complexity of Boolean functions has concentrated on the development of techniques for bounding the combinational complexity of simple, explicitly defined functions. The marriage function [9] is an example of such functions and is investigated in this paper.

In search of improved methods for bounding combinational complexity, Harper [10] has studied synchronous combinational complexity, a measure that is related to combinational complexity but one that highlights the structure of circuits, as seen below. (Lupanov [11] has developed asymptotic bounds on the synchronous combinational complexity of the most complex Boolean functions on n variables.) Harper has shown in a nonconstructive manner that Boolean functions exist which have synchronous combinational complexity that is $\Omega(n \log n)$, where n is the number of variables of the function. Wolfgang Paul in an unpublished manuscript has demonstrated that a function which uses several levels of indirect addressing does satisfy Harper's condition for an $\Omega(n \log n)$ lower bound. In this paper we weaken and extend Harper's condition so that it can be applied to two interesting functions, the determinant and marriage functions modulo 2.

The paper has five sections. In § 2, we define synchronous and standard combinational complexity while in § 3, we derive the principal result, a lower bound to the synchronous combinational complexity of a function in terms of the number of its

* Received by the editors August 10, 1976, and in final revised form February 23, 1978.

† Department of Mathematics, University of California, Riverside, Riverside, California 92502. This work was supported in part by the National Science Foundation under Grant GJ 42907.

‡ Division of Engineering, Brown University, Providence, Rhode Island 02912. This work was supported in part by the National Science Foundation under Grants DCR72-03612 and MCS76-20023.

subfunctions. Applications are studied in the fourth section, and the last section is devoted to comments and conclusions.

2. Synchronous combinational complexity. An *n-logic circuit* (or simply a *circuit*) is a directed labeled, acyclic graph with node labels that are either Boolean variables from $\{x_1, \cdots, x_n\}$ or Boolean functions from a *basis* $\Omega = \{h_i: \{0,1\}^{n_i} \to \{0,1\}\}$ (see Savage [6]). A basis has *fan-in r* if $n_i \leqq r$ and for some i, $n_i = r$. Nodes that are labeled with variables are called *source nodes* and have no edges directed into them. The others are called *computation nodes* and have edges directed in and generally have edges directed out as well. To each node v we associate a Boolean function f_v. If v is a source node, f_v is a projection operator and otherwise it is defined recursively by the composition of its node label h_i with the n_i functions associated with nodes from which it has incoming edges. The *depth* of a node in a circuit is the length of (number of elements on) the longest path from that node to source nodes. A circuit is said to *compute* a function $f: \{0,1\}^n \to \{0,1\}^m$, where $f = (f_1, f_2, \cdots, f_m)$ and $f_i: \{0,1\}^n \to \{0,1\}$ if there exist nodes in the circuit whose associated functions are the functions $\{f_i\}$. The *combinational complexity* of f relative to a basis Ω, denoted $C_\Omega(f)$, is the minimum number of logic elements needed to compute it with a circuit over Ω. The depth of a circuit is the length of its longest path. The *delay complexity* of f relative to Ω, denoted by $D_\Omega(f)$, is the depth of the smallest depth circuit for f over Ω.

A logic circuit is *synchronous* (it is an *s-circuit*) if for each logic element the length of each path from that element to a source node is the same and the length of all paths from inputs to outputs are the same. If a circuit is not synchronous it can be made so by introducing delay elements. The *synchronous combinational complexity* (or *s-complexity*) of a Boolean function f relative to a basis Ω, denoted $C^s_\Omega(f)$, is the minimum number of logic and delay elements needed to compute it with an s-circuit. Source nodes in an s-circuit are said to be at *level* 0 and nodes with paths of length l from source nodes are said to be at *level l*. It is fairly easy to see that s-complexity is a crude measure of the area occupied by a circuit that is placed on a rectangular grid.

3. A lower bounding method. Let $f: \{0,1\}^n \to \{0,1\}^m$ and let $A \subseteq \{1, 2, \cdots, n\}$. Then, a *subfunction of f on A* is a function $f|^{x_j = c_j}_{j \notin A}$ for some $c_j \in \{0,1\}$, $j \notin A$. Let $\nu(f, A)$ *denote the number of distinct subfunctions of f on A* and let

$$\operatorname*{ave}_{|A|=a} \log_2 \nu(f, A)$$

denote the *average* of $\log_2 \nu(f, A)$ over subsets A of $\{1, 2, \cdots, n\}$ of cardinality a with a uniform distribution. A Boolean function $g: \{0,1\}^n \to \{0,1\}$ is dependent on variable x_i if there exist values for its remaining variables such that a change in x_i causes a change in g.

LEMMA 1. *Let* $g: \{0,1\}^n \to \{0,1\}$ *be dependent on variables with indices in B where* $b = |B|$. *Then,*

$$\operatorname*{ave}_{|A|=a} \log_2 \nu(g, A) \leqq \frac{1}{\binom{n}{a}} \sum_k \binom{b}{k}\binom{n-b}{a-k} 2^k \leqq \frac{1}{\left(1 - \dfrac{2ab}{n-a-b+1}\right)}$$

if $2ab/(n-a-b+1) < 1$.

Proof. There are 2^{2^r} distinct Boolean functions $h: \{0,1\}^r \to \{0,1\}$. Since a subfunction of g on A depends only on variables in $A \cap B$, $\log_2 \nu(g, A) \leqq 2^k$ where $k = |A \cap B|$. There are $\binom{n}{a}$ sets A of cardinality a and $\binom{b}{k}\binom{n-b}{a-k}$ which have k

elements in common with B. The first upper bound follows directly from these facts.

The ratio of two consecutive binomial coefficients is

$$\binom{c}{j+1}\Big/\binom{c}{j}=\frac{c-j}{j+1}$$

and the ratio of two terms in the sum is

$$R_k=2\frac{(b-k)}{(k+1)}\frac{(a-k)}{(n-a-b+1+k)}\leqq R_0=\frac{2ab}{(n-a-b+1)}.$$

It follows that the average is bounded above by

$$\frac{\binom{n-b}{a}}{\binom{n}{a}}\sum_{k=0}^{\infty}(R_0)^k\leqq\frac{1}{1-R_0}$$

since $R_0<1$. □

We are now prepared to derive the principal result of the paper, a lower bound to the synchronous combinational complexity of a function in terms of the number of its subfunctions.

THEOREM 1. *Let Ω be a basis of fan-in r and let $0<\delta<1$. If f: $\{0,1\}^n\to\{0,1\}^m$ and $D_\Omega(f)\geqq L=\lfloor\log_r\delta(n-a+1)/(2a+\delta)\rfloor$ then*

$$C_\Omega^s(f)\geqq(1-\delta)L\operatorname*{ave}_{|a|=a}\log_2\nu(f,A).$$

Proof. Let $\{g_1,g_2,\cdots,g_p\}$ be the p Boolean functions associated with nodes in level l of an s-circuit for f, where g_i: $\{0,1\}^n\to\{0,1\}$. Let g: $\{0,1\}^n\to\{0,1\}^p$ be defined by $g=(g_1,g_2,\cdots,g_p)$. It follows that

$$\nu(f,A)\leqq\nu(g,A)\leqq\prod_{i=1}^{p}\nu(g_i,A) \tag{1}$$

because we can write f as the composition of h: $\{0,1\}^p\to\{0,1\}^m$ with g, namely, $f=h\circ g$, from which it is clear that f has no more subfunctions over A than g. The second inequality given above is trivial.

Applying logarithms to (1), taking averages, and using the additivity of expectations, we have the following inequality

$$\operatorname*{ave}_{|A|=a}\log_2\nu(f,A)\leqq\sum_{i=1}^{p}\operatorname*{ave}_{|A|=a}\log_2\nu(g_i,A).$$

Let b_i be the number of variables on which g_i depends and let $b=\max_i(b_i)$. Then, applying Lemma (1) we have

$$\operatorname*{ave}_{|A|=a}\log_2\nu(f,A)\leqq p/(1-R) \tag{2}$$

when $R<1$ where

$$R=2ab/(n-a-b+1).$$

Since the functions $\{g_1,\cdots,g_p\}$ correspond to nodes at level l and since Ω has fan-in r, it follows that none of these functions depend upon more than r^l variables, that is, $b\leqq r^l$.

Since R is an increasing function of b, if

$$r' \leqq \frac{\delta(n-a+1)}{2a+\delta}$$

then the condition $R < \delta$ will be met. Let L be the largest integer satisfying this condition. If $L \leqq D_\Omega(f)$, then every s-circuit for f will have at least L levels and from (2) at least $(1-\delta)\,\mathrm{ave}_{|A|=a} \log_2 \nu(f, A)$ nodes at each level. From this we have the desired conclusion. □

We now apply this theorem to two important problems.

4. Applications. Given a graph with $p \times p$ adjacency matrix $X = (x_{ij})$, the *marriage problem* is to find a $p \times p$ permutation matrix $P = (\pi_{ij})$ which maximizes

$$X \cdot P = \sum_{i,j} \pi_{ij} x_{ij}$$

where $\sum$ denotes integer addition. The maximum is denoted $m(X)$. In [9] we examine $m_0(X) = m(X)$ modulo 2 which is a Boolean function and $m_0(X): \{0, 1\}^n \to \{0, 1\}$ where $n = p^2$. In that paper we show that if A covers entries in X that form a permutation matrix P^*, such as the elements on the diagonal, then the number of subfunctions $m_0|_{i,j \in A^c}^{x_{ij}=c_{ij}}$ is at least 2^t where t is the number of entries in A^c (the complement of A) which lie in the upper right-hand quadrant of X after X has been permuted so that elements of P^* lie on the main diagonal. If $a = |A|$ then

$$t = \left(\left\lceil \frac{p-1}{2} \right\rceil\right)^2 - (a-p). \tag{3}$$

Erdös and Rényi [12] have shown that if $a = p \log_e p + \alpha p$ where $\alpha = \alpha(p)$ is any growing function of p, then the fraction of the sets A which do not cover a permutation approaches zero with increasing p. Combining this result with the above we have the following.

THEOREM 2. *Let Ω be a basis of fan-in r. Then,*

$$C_\Omega^s(m_0) \geqq (1-o(1))\frac{n}{4}\log_2 n$$

where $n = p^2$ is the number of variables of m_0.

Proof. For all but a vanishingly small set of sets A, $\log_2 \nu(m_0, A) \geqq t$ where t is given by (3). It follows that the average of this quantity is greater than or equal to $(1-o(1))t$. The conclusion follows from the observation that m_0 depends on each of its $n = p^2$ variables so that $D_\Omega(m_0) \geqq \log_r n \geqq L$ for $\delta = o(1)$. □

A similar result applies to $\det D: \{0, 1\}^p \to \{0, 1\}$ which is the determinant of a $p \times p$ matrix $D = (d_{ij})$ over $GF(2)$. Suppose that A covers a permutation of D such as the main diagonal. Set to 1 the values of all variables in A or A^c which lie below the diagonal. Let A_U^c be A^c restricted to the elements above the diagonal.

Then two subfunctions $\det D|_{i,j \in A_U^c}^{d_{ij}=c_{ij}}$, $\det D|_{i,j \in A_U^c}^{d_{ij}=c'_{ij}}$ which differ in an element $(u, v) \in A_U^c$ are different because one ($c'_{u,v} = 1$) contains $\prod_{i \neq u,v} d_{ii}$ in its ring-sum expansion while the other ($c_{u,v} = 0$) does not. Thus, $\det D$ restricted to A has at least 2^t subfunctions for $t = \frac{1}{2}(|A^c| - (a-p)) = \frac{1}{2}(p^2 + p - 2a)$. Again applying the Erdös and Rényi result we have the following theorem, the proof of which parallels that of Theorem 2.

THEOREM 3. *Let Ω be a basis of fan-in r. Then,*

$$C_\Omega^s(\det D) \geqq (1-o(1))\frac{n}{2}\log_2 n$$

where $n = p^2$ is the number of variables of $\det D$.

5. Comments and conclusions. We have shown that lower bounds to the synchronous combinational complexity of functions can be derived in terms of the number of subfunctions which they contain. The method has been applied to the marriage function and the determinant function, both modulo 2. These two binary functions have multiple inputs and a single output.

Lower bounds of the order $n \log_2 n$ can be derived for a number of n-input, multiple output binary functions such as Boolean matrix-matrix multiplication, the Fourier transform function and binary addition and multiplication. For each case it is sufficient to show that the cardinality of the range of the function in question is at least $2^{\alpha n}$, $\alpha > 0$, so that s-circuits have widths at least αn, and to show that the delay complexity of the function is on the order of $\log_2 n$. To restate the result for binary addition: if a binary adder is laid out on a rectangular grid, most of the area (which is $O(n \log n)$) occupied by the circuit may consist of wires because there exist binary adders with $O(n)$ logic elements.

Symmetric Boolean functions on n inputs and the (symmetric) n-input, n-output binary sorting function have linear s-complexity because they can be realized by s-circuits that have a small neck through which passes the binary representation for the number 1's among their inputs.

Synchronous combinational complexity and the structure of circuits which it highlights may prove helpful in developing strong lower bounds on the standard combinational complexity of Boolean functions.

Acknowledgment. The authors express their sincere appreciation to several referees whose suggestions have improved the clarity of this paper.

REFERENCES

[1] J. E. SAVAGE, *Computational work and time on finite machines*, J. Assoc. Comput. Mach., 19 (1972), pp. 660–74.

[2] N. PIPPENGER AND M. FISCHER, *Relations among complexity measures*, IBM Res. Rep. RC6569, June 1977. (See [6, Chap. 5].)

[3] S. A. COOK, *The complexity of theorem-proving procedures*, Proc. Third ACM Symp. on Theory of Computing, 1971, pp. 151–158.

[4] R. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[5] M. S. PATERSON, *An introduction to Boolean function complexity*, Astérique, to appear; also Stanford Univ. CS Rep. STAN-CS-76-557.

[6] J. E. SAVAGE, *The Complexity of Computing*, Wiley-Interscience, New York, 1976.

[7] A. EHRENFEUCHT, *Practical decidability*, Rep. Cu-Cs-008-72, Dept. of Computer Science, Univ. of Colorado, Boulder, 1972.

[8] L. J. STOCKMEYER AND A. R. MEYER, *Inherent computational complexity of decision problems in logic and automata theory*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1977.

[9] L. H. HARPER AND J. E. SAVAGE, *On the complexity of the marriage problem*, Advances in Math., 9 (1972), pp. 299–312.

[10] L. H. HARPER, *An n log n lower bound on synchronous combinational complexity*, Trans. Amer. Math. Soc., to appear.

[11] O. B. LUPANOV, *On networks of functional elements with delays*, Systems Theory Res., 23 (1973), pp. 43–83.

[12] P. ERDÖS AND A. RÉNYI, *On random matrices*, Publ. Math. Inst. Hung. Acad. Sci., 8 (1963), pp. 455–461.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0002 $01.00/0

ON THE PARALLEL EVALUATION OF MULTIVARIATE POLYNOMIALS*

LAURENT HYAFIL†

Abstract. We prove that any multivariate polynomial P of degree d that can be computed with $C(P)$ multiplications-divisions can be computed in $O(\log d \cdot \log C(P))$ parallel steps and $O(\log d)$ parallel multiplicative steps.

Key words. Arithmetic complexity, parallel computation

1. Introduction. We prove that any multivariate polynomial P of degree d that can be computed with $C(P)$ multiplications-divisions can be computed in $O(\log d \cdot \log C(P))$ parallel steps and $O(\log d)$ parallel multiplicative steps. This result has to be compared with the best known lower bound of max $(\log d, \log C(P))$. (See for instance [1] for exposition).

If we apply this result to the parallel inversion of a matrix $n \times n$, it shows the existence of an algorithm in $O(\log^2 n)$ parallel steps: by Cramer's rule, the inverse of an $n \times n$ matrix is a set of quotients of polynomials of degree $\leqq n$ and of complexity $O(n^{2.81})$ (determinants). Such a result was already known by a method specific to this problem [2]. This specific method uses $O(n^4)$ processors whereas our method uses $O(n^{\log n})$ processors.

2. Definition. For $R_1, R_2, \cdots, R_m \in K(x_1, x_2, \cdots, x_n)$, $C^*(R_1, R_2, \cdots, R_m)$ will denote the minimum number of scalar multiplications-divisions necessary to compute $R_1, R_2, \cdots, R_m$ given $K \cup \{x_1, x_2, \cdots, x_n\}$.

A program β will be called homogeneous of degree d if:

(a) For any additive operation of β, $P_i = Q_i + R_i$, Q_i and R_i are homogeneous polynomials of the same degree $\leqq d$;

(b) β has no division;

(c) For any multiplication $P_i = Q_i \times R_i$ of β, Q_i and R_i are homogeneous and the degree of P_i is $\leqq d$.

If $P_1, P_2, \cdots, P_m \in K[x_1, x_2, \cdots, x_n]$, $C_d(P_1, P_2, \cdots, P_m)$ will denote the minimum number of nonscalar multiplications necessary to compute $P_1, P_2, \cdots, P_m$ with a homogeneous program of degree d.

3. Statement of the results.

THEOREM 1. *Let $\mathcal{H}$ be an homogeneous program computing homogeneous polynomials in n indeterminates $P_1, P_2, \cdots, P_m$ of degrees $\leqq d$ with $C_d(P_1, P_2, \cdots, P_m)$ multiplications. Then there exist two sets of homogeneous polynomals*:

$U(\mathcal{H}) = (U_i)$ *for* $1 \leqq i \leqq I$ *where* $I \leqq n + C_d(P_1, P_2, \cdots, P_m)$;

$V(\mathcal{H}) = (V_{i,j})$ *for* $1 \leqq i \leqq I$ *and* $1 \leqq j \leqq \lambda$ *where* λ *is the number of operations of* $\mathcal{H}$, *satisfying*:

(a) $\dfrac{d}{3} \leqq \deg(U_i) \leqq 2d/3$ *for* $1 \leqq i \leqq I$.

(b) $\deg(V_{i,j}) \leqq \dfrac{2d}{3}$ *for* $1 \leqq i \leqq I$, $1 \leqq j \leqq \lambda$.

(c) *If* $\mathcal{H}$ *computes* f_i $(1 \leqq i \leqq \lambda)$ *and* $d/3 \leqq \deg(f_i) \leqq d$ *then* $f_i = \sum_{j=1}^{I} U_j V_{j,i}$.

* Received by the editors October 6, 1977, and in final revised form June 20, 1978.

† Compagnie IBM France, Centre Scientifique, 36 Avenue Raymond Poincaré, 75116 Paris, France.

(d) $C_d(U_i) \leqq C_d(P_1, P_2, \cdots, P_m)$ *for* $1 \leqq i \leqq I$.

(e) $C_d(V_{i,j}) \leqq C_d(P_1, P_2, \cdots, P_m)$ *for* $1 \leqq i \leqq I$, $1 \leqq j \leqq \lambda$.

Proof. Let $L(P_1, P_2, \cdots, P_m)$ be the minimal number of operations of a homogeneous program which computes $P_1, P_2, \cdots, P_m$ with $C_d(P_1, P_2, \cdots, P_m)$ multiplications.

The proof is by induction on $L(P_1, P_2, \cdots, P_m)$.

The case $L(P_1, P_2, \cdots, P_m) \leqq \lambda$ being obvious assume Theorem 1 is true for $L(P_1, P_2, \cdots, P_m) \leqq \lambda$ and consider a set of polynomials $P_1, P_2, \cdots, P_m$ such that $L(P_1, P_2, \cdots, P_m) = \lambda + 1$.

The homogeneous program $\mathcal{H}$ which computes $P_1, P_2, \cdots, P_m$ in $\lambda + 1$ operations has a last operation which can be assumed to be $P_1 = f \circ f'$ without loss of generality. Let $\mathcal{H}'$ denote the program $\mathcal{H}$ without this last operation.

We denote by V_j and V'_j for $1 \leqq j \leqq I$, polynomials of $V(\mathcal{H}')$ corresponding to f and f':

$$\text{if } \deg(f) \geqq d/3 \text{ then } f = \sum_{j=1}^{I} U_j V_j,$$

$$\text{if } \deg(f') \geqq d/3 \text{ then } f' = \sum_{j=1}^{I} U_j V'_j.$$

Case 1: $P_1 = f + f'$. We first show that

$$C_d(P_1, P_2, \cdots, P_m) = C_d(f, f', P, \cdots, P_m).$$

It is obvious that

$$C_d(f, f', P_2, \cdots, P_m) \leqq C_d(P_1, P_2, \cdots, P_m).$$

Assume $C_d(f, f', P_2, \cdots, P_m) < C_d(P_1, P_2, \cdots, P_m)$.Since we can obviously build a homogeneous program which computes $P_1, P_2, \cdots, P_m$ in $C_d(f, f', P_2, \cdots, P_m)$ multiplications, we have a contradiction.

Since $C_d(f, f', P_2, \cdots, P_m) = C_d(P_1, P_2, \cdots, P_m)$ then $L(f, f', P_2, \cdots, P_m) = \lambda$ and we can apply the induction hypothesis to $f, f', P_2, \cdots, P_m$ obtaining two sets of polynomials $U(\mathcal{H}')$ and $V(\mathcal{H}')$ satisfying the above conditions (a) to (e).

Since the last operation of $\mathcal{H}$ is $P_1 = f + f'$ and $P_1 = \sum_{j=1}^{I} U_j(V_j + V'_j)$ we can define $U(\mathcal{H})$ and $V(\mathcal{H})$ from $U(\mathcal{H}')$ and $V(\mathcal{H}')$ by

$$U(\mathcal{H}) = U(\mathcal{H}'),$$

$$V(\mathcal{H}) = V(\mathcal{H}') \cup \{V_{j,\lambda+1} | \text{for } 1 \leqq j \leqq I\}, \qquad \text{where } V_{j,\lambda+1} = V_j + V'_j \text{ for } 1 \leqq j \leqq I.$$

It is obvious to check that $U(\mathcal{H})$ and $V(\mathcal{H})$ satisfy the above conditions (a) to (e).

Case 2: $P_1 = f \times f'$. It is obvious that

$$C_d(P_1, P_2, \cdots, P_m) \leqq C_d(f, f', P_2, \cdots, P_m) + 1$$

and since if $C_d(P_1, P_2, \cdots, P_m) < C_d(f, f', P_2, \cdots, P_m) + 1$ were true an immediate contradiction would appear, we have $C_d(P_1, P_2, \cdots, P_m) = C_d(f, f', P_2, \cdots, P_m) + 1$ and $L(f, f', P_2, \cdots, P_m) = \lambda$. We can apply the induction hypothesis to $f, f', P_2, \cdots, P_m$ obtaining two sets of polynomials $U(\mathcal{H}')$ and $V(\mathcal{H}')$ satisfying the above conditions (a) to (e). We consider the following cases:

(α) $\deg(P_1) < d/3$. If we take $U(\mathcal{H}) = U(\mathcal{H}')$ and $V(\mathcal{H}) = V(\mathcal{H}') \cup \{V_{j,\lambda+1} | 1 \leqq j \leqq I\}$ where $V_{j,\lambda+1} = 0$ for $1 \leqq j \leqq I$, $U(\mathcal{H})$ and $V(\mathcal{H})$ obviously satisfy the above conditions (a) to (e).

(β) $\deg(f) \geqq d/3$. $P_1 = f \times f' = \sum_{j=1}^{I} U_j(V_j f')$, and we choose $U(\mathcal{H}) = U(\mathcal{H}')$ and $V(\mathcal{H}) = V(\mathcal{H}') \cup \{V_{j,\lambda+1} | 1 \leqq j \leqq I\}$ where $V_{j,\lambda+1} = V_j f'$ if $\deg(U_j V_j f') \leqq d$, $= 0$ otherwise for $1 \leqq j \leqq I$. $U(\mathcal{H})$ and $V(\mathcal{H})$ obviously satisfy conditions (a) to (d). To establish

condition (e), we use the induction hypothesis:

$$C_d(V_j) \leqq C_d(f, f', P_2, \cdots, P_m) \quad \text{for } 1 \leqq j \leqq I.$$

Hence $C_d(V_j f') \leqq C_d(f, f', P_2, \cdots, P_m) + 1$ and

$$C_d(V_j f') \leqq C_d(P_1, P_2, \cdots, P_m).$$

(γ) $\deg(f') \geqq d/3$. The same proof as in (β) holds by permuting f and f'.

(δ) $\deg(f) < d/3$ and $\deg(f') < d/3$. We have $d/3 \leqq \deg(P_1) < 2d/3$, having assumed (α), (β), (γ) do not hold. We take $U(\mathcal{H}) = U(\mathcal{H}') \cup \{U_{I+1}\}$ with $U_{I+1} = P_1$. $I+1$ satisfies: $I+1 \leqq n + C_d(P_1, P_2, \cdots, P_m)$ since we know $I \leqq n + C_d(f, f', P_2, \cdots, P_m)$ and $C_d(f, f', P_2, \cdots, P_m) = C_d(P_1, P_2, \cdots, P_m) - 1$. We take $V(\mathcal{H}) = V(\mathcal{H}') \cup \{V_{j,\lambda+1} | 1 \leqq j \leqq I\} \cup \{V_{I+1,k} | 1 \leqq k \leqq \lambda + 1\}$ with $V_{I+1,\lambda+1} = 1$ and $V_{i,j} = 0$ for $i = I+1$ and $j \neq \lambda + 1$ or $i \neq I+1$ and $j = \lambda + 1$. With such a choice $U(\mathcal{H})$ and $V(\mathcal{H})$ satisfy conditions (a) to (e). □

In order to establish Theorem 2 we first show:

LEMMA 1. *Let P be a homogeneous polynomial of degree $\leqq d$ in n indeterminates, then P can be computed in $(1/\log_2 3 - 1) \log_2 d$ parallel multiplicative steps, and in*

$$\left[1 + \frac{\log_2 d}{\log_2 3 - 1}\right] (\lceil \log_2 [C_d(P) + n] \rceil + 1) \quad \textit{parallel steps.}$$

Proof. The proof is by induction on d. For $d = 1$, the proof is trivial. Assume it is true for $d' < d$, and we prove it for $d' = d$.

From Theorem 1 we know that: $P = \sum_{j=1}^{I} U_j V_j$ with $I \leqq n + C_d(P)$ and U_j and V_j are homogeneous polynomials which satisfy for $1 \leqq j \leqq I$:

(a) $\deg(U_j) \leqq 2d/3$.
(b) $\deg(V_j) \leqq 2d/3$.
(c) $C_d(U_j) \leqq C_d(P)$.
(d) $C_d(V_j) = C_d(P)$.

Applying the induction hypothesis shows that U_j and V_j $(1 \leqq j \leqq I)$ can be computed in less than $(1/(\log_2 3 - 1)) \log_2 (2d/3)$ parallel multiplicative steps $\log_2 (2d/3)/(\log_2 3 - 1)(\lceil \log_2 [C_d(P) + n] \rceil + 1)$ parallel steps.

To compute P, we compute in parallel U_j and V_j for $1 \leqq j \leqq I$ multiply U_j by V_j in one parallel multiplicative step and sum up in $\lceil \log_2 (I) \rceil \leqq \lceil \log_2 [n + C_d(P)] \rceil$ additive steps. Summing the total number of steps gives the announced result to compute P. □

LEMMA 2. *Let P be a multivariate polynomial of degree d and $P_1, P_2, \cdots, P_d$ the homogeneous terms of P then*

$$C_d(P_1, P_2, \cdots, P_d) \leqq \left[\frac{d(d-1)}{2}\right]^2 C^*(P).$$

Proof. The proof given in [3] consists in first eliminating division using Strassen's transformation [4] then in separating homogeneous components in every intermediary result. □

THEOREM 2. *A polynomial P of degree $\leqq d$ in n indeterminates which can be computed with $C^*(P)$ multiplications-divisions can be computed with no more than $\lceil (1/[\log_2 3 - 1]) \log_2 d \rceil$ parallel multiplicative steps, and than*

$$\left[1 + \frac{\log_2 d}{[\log_2 3 - 1]}\right] \left[\log_2 \left[\left(\frac{d(d-1)}{2}\right)^2 C^*(P) + n\right] + 1\right] + \lceil \log_2 d \rceil$$

parallel steps.

Proof. To compute P in parallel, we compute in parallel each of the d homogeneous components of $P: P_1, P_2, \cdots, P_d$ and then add them in parallel in $\lceil \log_2 d \rceil$ additive steps. The result is then deduced immediately from Lemmas 1 and 2. □

4. Acknowledgments. Many thanks to A. Borodin for helpful corrections on the first draft of this paper.

REFERENCES

[1] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[2] L. CSANKY, *Fast parallel matrix inversion*, this Journal, (1976), pp. 618–628.

[3] L. HYAFIL, *The power of commutativity*, 18th F.O.C.S. Conference Proc., 1977, pp. 171–174.

[4] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0003 $01.00/0

A NEW REPRESENTATION OF THE RATIONAL NUMBERS FOR FAST EASY ARITHMETIC*

E. C. R. HEHNER† AND R. N. S. HORSPOOL‡

Abstract. A novel system for representing the rational numbers based on Hensel's p-adic arithmetic is proposed. The new scheme uses a compact variable-length encoding that may be viewed as a generalization of radix complement notation. It allows exact arithmetic, and approximate arithmetic under programmer control. It is superior to existing coding methods because the arithmetic operations take particularly simple, consistent forms. These attributes make the new number representation attractive for use in computer hardware.

Key words. number systems, number representation, rational arithmetic, p-adic numbers, radix complement, floating-point

1. Introduction. "It's very illuminating to think about the fact that some—at most four hundred—years ago, professors at European universities would tell the brilliant students that if they were very diligent, it was not impossible to learn how to do long division. You see, the poor guys had to do it in Roman numerals. Now, here you see in a nutshell what a difference there is in a good and bad notation." (Dijkstra 1977)

We consider that a good scheme for representing numbers, especially for computers, would have the following characteristics.

(a) All rational numbers are finitely representable. This requires that the representation be variable-length.

(b) The representation is compact. It should require less space, on average, than the fixed-length schemes commonly used in computers. Since the numbers provided by a fixed-length representation are not used equally often, compactness can be achieved by giving frequently-occurring numbers short encodings at the expense of longer encodings for less-frequent numbers.

(c) The addition, subtraction, and multiplication algorithms are those of the usual integer arithmetic. The division algorithm is as easy as multiplication, and it proceeds in the same direction as the other three algorithms. This property is important for the storage and retrieval of variable-length operands and results.

Although the desired representation is variable-length, there is no implication that operations must be performed serially by digit. Just as data can be retrieved and stored d digits at a time, so the arithmetic unit can be designed to perform operations d digits at a time. By choosing d to be large compared to the average length of operands, we can obtain the speed of a fixed-length design together with the ability to handle operands that are not representable in a fixed length (Wilner 1972).

2. Background. Before we present our proposal, we shall briefly review representations in common use. (For their history, see (Knuth 1969a).)

Almost universally, the sequence of digits

$$\cdots d_i \cdots d_3 \; d_2 \; d_1 \; d_0$$

is used to represent the nonnegative integer

$$n = \sum_i d_i b^i$$

* Received by the editors November 18, 1977, and in final revised form June 19, 1978.
† Computer Systems Research Group, University of Toronto, Toronto, Ontario, Canada M5S 1A4.
‡ School of Computer Science, McGill University, Montreal, Quebec H3A 2K6.

where b (the base) is an integer greater than one (usually two or ten), and each digit d_i represents an integer in the range $0 \leqq d_i < b$. As a rule, we do not write leading 0s. We must break the rule, however, to represent zero (if we follow it, there is nothing to write).

There is no direct representation of negative integers in common use. Instead we prefix a unary operator to the representation of positive integers. The combination of "sign and magnitude" is indirect because, to perform arithmetic, we may first have to apply some algebraic transformations. If asked to add two numbers, we first examine the signs to determine whether to use the addition or subtraction algorithm; if we are using the subtraction algorithm, we compare the magnitudes to determine which is to be subtrahend, and which minuend. With a direct representation, if asked to add, we simply add. The radix complement representation is direct in this sense, but it includes only a finite subset of the integers.

Rationals are commonly represented by a pair of integers: a numerator and denominator. In this form, multiplication and division are reasonably easy, but addition and subtraction are relatively hard, and normalization is difficult (Horn 1977). When addition and subtraction are wanted more often than multiplication and division, a representation that makes the former easier at the expense of the latter would be preferable. For this reason, we usually restrict our numbers to a subset of the rationals known as the "fixed-point" or "floating-point" numbers. By inserting a radix point in a sequence of digits (fixed-point), or indicating by means of an exponent where a radix point should be placed (floating-point), we represent those rationals such that, in lowest terms, the denominator divides some power of the base. In this form, addition and subtraction are, after alignment of the radix point, the same as for integers. With a variable-length representation, a major difficulty with the usual division algorithm is that it proceeds from left to right, opposite to the direction of the other three algorithms. To simplify retrieval, processing, and storage, all algorithms should examine their operands and produce their results in the same direction.

The left-to-right division algorithm gives us a way of extending the fixed/floating-point representation to include all positive rationals: an infinite but eventually repeating sequence of digits can be finitely denoted by indicating the repeating portion. On paper, the repeating portion is sometimes denoted by overscoring it; for example, $611/495 = 1.2\overline{34}$. A minor annoyance is the fact that representations are not unique; for example, $0.\overline{9} = 1.$ and $0.4\overline{9} = 0.5$. A major annoyance is that further arithmetic is awkward: addition normally begins with the rightmost digit, but a sequence that extends infinitely to the right has no rightmost digit.

The usual representation of nonnegative integers can be extended in various ways. The base may be negative (Songster 1963), or even imaginary (Knuth 1960). In the "balanced ternary" representation (Avizienis 1971), the base is three, and the digits represent the integers minus-one, zero, and one. Our extension, which we now present, is in quite a different direction.

3. Constructing the representation. To construct our representation, we shall follow the approach of Hensel's p-adic arithmetic (Hensel 1908, 1913). Hensel begins with the usual representation of nonnegative integers, that is, a sequence of digits $\cdots d_3\, d_2\, d_1\, d_0$. Each digit d_i represents an integer in the range $0 < d_i < b$, where the base b is an integer greater than one. He then constructs the representation of other numbers (all rationals, some irrationals and some imaginary numbers (Knuth 1969b)) by means of arithmetic. We shall limit ourselves to rational numbers, and give a finite representation of them that is implementable in computer hardware.

3.1. Addition, subtraction and multiplication. For the addition and subtraction algorithms, we adopt the usual algorithms for positive integers, with one qualification. When subtracting a digit of the subtrahend from a smaller digit of the minuend, rather than "borrow" one from a minuend digit some distance away, we "carry" one to the immediately neighboring digit of the subtrahend. For example,

$$\begin{array}{r} 2\ 2\ 0\ 0\ 4 \\ -\ \ 3_1\ 5_1\ 2_1\ 6 \\ \hline 2\ 3\ 4\ 7\ 8 \end{array}$$

beginning at the right, we subtract 6 from 4 to get 8 with a carry; then, rather than subtracting the 2 in the next position, we subtract 3 from 0 to get 7 with a carry; etc. Whether we "carry" or "borrow" is inconsequential; the important point is that we affect only the immediately neighboring digit, not a digit an arbitrary distance away. This form of subtraction is usual in circuit design; it is crucial to our number representation.

We now construct minus-one by subtracting one from zero.

$$\begin{array}{r} \cdots 0\ 0\ 0\ 0 \\ -\cdots 0_1\ 0_1\ 0_1\ 1 \\ \hline \cdots 9\ 9\ 9\ 9 \end{array}$$

Beginning at the right, the subtraction algorithm generates a sequence of 9s. Though the sequence is unending, it is repetitive, and can be specified finitely. We shall use a quote (quotation mark) to means that the digit(s) to its left is (are) to be repeated indefinitely to the left. Twenty-five, for example, is represented by 0'25 or by 0'025 or by 00'25. The first of these is called "normalized"; in general, a representation is normalized when it is as short as possible. A table of normalized representations of integers follows; each entry specifies the usual representation together with our representation.

		0:	0'
1:	0'1	−1:	9'
2:	0'2	−2:	9'8
3:	0'3	−3:	9'7
:		:	
9:	0'9	−9:	9'1
10:	0'10	−10:	9'0
11:	0'11	−11:	9'89
:		:	

For brevity on the written page (the abbreviation is inapplicable in computer memories), we make the convention that when no quote appears in a number, 0' is assumed to be appended to its left. Thus the positive integers take their familiar form.

The reader may verify, with examples, that the addition and subtraction algorithms work consistently throughout the scheme. This is no surprise to anyone familiar with radix complement arithmetic, for the similarity is apparent. The rule for negation is the radix complement rule: complement each digit (in decimal, change 0 to 9, 1 to 8, 2 to 7, etc.) and then add one. (Negation can be performed more simply; for example, in binary starting at the right, leave trailing 0s and rightmost 1 alone, flip the rest.) The definition of radix complement is made in terms of a fixed-size space available for storing numbers in computers (the "word"); for example, if w bits are available for encoding integers, two's complement is defined by performing arithmetic modulo 2^w. We prefer to define it

independent of space constraints, and to view the fixed-size scheme as a truncation of p-adic numbers to w bits. Radix complement is often preferred in computers because of its nice arithmetic properties; our finite representation of p-adic rationals enjoys these properties without suffering from the wasted space and overflow problems inherent in a fixed-length representation.

The usual multiplication algorithm need not be altered for use with negative integers. For example, multiplying minus-two by three, either way round, yields the desired result.

```
  9'8          3
* 3          * 9'8
-----        -----
 9'4          2 4
           2 7
         2 7
           :
           :
        -------
          9'4
```

3.2. Division. Before we present the division algorithm, let us look at a simple example. Since 9' represents minus-one, we may expect division by 3 to give 3' as the representation of minus-one-third. Let us test the consistency of this representation with our arithmetic. To negate, we first complement, obtaining 6' (since 6' is double 3', it represents minus-two-thirds); then we add one, obtaining 6'7 as our representation of one-third. Subtracting 3' from 0' also produces 6'7. Multiplying 6'7 by 3 yields 1, confirming its consistency.

	0' − 1 = 9'	(minus-one)
	9' ÷ 3 = 3'	(minus-one-third)
complement of 3' = 6'		
or	3' * 2 = 6'	(minus-two-thirds)
	6' + 1 = 6'7	(one-third)
or	0' − 3 = 6'7	
	6'7 * 3 = 1	

We now present the division algorithm by means of an example. For the moment, we restrict ourselves to integer dividends and divisors, and further to divisors whose rightmost digit is nonzero and relatively prime to the base. (Two integers are relatively prime if and only if their only common divisor is 1. In decimal, the divisor's final digit must be 1, 3, 7, or 9.) We shall remove all restrictions shortly.

For our example, we divide 191 by 33. The rightmost digit of the result, d_0, when multiplied by the rightmost digit of the divisor, 3, must produce a number whose rightmost digit is the rightmost digit of the dividend, 1. Our restrictions ensure that there is exactly one such digit: $d_0 * 3$ must end with 1, therefore $d_0 = 7$. We subtract $7 * 33$ from 191; ignoring the rightmost digit, which must be 0, we use the difference in place of the dividend, and repeat.

```
  1 9 1 ÷ 3 3 = 1 2 7
- 2 3 1
-------
    9'6
-   6 6
-------
    9'6
-   3 3
-------
    9'6
```

When the difference is one we have seen before, we can insert the quote, and stop.

Like multiplication, the division algorithm produces each digit by table look-up. Compare this with the left-to-right division algorithm: a "hand calculation" requires a guess that may need to be revised; a "machine calculation" requires repeated subtraction to produce each digit of the result.

The right-to-left division algorithm is well-defined only when the divisor's final digit is relatively prime to the base. For example, an attempt to divide 1 by 2 in decimal fails because no multiple of 2 has 1 as its final digit. Therefore, the means presented so far enable us to represent only a subset of the rationals: those rationals such that, in lowest terms, the denominator has no factors that are factors of the base. Compare this with the fixed/floating-point representation; it represents those rationals such that, in lowest terms, the denominator has only factors that are factors of the base. The two representations are, in a sense, complementary; combining them, we are able to represent all rationals. We allow a radix point, whose position is independent of the quote, or an exponent, as in scientific notation. For example,

$$\begin{aligned} 12'34 \div 10 &= 12'3.4 \\ 12'3.4 \div 10 &= 12'.34 \\ 12'.34 \div 10 &= 1.2'34 \\ 1.2'34 \div 10 &= .12'34 \\ .12'34 \div 10 &= .21'234 \end{aligned}$$

The last example is probably best written with an exponent, and suggests that the more appropriate machine representation uses an exponent rather than a radix point. This requires a normalization rule: when normalized, the mantissa's rightmost digit is not 0. Thus 12'300E2 becomes 12'3E5, and 120'E2 becomes 012'E3. All rationals except zero have a unique normalized representation.

To divide two arbitrary integers in decimal, we must first "cast out" all 2 and 5 factors from the divisor. These factors can be determined, one at a time, by inspecting the divisor's final digit. A 2 is cast out by multiplying both dividend and divisor by 5; similarly, a 5 is cast out with a multiplication by 2. The generalization to arbitrary (rational) operands is now easy.

4. Binary is beautiful. Base two, a common choice in computers for circuit reasons, has two important advantages for us. First, it is a prime number. Since it has no factors, there is no "casting out"; division of two normalized operands can always proceed directly. Second, multiplication and division tables are trivial. This advantage for multiplication is well-known; the right-to-left division algorithm is a true analogue of multiplication, so the same advantage applies. At each step, the rightmost bit of the dividend remaining

(a) is the next bit of the result, and

(b) specifies whether or not to subtract the divisor.

For example, dividing 1 (one) by 11 (three) proceed as follows.

$$\begin{array}{rl} & 1 \div 1\ 1 = 0\ 1'1 \\ -\ 1\ 1 & \\ \hline 1' & \\ -\ 1\ 1 & \\ \hline 1'0 & \end{array}$$

1. Dividend = 1. Its rightmost bit = 1. Therefore
   (a) Rightmost bit of result = 1.
   (b) Subtract divisor, and ignore rightmost 0.
2. Dividend remaining = 1'. Its rightmost bit = 1. Therefore
   (a) Next bit of result = 1.
   (b) Subtract divisor, and ignore rightmost 0.
3. Dividend remaining = 1'0. Its rightmost bit = 0. Therefore
   (a) Next bit of result = 0.
   (b) Do not subtract divisor; ignore rightmost 0.
4. Dividend remaining = 1' as in step 2. Therefore insert quote. Result = 01'1.

5. Radix conversion algorithm. The algorithm for converting the representation of a positive integer from one base to another is well-known. The given integer is repeatedly divided by the base of the new representation; the sequence of remainders gives, from right to left, the digits of the new representation. For example, converting eleven from decimal to binary

$$\begin{aligned} 11 &= 2*5+1 \\ 5 &= 2*2+1 \\ 2 &= 2*1+0 \\ 1 &= 2*0+1 \end{aligned}$$

gives 1011. The algorithm is usually considered to terminate when the quotient is zero; if it is allowed to continue, an endless sequence of 0s is produced.

Let us apply the algorithm to a negative integer. For example, converting minus-eleven from decimal to binary

$$\begin{aligned} -11 &= 2*-6+1 \\ -6 &= 2*-3+0 \\ -3 &= 2*-2+1 \\ -2 &= 2*-1+0 \\ -1 &= 2*-1+1 \\ -1 &= 2*-1+1 \\ &\cdot \quad \cdot \ \cdot \\ &\cdot \quad \cdot \ \cdot \end{aligned}$$

gives $\cdots 1110101$, which is the p-adic form. Note that the remainders must be digits in the new base. By recognizing a repeated state of the computation, we are able to insert a quote and arrive at our finite representation 1'0101.

If we begin with a fraction, we must allow the quotients to be fractions. We then need an extra criterion to make the algorithm deterministic. For example, converting one-third to binary, we might begin either with

$$1/3 = 2*-1/3+1$$

or with

$$1/3 = 2*1/6+0.$$

To produce a result without a radix point, the original fraction, and each quotient fraction, in lowest terms, must have denominators that are relatively prime to the new base. The implied criterion is that the denominator must remain unchanged.

$$\begin{aligned} 1/3 &= 2*-1/3+1 \\ -1/3 &= 2*-2/3+1 \\ -2/3 &= 2*-1/3+0 \end{aligned}$$

When a quotient is obtained that has appeared before, a cycle is recognized. Placing the quote accordingly, we have produced the binary representation of one-third, namely 01'1. Thus the usual radix conversion algorithm naturally produces our representation.

In the preceding examples, we began with numbers in their familiar decimal representations: sign-and-magnitude, numerator/denominator. If we begin instead with our representation, the algorithm remains the same for rationals (without radix points) as for positive integers. For example, beginning with 6'7 (one-third in decimal), we see that the rightmost digit is odd, therefore the rightmost binary digit is 1.

$$(6'7-1)\div 2 = 3'$$
$$(\ 3'-1)\div 2 = 6'$$
$$(\ 6'-0)\div 2 = 3'$$

The division by two is performed as a multiplication by five, discarding the rightmost digit of the result, which is 0. In general, when converting from base b1 to base b2, a finite number of rightmost digits of each quotient is sufficient to determine the next digit of the result if all factors of b2 are factors of b1.

6. Properties of the proposed number system. Excluding the radix point or exponent, the general form of our representation is

$$d_{n+m}d_{n+m-1}\cdots d_{n+1}{}'d_nd_{n-1}\cdots d_2d_1d_0$$

The number represented is

$$\sum_{i=0}^{n} d_ib^i - \sum_{i=n+1}^{n+m} d_ib^i/(b^m-1)$$

where b is the base of the representation. To justify this formula, we shall break the digit sequence into two parts: the digits to the left of the quote will be called the "negative part", and the digits to its right will be called the "positive part". The positive part was our starting point for the construction of the representation.

$$d_n\cdots d_0 = \sum_{i=0}^{n} d_ib^i.$$

The negative part can be found as follows.

$$d_{n+m}\cdots d_{n+1}{}' = d_{n+m}\cdots d_{n+1}{}'\underbrace{0\,0\cdots 0}_{m \text{ zeros}} + d_{n+m}\cdots d_{n+1}$$

$$= d_{n+m}\cdots d_{n+1}{}' * b^m + \sum_{i=n+1}^{n+m} d_ib^{i-n-1}.$$

Therefore

$$d_{n+m}\cdots d_{n+1}{}' = -\sum_{i=n+1}^{n+m} d_ib^{i-n-1}/(b^m-1).$$

Putting the positive and negative parts together, we find

$$d_{n+m}\cdots d_{n+1}{}'d_n\cdots d_0 = d_{n+m}\cdots d_{n+1}{}' * b^{n+1} + d_n\cdots d_0$$

and hence we obtain the above formula.

From the formula, we see that sign determination is trivial. If both positive and negative parts are present, we merely compare their leading digits. Assuming the representation to be normalized, $d_{n+m} \neq d_n$.

If $d_{n+m} < d_n$, the number is positive.

If $d_{n+m} > d_n$, the number is negative.

If one part is absent, the sign is given by the part that is present. If both parts are absent, the number is zero.

The formula can be used to convert from our representation to numerator/denominator representation, if one so desires. For example, in decimal $12'7 = 7 - 120/99 = 191/33$; in binary $01'1 = 1 - 010/11 = 1/11$ (one-third). From this formula, we can also derive an easy method for converting to a right-repeating decimal expansion (or binary expansion). Simply subtract the negative part from the positive part with their leading digits aligned and the negative part repeated indefinitely to the right.

$$12'345 = 345 - 121.\overline{21} = 223.\overline{78}$$

$$123'45 = 45 - 12.\overline{312} = 32.\overline{687}$$

$$43'21 = 21 - 43.\overline{43} = -22.\overline{43}$$

The opposite conversion can be performed by reversing the steps.

$$2.\overline{34} = 2 - 34' = 56'8$$

The above paragraphs suggest two comparison algorithms. The first is to subtract the comparands, then determine the sign of the result. The second is to convert the comparands to right-repeating form, then perform the usual digit-by-digit comparison. The first has the advantage that it is a right-to-left algorithm, but the second may have an efficiency advantage.

7. Length of representation. When two p-adic rational numbers are added, subtracted, multiplied, or divided, the result is an infinite but eventually repeating sequence of digits. For termination of the arithmetic algorithms, and for finite representation of the result (placing the quote), one must be able to recognize when the state of the computation is one that has occurred before. In a hand calculation, recognition poses no problem: one merely scans the page. The analogous approach for machines requires the arithmetic unit to contain some associative memory. Each state of the computation is compared (associatively, in parallel) with all stored states; if a match is found, the operation is complete, otherwise the state is stored and the operation continues. The resulting digit sequence must be checked for normalization.

A method of recognizing the repeated state that requires storing only one state, at the expense of possibly delaying recognition by a few digits, is the following (Brent 1978). If the states are $S_1, S_2, S_3, \cdots$, then test (S_1 vs. S_2), (S_2 vs. S_3, S_4), (S_4 vs. $S_5, \cdots, S_8$), etc., until a match is found. Again, the resulting digit sequence must be normalized.

7.1. Length Bounds. An alternative method of implementing the arithmetic algorithms is as follows:

(a) calculate a bound for the length (number of digits) of the result;

(b) calculate a correct, though not necessarily minimal, length for the negative (repeating) part;

(c) produce a sufficient number of digits to ensure that a repetition of the length calculated in (b) has occurred;

(d) normalize the result.

This approach obviates the need to recognize a repeated state. However, the complexity of our formulae for the length bounds makes the approach unattractive.

The bounds are summarized in Table 1. The notation used in the table is the following.

b: the base of the representation
p_i: the length of the positive part (i.e. the number of digits to the right of the quote) in the base b representation of x_i
n_i: the length of the negative part (i.e. the number of digits to the left of the quote) in the base b representation of x_i
ϕ: Euler's ϕ function; $\phi(m)$ is defined, for positive integer m, as the number of positive integers not exceeding m that are relatively prime to m.

TABLE 1
Length bounds.

$x_3 = x_1 + x_2$ $x_3 = x_1 - x_2$	$p_3 \leqq \text{MAX}(p_1, p_2) + n_3 + 2$ n_3 is a divisor of LCM (n_1, n_2)
$x_3 = x_1 * x_2$	$p_3 \leqq p_1 + p_2 + n_3 + 1$ n_3 is a divisor of LCM (n_1, n_2)
$x_3 = x_1 + x_2$	$p_3 \leqq \begin{cases} p_1 + p_2 + n_3 + 1 & \text{if } p_2 \leqq n_2 \\ p_1 - p_2 + n_3 + 1 & \text{if } n_2 < p_2 \leqq p_1 \\ n + 1 & \text{if } p_1, n_2 < p_2 \end{cases}$ n_3 is a divisor of LCM $(n_1, \phi(x_2(b^{n_2} - 1)))$

7.2. Approximation. Our representation provides a service that is unavailable to users of fixed-length floating-point hardware: exact results. As arithmetic operators are applied repeatedly during a computation, this extra service may begin to cost extra: the length of representation of the results may tend to grow. As the lengths grow, storage costs increase; as the lengths become greater than the number of digits that a data path or arithmetic unit can accommodate at one time, processing time increases. Many users do not require perfect accuracy, and are unwilling to pay extra for it.

Users of fixed-length floating-point hardware have their accuracy and expense chosen for them by the computer designer. The designer's impossible task is to choose one accuracy and expense (amount of storage per number) to satisfy all users at all times. Some designers give their users a choice of two accuracies (single and double precision); it seems preferable to allow each user to choose any accuracy, from none to complete.

For reasons of mathematical cleanliness, we prefer not to provide approximate versions of the addition, subtraction, multiplication, and division operators, but to provide a separate approximation operator. Two possible (and ideal) forms of this operator are:

(a) Given a number n, and a number of digits d, $n \,@\, d$ = a number that is closest to n and whose representation has no more than d digits.
(b) Given a number n, and a tolerance t, $n \,@\, t$ = a number in the range $n \pm nt$ whose representation is shortest.

The second form has the advantage that its specification is independent of representation. In either form, the problem of finding a "closest" or "shortest" result is a hard one; in practice, a reasonably close or short result is easy to obtain, and acceptable. One method is to convert the number to right-repeating form (using the algorithm in § 6) and then truncate the result. For example, 12'34.567 is first converted to 22.445 $\overline{78}$ and this

may be approximated to 0'22.4458, to 0'22.446, or to 0'22.45, etc. In general, if the final truncated result has k digits to the right of the radix point, then it is easy to show that the error introduced by the approximation is less than b^{-k} where b is the base of the number system. This form of approximation corresponds to the first style of approximation operator given above.

The role of the numerical analyst has traditionally been to analyze the accuracy (error) provided by the manufacturer, and to design algorithms which make the best use of this accuracy (i.e., whose final result is most accurate). With our proposal, the numerical analyst's role will be to choose approximations that make a computation as cheap as possible, while achieving the desired accuracy. Correctness proofs will be facilitated by making the chosen approximations explicit.

7.3. Compactness. We would like to compare the computing expense of solving a sample of numerical problems using a fixed-length floating-point representation with the expense using our representation. To make the comparison, it would be unfair to use existing programs, since they are designed to run on existing floating-point machines. We should proceed as follows.

1. Ascertain the accuracy achieved by each of the existing programs for floating-point.
2. Write programs for our representation that achieve the same accuracy as cheaply as possible. This requires numerical analysis that has not yet been developed.
3. Compare the cost of running the programs on their respective machines.

Even then, the comparison will be biased in favor of fixed-length floating-point, since its users are not given the option of more or less accuracy. Needless to say, we have not made the desired comparison.

We have, however, tested our representation on integer data (Hehner 1976). Our sample was a large, well-known compiler (XCOM, the compiler for XPL (McKeeman 1970)). The integer constants in the program require, on average, 5.7 bits each. A complete trace of the values of all integer variables and integer-valued array elements during an execution of the program (compiling itself) revealed that they require, on average, 6.4 bits each. There are some essential overhead costs associated with the use of variable-length data items. To be easily accessible, the data items may need to be addressed indirectly. Furthermore, memory compactions may be necessary with data items dynamically changing in size. Even when these costs are taken into account, there is a 31% saving compared to a 32-bit fixed-length encoding. (The cost is measured as a space-time product, since space is being traded for time when using a variable-length encoding.) Therefore, for this data, our representation is significantly more economical than a well-known standard.

8. Conclusion. The representation of the rational numbers presented in this paper has several appealing properties. Given the usual representation of the positive integers, it is, in a genuine sense, the natural extension to the rationals. The algorithms for addition, subtraction, and multiplication are those of the usual integer arithmetic; the division algorithm is truly the analogue of multiplication.

The complexity of rational arithmetic in numerator/denominator form has not been reduced by our representation, but it has been redistributed and its character has changed. For humans, the problems of detecting a repeated state and normalization have the character of a pattern-match. If they can be solved economically for computers, then we believe rational arithmetic with programmer-controlled accuracy will become an attractive proposition.

The notation introduced in this paper may be generalized in several directions. For integer base $b > 1$, the digit set can represent any b integers that are all different modulo b (balanced ternary is a special case). The base need not be a positive integer. And the representation can be used for rational power series. Krishnamurthy has considered the use of a fixed-length truncation of p-adic numbers with the result that arithmetic is exact within a limited range (Krishnamurthy 1977).

Acknowledgments. We thank Art Sedgwick and Bill McKeeman for their interest in and comments on this work. We especially thank Don Knuth for informing us of previous related work.

REFERENCES

A. AVIZIENIS (1971), *Digital Computer Arithmetic: A Unified Algorithmic Specification*, Proc. Symposium on Computers and Automata, Polytechnic Press, Polytechnic Institute, Brooklyn, NY.

R. BRENT (1978), communicated via D. E. Knuth.

EDSGER W. DIJKSTRA (1977), *An interview with prof. dr. Edsger W. Dijkstra*, Datamation, 23, no. 5, p. 164.

E. C. R. HEHNER (1976), *Computer design to minimize memory requirements*, Computer, 9, no. 8, pp. 65–70.

K. HENSEL (1908), *Theorie der algebraischen Zahlen*, Leipzig-Berlin.

——— (1913), *Zahlentheorie*, Berlin-Leipzig.

B. K. P. HORN (1977), *Rational Arithmetic for Minicomputers*, MIT Press, Cambridge, MA.

D. E. KNUTH (1960), *An Imaginary Number System*, Comm. ACM, 3, pp. 245–247.

——— (1969a), *Seminumerical algorithms*, The Art of Computer Programming, vol. 2, Addison-Wesley, Reading, MA.

——— (1969b), Ibid., § 4.1, ex. 31, p. 179.

E. V. KRISHNAMURTHY (1977), *Matrix processors using p-adic arithmetic for exact linear computations*, IEEE Trans. Computers, C-26, pp. 633–639.

W. M. MCKEEMAN, J. J. HORNING AND D. B. WORTMAN (1970), *A Compiler Generator*, Prentice-Hall, Englewood Cliffs, NJ.

G. F. SONGSTER (1963), *Negative base number representation systems*, IEEE Trans. Computers, EC-12, pp. 274–277.

W. T. WILNER (1972), *Design of the B1700*, Proceedings of AFIPS 1972, FJCC vol. 41, AFIPS Press, Montvale, NJ, pp. 489–497.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0004$01.00/0

MAXIMUM FLOW IN PLANAR NETWORKS*

ALON ITAI† AND YOSSI SHILOACH‡

Abstract. Efficient algorithms for finding maximum flow in planar networks are presented. These algorithms take advantage of the planarity and are superior to the most efficient algorithms to date. If the source and the terminal are on the same face, an algorithm of Berge is improved and its time complexity is reduced to $O(n \log n)$. In the general case, for a given $D > 0$ a flow of value D is found if one exists; otherwise, it is indicated that no such flow exists. This algorithm requires $O(n^2 \log n)$ time. If the network is undirected a minimum cut may be found in $O(n^2 \log n)$ time. All algorithms require $O(n)$ space.

Key words. algorithm, network flow, planar graph

1. Introduction.

1.1. Basics. A *directed flow network* $N = (G, s, t, c)$ is a quadruple, where:

(i) $G = (V, E)$ is a directed linear graph;

(ii) s and t are distinct vertices, the source and the terminal respectively;

(iii) $c: E \to R^+$ is the capacity function (R^+ denotes the set of nonnegative real numbers).

Henceforth, n and m denote the number of vertices and edges respectively and $u \to v$ denotes a directed edge from u to v.

A function $f: E \to R^+$ is a *flow* if it satisfies:

(a) the capacity rule: $f(e) \leqq c(e)\ \forall e \in E$;

(b) the conservation rule:

$$\mathrm{IN}(f, v) = \mathrm{OUT}(f, v) \quad \forall v \in V - \{s, t\}.$$

Where $\mathrm{IN}(f, v) = \sum_{\{u: u \to v \in E\}} f(u \to v)$ is the total flow entering v; and $\mathrm{OUT}(f, v) = \sum_{\{w: v \to w \in E\}} f(v \to w)$ is the total flow emanating from v.

The flow value $|f|$ is defined by

$$|f| = \mathrm{OUT}(f, s) - \mathrm{IN}(f, s).$$

A flow is a *maximum flow* if $|f| \geqq |f'|$ for any other flow f'.

1.2. Results. Ford and Fulkerson [6] stated and proved the Max Flow-Min Cut theorem and established the technique of augmenting paths for finding a maximum flow. Edmonds and Karp [5] provided the first polynomial algorithm ($O(nm^2)$), based on finding shortest augmenting paths. By using auxiliary graphs, Dinic [3] managed to reduce the time bound to $O(n^2 m)$ (see also [4]). By the method of preflows Karzanov implemented Dinic's algorithm in $O(n^3)$ time [9]. Note that when $m = O(n)$ all these algorithms require $O(n^3)$ time [1].

A flow network $N = (G, s, t, c)$ is *planar* if G is a planar graph. (See [7, Chap. 11] for the properties of planar graphs.) In this paper we discuss the problem of finding a maximum flow in planar networks.

Section 2 deals with (s, t) planar networks (s and t are on the same face of G). Berge [2, p. 190] proposed an algorithm to find a maximum flow, a straightforward implementation of which requires $O(n^2)$ time. Here, an $O(n \log n)$ implementation is presented. It is also shown that $O(n \log n)$ is a lower bound to any implementation of

* Received by the editors February 7, 1977, and in revised form July 6, 1978.

† Department of Computer Science, Technion—Israel Institute of Technology, Haifa, Israel.

‡ Department of Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel.

Berge's algorithm. (An $O(n \log n)$ algorithm to find a minimum (s, t)-cut for this case appears in [8, p. 151]; however, this algorithm does not produce the flow function itself.)

In § 3, for $D > 0$ we find a flow of value D in a directed planar network if such a flow exists, otherwise we indicate this fact. This algorithm requires $O(n^2 \log n)$ time.

In undirected graphs, let u–v denote an undirected edge between the vertices u and v. A flow network is *undirected* if the graph is symmetric, i.e. if $u \to v \in E$ then also $v \to u \in E$ and $c(u \to v) = c(v \to u)$. In this case G is considered to be undirected (each pair of directed edges $u \to v$ and $v \to u$ is replaced by the undirected edge u–v with the same capacity).

In § 4, we present an $O(n^2 \log n)$ algorithm for finding a minimum (s, t) cut in an undirected planar network. Thereby, a maximum flow in an undirected network may be found in $O(n^2 \log n)$ time.

The Appendix contains an alternative proof of the validity of Berge's algorithm.

1.3. Data structures. Throughout the paper we assume that the graph G has a fixed planar representation.

The graph is represented by incidence lists, i.e. each vertex v has a list E_v of all the edges to which v is incident (edges of the form $u \to v$ or $v \to w$).

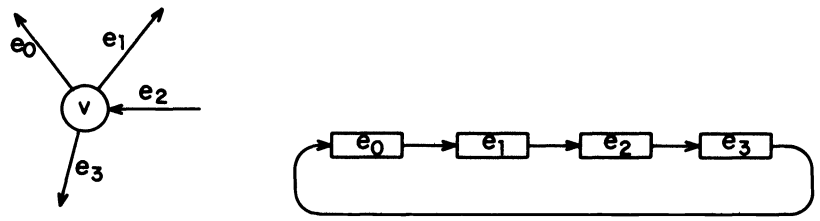


FIG. 1

The set E_v is represented by a circular list corresponding to the circular clockwise ordering of the edges around v (see Fig. 1). Each edge $e \in E_v$ has a unique successor edge $\text{succ}_v(e)$ in E_v. The lists E_v are used to find successor edges. In the course of the algorithm some edges are deleted from the network. The deletion of an edge from E_v is deferred to the time it is traversed when looking for a successor edge. At this time the predecessor edge is known; consequently, singly linked lists suffice. Each edge induces a linear order on E_v as follows:

$$e_0 = e, \qquad e_i = \text{succ}_v(e_{i-1}); \qquad i = 1, \cdots, |E_v| - 1.$$

2. Maximum flow algorithm on (s, t) planar networks. This sections deals with (s, t) planar networks, i.e. s and t belong to the same face, and can be connected by an edge without violating the planarity. Without loss of generality, $t \to s \in E$, (otherwise it may be added with zero capacity). We also assume that $t \to s$ is incident with the exterior face.

$P = (v_0, \cdots, v_k)$ is a directed (v_0, v_k)-*path* if $v_{i-1} \to v_i \in E$, $i = 1, \cdots, k$. A path is *simple* if all its vertices are distinct. Let $P_1 = (s = v_0, \cdots, v_k = t)$ and $P_2 = (s = u_0, \cdots, u_e = t)$ be two simple (s, t) paths. P_1 *lies above* P_2 if $v_i = u_i$, $i = 0, \cdots, r$, $u_{r+1} \neq v_{r+1}$ and $v_r \to v_{r+1}$ precedes $v_r \to u_{r+1}$ in the linear order of E_{v_r} induced by $v_{r-1} \to v_r$. (If $r = 0$ then the order on E_s is induced by $t \to s$.)

The "lies above" relation is a full anti-symmetric order relation on the set of all simple (s, t)-paths. Hence, it has a unique maximum the *uppermost path*. (See Fig. 2.)

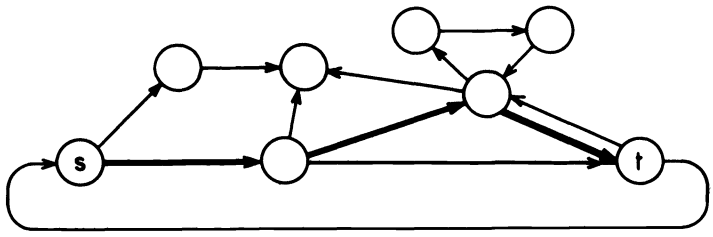


FIG. 2. *The uppermost path appears in boldface.*

2.1. Berge's algorithm. If s and t are on the exterior face, maximum flow may be found by Berge's algorithm. The algorithm starts by pushing as much flow as possible through the uppermost path. Thereby, at least one edge becomes saturated. Such an edge is deleted, and the process is repeated using the uppermost path of the resultant graph.

The algorithm uses the *residual capacities*: $\text{res}(e) = c(e) - f(e)$, where f denotes the flow found thus far by the algorithm.

Let P be an (s, t) path, an edge $e^B \in P$ is a *bottleneck* if $\text{res}(e^B) = \text{Min}_{e \in P} \text{res}(e)$. The bottleneck value is $\text{res}(e^B)$.

BERGE'S ALGORITHM.

1. Initialize: set $i = 1$;
   start with zero flow:
   for all $e \in E$ set $f_0(e) = 0$, $\text{res}(e) = c(e)$.
2. Find the uppermost path P_i^B, if none exists then stop.
3. Let e_i^B be a bottleneck of P_i^B.
4. Increase the flow by $\text{res}(e_i^B)$ units along P_i^B:

$$f_i^B(e) = \begin{cases} f_{i-1}^B(e) + \text{res}(e_i^B) & \text{if } e \in P_i^B \\ f_{i-1}^B(e) & \text{otherwise} \end{cases}$$

$$\text{res}(e) = c(e) - f_i^B(e).$$

5. Delete the bottleneck e_i^B from G.
6. Set $i = i + 1$ and go to 2.

The algorithm is illustrated in Fig. 3.

A proof of the validity of Berge's algorithm can be found in [2]. See the Appendix for an alternative self-contained proof.

A straightforward implementation of Berge's algorithm (even step 4 alone) requires $O(n^2)$ time for the network of Fig. 4. (Note that all the algorithms mentioned in the introduction require $O(n^2)$ time for this network.)

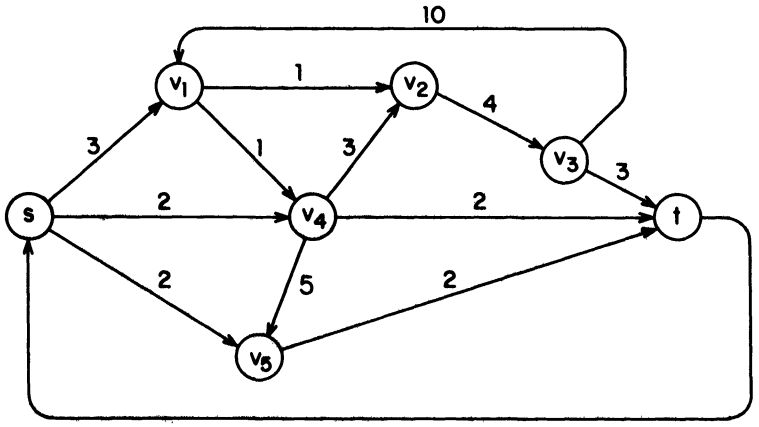
Let $I(e)$ and $L(e)$ denote the index of the first and last uppermost paths in which the edge e participates. The following lemma reveals a useful property of Berge's algorithm; its proof follows from Lemma 2.5 below.

LEMMA B. *If e participates in any uppermost path then e participates in all the paths between $P_{I(e)}^B$ and $P_{L(e)}^B$.*

COROLLARY. *Let $e \in E$ and $I(e) \leqq i \leqq L(e)$ then $f_i^B(e) = |f_i^B| - |f_{I(e)-1}^B|$.*

The proof follows immediately by induction on i using Lemma B.

2.2. The modified capacity method. We propose an $O(n \log n)$ implementation of Berge's algorithm. To this end, we use modified capacities instead of residual capacities.



The capacities are depicted above the edges:

i	The uppermost path P_i^B	Residual capacity	Bottleneck	$\lvert f_i^B \rvert$
1	(s, v_1, v_2, v_3, t)	(3, 1, 4, 3)	$v_1 \to v_2$	1
2	$(s, v_1, v_4, v_2, v_3, t)$	(2, 1, 3, 3, 2)	$v_1 \to v_4$	2
3	(s, v_4, v_2, v_3, t)	(2, 2, 2, 1)	$v_3 \to t$	3
4	(s, v_4, t)	(1, 2)	$s \to v_4$	4
5	(s, v_5, t)	(2, 2)	$v_5 \to t$	6

FIG. 3

Let f_i^M denote the flow after finding the ith uppermost path, then the *modified capacity* is defined by $M(e) - |f_{I(e)-1}^M| + c(e)$. Note that the modified capacity of each edge receives a value once in the algorithm and is not updated (in contrast to the residual capacity which is updated in each iteration). The flow at each iteration, is not found explicitly for each edge only its value, $|f_i^M|$, is found.

ALGORITHM M.

1. Initialize: set $|f_0^M| = 0$; $P_0^M = \varnothing$; $i = 1$.
2. Find the uppermost path P_i^M, if none exists then go to 7.
3. For $e \in P_i^M - P_{i-1}^M$, set $M(e) = c(e) + |f_{i-1}^M|$.
4. Find a bottleneck $e_i^M \in P_i$. $M(e_i^M) = \text{Min}_{e \in P_i^M} M(e)$; set $|f_i^M| = M(e_i^M)$.
5. Delete e_i^M from E.
6. Set $i = i + 1$ and go to 2.

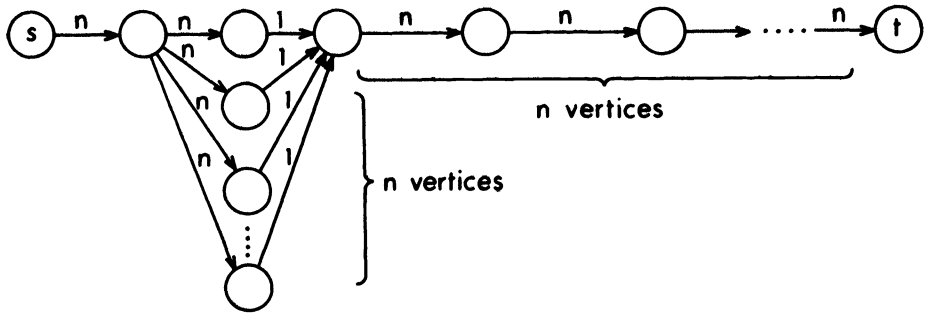
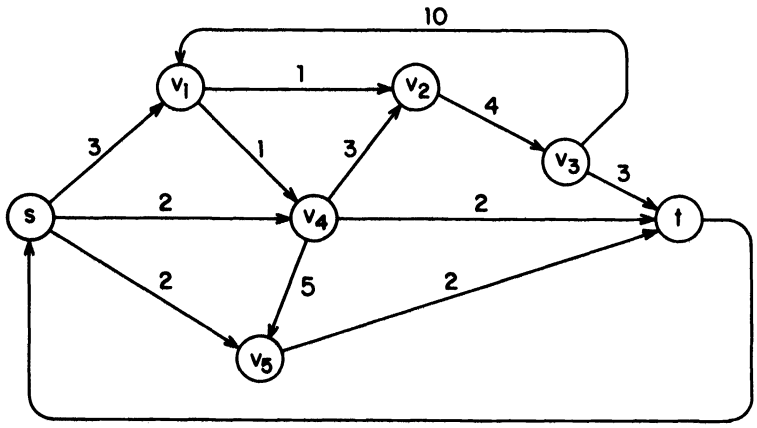


FIG. 4

7. Find the flow of each edge: set

$$f^M(e) = \begin{cases} 0 & \text{if } e \text{ does not belong to any uppermost path,} \\ |f^M_{L(e)}| - |f^M_{I(e)-1}| & \text{otherwise.} \end{cases}$$

Algorithm M as applied to the network of Fig. 3 is illustrated in Fig. 5.



i	The uppermost path	Modified capacities of the path	The bottleneck	$\lvert f^M_i \rvert$
1	(s, v_1, v_2, v_3, t)	(3, 1, 4, 3)	$v_1 \to v_2$	1
2	$(s, v_1, v_4, v_2, v_3, t)$	(3, 2, 4, 4, 3)	$v_1 \to v_4$	2
3	(s, v_4, v_2, v_3, t)	(4, 4, 4, 3)	$v_3 \to t$	3
4	(s, v_4, t)	(4, 5)	$s \to v_4$	4
5	(s, v_5, t)	(6, 6)	$v_5 \to t$	6

e	$I(e)$	$L(e)$	$f(e)$
$s \to v_1$	1	2	2
$s \to v_4$	3	4	2
$s \to v_5$	5	5	2
$v_1 \to v_2$	1	1	1
$v_1 \to v_4$	2	2	1
$v_2 \to v_3$	1	3	3

e	$I(e)$	$L(e)$	$f(e)$
$v_3 \to v_1$	—	—	0
$v_3 \to t$	1	3	3
$v_4 \to v_2$	2	3	2
$v_4 \to v_5$	—	—	0
$v_4 \to t$	4	4	1
$v_5 \to t$	5	5	2
$t \to s$	—	—	0

FIG. 5

The following lemma shows that the two algorithms are equivalent.

LEMMA 2.1. *Let f^B_1 be the flow found in the ith iteration of Berge's algorithm. Let $P^B_1, \cdots, P^B_k$ be the uppermost paths found in Berge's algorithm, $P^M_1, \cdots, P^M_l$ the uppermost paths found by algorithm M. If each P^B_i has a unique bottleneck e^B_i then*

i) $k = l$,
ii) $P^B_i = P^M_i$
iii) $e^B_i = e^M_i$
iv) $f^B_i = f^M_i$

} for $i = 1, \cdots, k$.

Proof. By induction on i. If $i = 1$ then since both P^B_1 and P^M_1 are the uppermost path of the same graph G, $P^B_1 = P^M_1$.

At this point, for each $e \in P_1^M$, $\text{res}(e) = c(e) = M(e)$. $M(e_1^M) = \text{Min}_{e \in P_1^M} M(e) = \text{Min}_{e \in P_1^B} \text{res}(e) = \text{res}(e_1^B)$.

Therefore, e_1^M is the unique bottleneck of P_1^B, i.e. $e_1^B = e_1^M$. Also, $|f_1^M| = M(e_1^M) = \text{res}(e_1^B) = |f_1^B|$.

Suppose the lemma is valid for all $j < i$. At this stage, the graph is the same in both algorithms, since by the induction hypothesis the same bottlenecks have been deleted. Both P_i^B and P_i^M are the uppermost path of the same graph; therefore $P_i^M = P_i^B$.

For $e \in P_i^B$

$$\begin{aligned}\text{res}(e) &= c(e) - f_{i-1}^B(e) \qquad \text{(from corollary to Lemma B)}\\ &= c(e) - (|f_{i-1}^B| - |f_{I(e)-1}^B|)\\ &= c(e) + |f_{I(e)-1}^M| - |f_{i-1}^B|\\ &= M(e) - |f_{i-1}^B|.\end{aligned}$$

Since for the edges $e \in P_i^B = P_i^M$, $\text{res}(e)$ and $M(e)$ differ only by a fixed value—$|f_{i-1}^B|$, $M(e_i^B) = \text{Min}_{e \in P_1^M} M(e) = M(e_i^M)$. The equality $e_i^M = e_i^B$ follows from the hypothesis that e_i^B is the unique bottleneck of P_i^B.

Furthermore,

$$|f_1^B| = |f_{i-1}^B| + \text{res}(e_i^B) = |f_{i-1}^B| + (M(e_i^B) - |f_{i-1}^B|) = M(e_i^B) = M(e_i^M) = |f_i^M|.$$

Q.E.D.

If a path P_i has more than one bottleneck, Berge's algorithm does not specify which bottleneck is chosen. Therefore, for any choice of the bottlenecks in Algorithm M there is a corresponding choice in Berge's algorithm such that the sequences of paths, bottlenecks and flow values are identical in both algorithms. Since both algorithms find the same flow, and Berge's algorithms finds a maximum flow, we have:

THEOREM 2.1. *The modified capacity method (Algorithm* M*) finds a maximum flow.*

In order to determine the time complexity of Algorithm M, we must first specify how the uppermost paths, the bottlenecks and the indices $l(e)$ and $L(e)$ are found.

2.3. Finding uppermost paths. Let $P_{i-1} = (s = v_0, \cdots, v_r = t)$ be the $(i-1)$st uppermost path. Deleting a bottleneck $v_j \to v_{j+1}$ from P_{i-1} breaks it into two paths: P^s from s to v_j and P^t from v_{j+1} to t.

Algorithm U below constructs P_i by continuing P^s until it meets P^t (P_1 is found by connecting $P^s = (s)$ and $P^t = (t)$.) To this end, we conduct a partial depth first search from v_j until we reach a vertex of P^t.

ALGORITHM U

1. $P_i = P^s$, $v = v_j$;
2. Let $e = u \to v$ be the edge in P_i which enters v (if $v = s$ then $e = t \to s$).
3. If $E_v = \{e\}$ then (v is a deadend)
   if $v = s$ then stop (no (s, t) path exists).
   Otherwise, (backtrack) set $v = u$; delete e from G and P_i;
   go to 2. (See Fig. 6a.)
4. Let $e' = \text{succ}_v(e)$. If e' enters v (e' is in the wrong direction) delete e' and go to 3.
   (See Fig. 6b.)
5. (In this case $e' = v \to w$.) If $w \notin P_i \cup P^t$ then include e' in P_i, set $v = w$ and go to 2.
   (See Fig. 6c.)

6. If $w \in P^t$ (the desired path has been found) include e' in P_i; delete the edges from v_{j+1} to w along P^t; add the remaining edges of P^t to P_i, and stop. (See Fig. 6d.)
7. ($w \in P_i$). Delete the edge e' and the edges P_i between w and v; Set $v = w$ and go to 2. (See Fig. 6e.)

Note that $I(e)$ and $L(e)$ can be found in Algorithm U as follows: Whenever an edge e is included in P_i (step 5 or 6) set $I(e) = i$. If an edge e is deleted in the ith iteration then set $L(e) = i - 1$; if e is not deleted $L(e)$ gets the index of the last uppermost path.

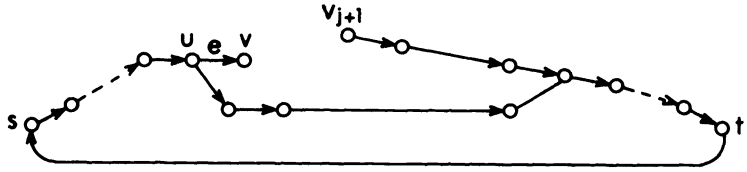


FIG. 6a

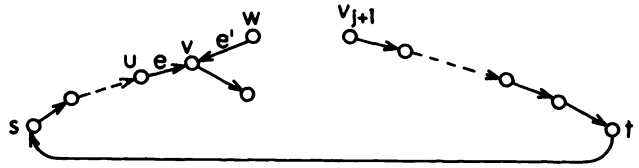


FIG. 6b

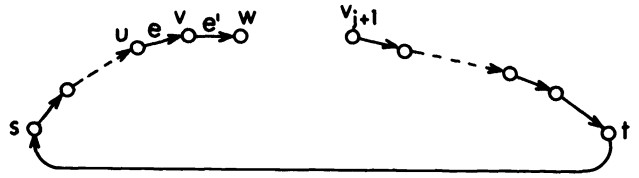


FIG. 6c

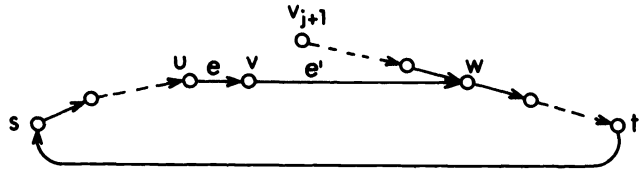


FIG. 6d

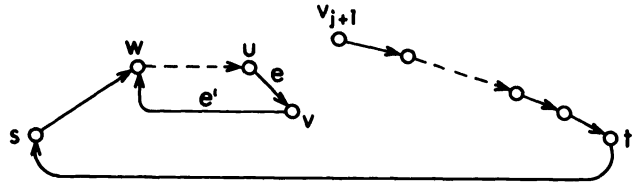


FIG. 6e

FIG. 6

2.4. A validity proof of Algorithm U. An edge e incident with the exterior face is *left-exterior* (l.e.) if it is either incident only with the exterior face, or it is incident also with another face but the exterior face is on its left hand side (see Fig. 7).

Whether an edge is l.e. depends also on the planar representation of G; we choose a particular representation in which $t \to s$ is l.e.

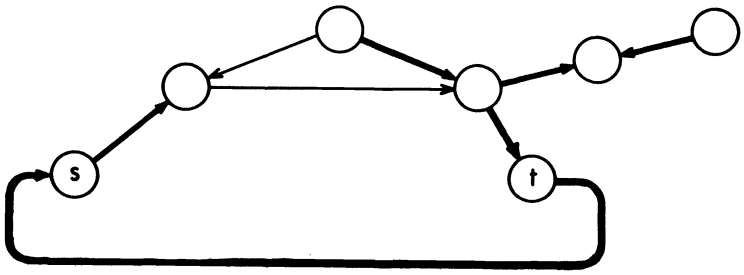


FIG. 7. *The l.e. edges appear in boldface.*

A path is l.e. if all its edges are l.e. The above definition implies the following lemma:

LEMMA 2.2. *If $u \to v$ is an l.e. edge and $v \to w = \mathrm{succ}_v(u \to v)$ then $v \to w$ is also l.e.*

The proof follows immediately from the definition of l.e.

LEMMA 2.3. *Let G_i be the graph resulting after finding the path P_i. If P^s and P^t are l.e. in G_{i-1} then P_i is l.e. in G_i.*

Proof. If $P_i = (s)$ and the edge $s \to w$ is added to P_i then $s \to w = \mathrm{succ}_s(t \to s)$ and therefore is l.e.

Assume that P_i is a nontrivial path, and $u \to v$ is its last edge. When an edge $v \to w$ is added to P_i, $v \to w = \mathrm{succ}_v(u \to v)$ and by Lemma 2.2, $v \to w$ is also l.e. The algorithm may delete edges but if an edge is l.e., then the deletion of other edges does not change this property.

The edges of P^t added to P_i (at step 6) are l.e. since P^t was l.e. in G_{i-1}. Q.E.D.

COROLLARY. *Every path P_i found by Algorithm* U *is l.e. in G_i.*

Proof. By induction on i. For $i = 1$, $P^s = (s)$, $P^t = (t)$ and the premise of Lemma 2.3 holds. In general, assume that P_{i-1} is l.e. Deleting the bottleneck of P_{i-1} yields $P^s, P^t \subseteq P_{i-1}$ which are also l.e. and by Lemma 2.3 P_i is also l.e. Q.E.D.

LEMMA 2.4. *If v_1, v_2, v_3 and v_4 are on the exterior face in this cyclic order then every (v_1, v_3)-path and every (v_2, v_4)-path have a common vertex.*

Proof. Assume to the contrary that P_1 and P_2 are disjoint (v_1, v_3)- and (v_2, v_4)-paths. Add a vertex v_5 in the exterior face and the edges $v_5 \to v_i$, $i = 1, \cdots, 4$. Then the resulting graph is both planar and contractible to K_5—a contradiction (see Fig. 8). Q.E.D.

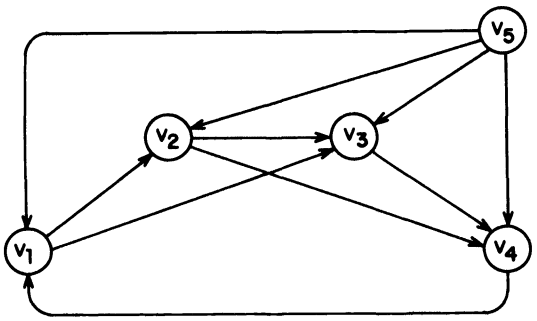


FIG. 8

LEMMA 2.5. *Eevery edge deleted by Algorithm* U *cannot participate in any subsequent uppermost path.*

Proof. Edges are deleted in four places:

i) (step 3). The vertex v is a deadend and no (s, t)-path can pass through v; therefore, $e = u \to v$ is useless (Fig. 6a).

ii) (step 4). Let $e' = w \to v$. Since $e \in P_i$, is an l.e. edge, (corollary to Lemma 2.3). $e' = \text{succ}_v(e)$ and therefore e' is incident with the exterior face. Since $t \in P_{i-1}$, s, v, w and t are on the exterior face in this cyclic order. Thus, by Lemma 2.4, every directed (s, t) path which uses $w \to v$ must cross itself. This property is not changed when edges are deleted. Therefore, any subsequent (s, t) path containing $v \to w$ is not simple and is not uppermost (Fig. 6b).

iii) (step 6). If edges are deleted in this step then $v_{j+1} \neq w$ and w is incident with three l.e. edges. Consequently, w is an articulation point separating the deleted edges from the vertices s and t, and any (s, t) path which uses any of the deleted edges is not simple (Fig. 6d).

iv) (step 7). Since the edge $v \to w$ is an l.e. edge then w is an articulation point and there is no simple (s, t)-path through any vertex x which belongs to the directed cycle closed by $v \to w$, (Figure 6e). Q.E.D.

The above lemmas yield:

THEOREM 2.2. *If there exists an* (s, t)*-path then Algorithm* U *finds the uppermost path.*

Proof. If there exists an (s, t)-path there exists an uppermost path. By Lemma 2.5 after deleting edges there still exists an (s, t)-path. In this case the algorithm terminates in step 6 and a path is returned. By the corollary to Lemma 2.3 this path is l.e. It is easy to see that any l.e. (s, t)-path is uppermost. Therefore, the path is the uppermost path of the resultant graph. Since by Lemma 2.5 only useless edges are deleted, this path is also the uppermost path of the initial graph. Q.E.D.

At this point we wish to make a few observations. Algorithm U finds the uppermost paths and can be used both in Berge's Algorithm and Algorithm M. The validity of Berge's Algorithm does not depend upon the method by which the uppermost paths are found. However, since by a proper choice of bottlenecks every method yields the same sequence of uppermost paths, Algorithm U may be used to prove properties of Berge's Algorithm, in particular Lemma B above.

Proof of Lemma B. It suffices to prove that if $e \in P_i$, $e \notin P_{i+1}$, then $e \notin P_j$ for $j > i$. If e is the bottleneck of P_i then it is deleted by Berge's Algorithm and cannot participate in any subsequent uppermost path. Otherwise, e is deleted by Algorithm U, and by Lemma 2.5 cannot participate in any subsequent uppermost path. Consequently, $e \notin P_j$ for $j > i$. Q.E.D.

Note that Lemma B is a property of Berge's Algorithm, not of Algorithm U. Therefore, it may be used to show the equivalence of Berge's Algorithm and Algorithm M.

2.5. Efficient implementation of Steps 5–7 of Algorithm U. To obtain an $O(n \log n)$ algorithm, Steps 5–7 must be implemented efficiently.

Step 5. In this step we should identify the new vertices (those vertices which have not appeared in P_i or any previous uppermost path). To this end, on initialization (step 1 of Algorithm M) we mark vertices s and t as **old** and all other vertices **new**. Step 5 should be:

5. If w is **new**, then: include e' in P_i,
mark w **old**, set $v = w$ and go to 2.

The paths P_i and P^t are represented as follows:
Every vertex belongs to at most one of the paths P_i or P^t. Every vertex x has one pointer field. If $x \in P_i$ then the pointer points to its predecessor in P_i; if $x \in P^t$ then it points to its predecessor in P^t.

Steps 6, 7. Here we should determine whether an **old** vertex w is in P^t or P_i. This is done by backtracking along the back pointers. If $w \in P^t$ then the backtracking from w stops when we encounter v_{j+1} and the backtracking from v stops when s is met. If $w \in P_i$ then when backtracking from w, s is encountered and when backtracking from v, w is encountered. If the backtracking is done from v and w in parallel and stopped when the first terminating condition is met, the number of edges processed is at most twice the number of edges deleted in Steps 6 and 7.

LEMMA 2.6. *The number of edge traversals in Algorithm* M (*insertions to an uppermost path, deletions from the graph and backtracking*) *is proportional to the number of edges.*

Proof. Each edge may be inserted and deleted at most once. An edge is traversed at insertion or deletion, and at backtracking. From the previous discussion, the total number of edge traversals caused by backtracking is at most twice the number of deletions, and thus it is also linear. Q.E.D.

2.6. The complexity of Algorithm M. In order to find a bottleneck efficiently, we use a priority queue. A priority queue [10] is a data structure to which we may insert or delete an element in $O(\log q)$ time (q is the number of elements in the queue), and find the minimum in constant time. We keep the modified capacities of the edges of the current P_i and P^t, in the same priority queue. Edges are inserted to the priority queue, when added to P_i in Steps 5 and 6 of Algorithm U. Whenever an edge of the graph is deleted, it is deleted also from the priority queue (provided it was there). Each edge is inserted and deleted at most once. Therefore, there may be at most m edges on the queue, and the entire deletion and insertion time is $O(m \log m) = O(n \log n)$. By Lemma 2.6 this bound also dominates the execution of the entire algorithm. Consider the graph of Fig. 9. The c_i's are the bottlenecks. In any implementation of Berge's Algorithm they are found in an increasing order. Therefore, Berge's Algorithm may be used to sort $\{c_1, \cdots, c_n\}$. Hence, Berge's Algorithm (in any implementation which uses comparisons to find the bottleneck) requires at least $O(n \log n)$ time.

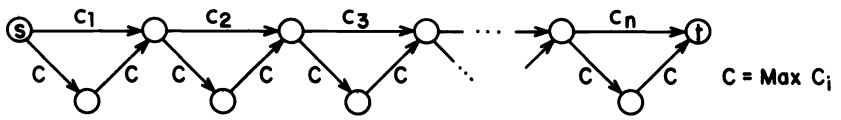


FIG. 9

3. Finding a flow in a general planar network.

3.1. Preliminaries. Let N be a general planar network (i.e. s and t are not necessarily on the same face) and let $D \in R^+$. We wish to find a flow f of value D in N. Algorithm G, described below, finds f if it exists, otherwise, the algorithm terminates indicating that there is no such flow. The algorithm requires at most $O(n^2 \log n)$ time. The Max Flow-Min Cut theorem [6] implies that such a flow exists iff $D \leqq C$—the value of a minimum cut. However, we did not find an $O(n^2 \log n)$ algorithm to determine C in a general directed planar network. In § 4 we present an $O(n^2 \log n)$ algorithm to find a minimum cut in an undirected planar network.

A function $f: E \to R^+$ is a *pseudo-flow* if it satisfies the conservation rule. Since the capacity rule is not necessarily satisfied, a pseudo-flow is not necessarily a flow. An edge

e is *over-flowed* (with respect to a pseudo-flow f) if $f(e)<c(e)$.If $e=u\to v$, then $\tilde{e}$ denotes the edge $v\to u$. We make use of two conventions concerning the edges $e, \tilde{e}$:

i) If $e\in E$ then also $\tilde{e}\in E$ ($\tilde{e}$ may be added with zero capacity).

ii) If a flow (pseudo-flow) passes through e, no flow passes through $\tilde{e}$ (i.e. if $f(e)>0$ then $f(\tilde{e})=0$).

Let f_1, f_2 be pseudo-flows; the pseudo-flows f_1+f_2 and f_1-f_2 are defined by:

$$(f_1\pm f_2)(e)=\text{Max}\{0, f_1(e)-f_1(\tilde{e})\pm(f_2(e)-f_2(\tilde{e}))\}.$$

Therefore, if for example, $f_1(e)=3$, $f_2(\tilde{e})=5$, then

$$(f_1+f_2)(e)=0, \qquad (f_1-f_2)(e)=8,$$
$$(f_1+f_2)(\tilde{e})=2, \qquad (f_1-f_2)(\tilde{e})=0.$$

3.2. General planar flow algorithm. Algorithm G starts with an initial pseudo-flow the value of which is equal to D.

At each stage we pick an over-flowed edge $x\to y$ and construct a new pseudo-flow of the same value. The new pseudo-flow satisfies the capacity rule for the edges which satisfied it before, as well as for the edge $x\to y$.

ALGORITHM G.

1. Find a shortest (s, t)-path, P.
2. Let f be the pseudo-flow obtained by pushing D units of flow through P.
3. Choose an over-flowed edge $e_0=x\to y$. If none exists stop—f is a legal flow of value D.
4. Let $N'=(G', x, y, c)$ where $G=(V, E')$, $E'=E-\{e_0, \tilde{e}_0\}$

   and
   $$c'(e)=\begin{cases}0 & \text{if} \quad f(e)>c(e),\\ c(e)-f(e) & \text{if} \quad c(e)\geqq f(e)>0,\\ c(e)+f(\tilde{e}) & \text{otherwise } (f(e)=0).\end{cases}$$

   Find a flow f' in N' such that $|f'|=f(e_0)-c(e_0)$. If none exists then stop, there exists no flow of value D in N.
5. Set $f'(\tilde{e}_0)=|f'|$; $f=f+f'$; go to 3.

3.3. The validity and complexity of Algorithm G. In this section we prove the following theorem:

THEOREM 3.1. *Let N be a general planar network and $D\in R^+$.*

i) *If there exists a flow of value D in N then Algorithm* G *finds one.*

ii) *If there exists no such flow then Algorithm* G *terminates indicating this fact (at step* 4).

iii) *Algorithm* G *requires at most $O(n^2\log n)$ time.*

First, we show that the algorithm always terminates.

LEMMA 3.1. *Let p denote the number of edges of the path P (found in Step* 1), *then the number of iterations of Algorithm* G *is bounded by p.*

Proof. From the definition of c' it follows that if an edge e satisfied the capacity rule for f, then after updating f in Step 5 the rule is still satisfied, i.e.

$$\text{if } f(e)\leqq c(e) \quad \text{then} \quad (f+f')(e)\leqq c(e).$$

Moreover, after the execution of Step 5, the edge e_0 also satisfies the capacity rule ($f(e_0)\leqq c(e_0)$). Consequently, after each iteration the number of over-flowed edges strictly decreases. Since there are at most p such edges, the number of iterations is bounded by p. Q.E.D.

The proof of the theorem depends on the following lemma.

LEMMA 3.2. *If $D \leqq C$ then in Step 4 there exists a flow f' in N' of value: $|f'| = f(e_0) - c(e_0)$.*

Proof. Let f^D be a flow of value D in N. Define $f^* = f^D - f$. $f^*_{|E'}$ (f^* restricted to E') is a flow in N': Since $|f| = |f^D| = D$, f^* satisfies the conservation rule at s and t as well as for all the other vertices. The capacity rule is satisfied because of the definition of c'.

Since f^* satisfies the conservation rule at x, the value of $f^*_{|E'}$ is:

$$|f^*_{|E'}| = f^*(\tilde{e}) - f^*(e) = (f^D(\tilde{e}) - f(\tilde{e})) - (f^D(e) - f(e))$$
$$= f^D(\tilde{e}) + f(e) - f^D(e) \geqq f(e) - f^D(e) \geqq f(e) - c(e) > 0.$$

Since N' has a flow, the value of which is at least $f(e) - c(e)$, it also has a flow f' of value $f(e) - c(e)$. Q.E.D.

Proof of Theorem 3.1.

i) If $D \leqq C$ then there exists a flow of value D in N. By Lemma 3.2 the algorithm terminates at Step 3, when no over-flowed edges exist, i.e. the final f is a flow. Since throughout the algorithm the value of f is not changed, at termination, flow of value D is found.

ii) If $D > C$ then the algorithm cannot terminate in Step 3. Since by Lemma 3.1 the algorithm is finite, it terminates in Step 4, indicating that no flow of value D exists.

iii) We bound the execution time of each step.
Step 1. requires $O(m) = O(n)$ time;
Step 2. $O(p) \leqq O(n)$ time;
Step 3–5. are executed at most p times. On each iteration, Step 3 requires at most $O(1)$ time.

In Step 4 a flow f' of value $f(e) - c(e)$ is required. To find f', N' is augmented by the vertex x_s and the edge $x_s \to x$ of capacity $f(e) - c(e)$.

Let $f^{\max}$ be a maximum flow from x_s to y. If $|f^{\max}| = f(e) - c(e)$ then the desired flow is $f^{\max}$ restricted to E'. Otherwise, $|f^{\max}| < f(e) < c(e)$, there exists no flow f', and the algorithm immediately terminates.

In N', x and y are on the same face. Hence, there exists a planar representation of the augmented network, in which x_s and y are also on the same face. Therefore, we may use Algorithm M to find $f^{\max}$ in $O(n \log n)$ time. Consequently, Step 4 requires $O(n^2 \log n)$ time.

Hence, the complexity of Algorithm G is $O(pn \log n) \leqq O(n^2 \log n)$. Q.E.D.

Note that in some cases a shorter initial path can be found by adding edges of zero capacity.

4. Finding a minimum (s, t) cut in an undirected planar network. In this section we present an $O(n^2 \log n)$ algorithm for finding a minimum (s, t)-cut in an undirected planar network.

Henceforth, we assume that G is triconnected. Otherwise, the graph may be triangulated in linear time using zero capacity edges. (Every triangulated planar graph with more than three vertices is triconnected.) The value of a minimum (s, t)-cut obviously does not change by this process. The minimum cut of the original graph consists of the original edges which participate in a minimum cut of the new graph.

Since G is triconnected, it has a unique dual $G^d = (X, A)$, [11, Chap. 3]. G^d is also triconnected. Let F and Φ denote the set of faces of G and G^d respectively. There exists a 1–1 correspondence between the elements of $V \leftrightarrow \Phi$, $E \leftrightarrow A$ and $F \leftrightarrow X$ (see Fig. 10). Let $\alpha^d \in E$ denote the dual of $\alpha \in A$. The length of an edge $\alpha \in A$ is defined by:

$$l(\alpha) = c(\alpha^d).$$

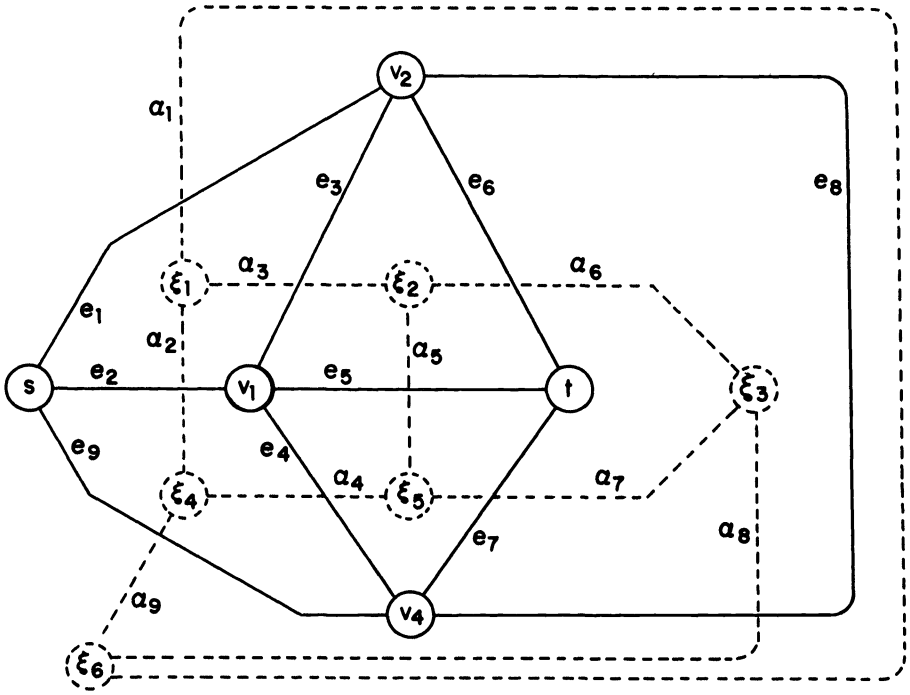


FIG. 10

Let φ_s and φ_t denote the faces in G^d which correspond to s and t respectively. Henceforth, we assume that φ_s is the exterior face of G^d. The following lemma is intuitive; however its formal proof is tedious, and therefore, omitted.

LEMMA 4.1. *If C is a minimum (s, t) cut then $C^d = \{\alpha | \alpha^d \in C\}$ is a cycle of minimum length enclosing φ_t.*

Let $\xi^s \in \varphi_s$, $\xi^t \in \varphi_t$ and let $\Pi = (\xi^s = \xi_1, \cdots, \xi_k = \xi^t)$ be a shortest (ξ^s, ξ^t)-path in G^d. Let $\alpha_i = \xi_{i-1} - \xi_i$ for $i = 2, \cdots, k$.

Let A_Π denote the set of all edges of G^d which have exactly one endpoint on Π. An edge $\xi - \xi_i \in A$ is Π-*left* if it precedes α_{i+1} in the linear order around ξ_i induced by α_i. (See § 1.3.) The edge $\xi - \xi_i$ is Π-*right* if it succeeds, α_{i+1} in this order. Two vertices ξ_0, ξ_{k+1} and two edges $\alpha_0 = \xi_0 - \xi_1$ and $\alpha_{k+1} = \xi_k - \xi_{k+1}$ are added to G^d (see Fig. 11) to make this definition meaningful also for the edges which are incident with $\xi^s = \xi_1$ and $\xi_k = \xi^t$.

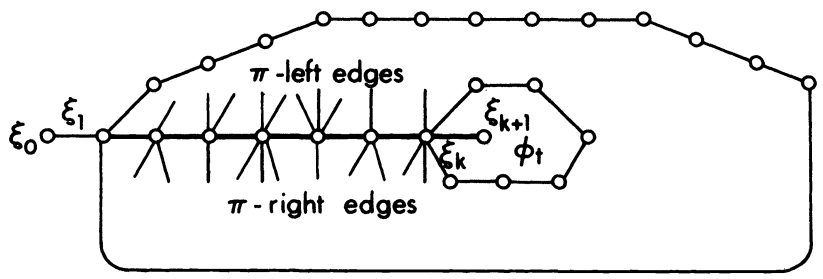


FIG. 11

Note that since G^d is triconnected no edge is both Π-left and Π-right. A *ξ_i-cycle* is a simple cycle which uses exactly one Π-left and one Π-right edge and its Π-left edge is incident with ξ_i, (see Fig. 12).

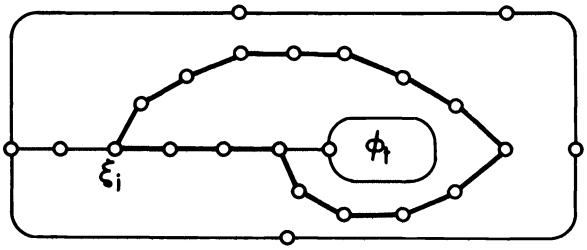


FIG. 12

It is easy to see that every ξ_i-cycle ($i = 1, \cdots, k$) encloses φ_t.

LEMMA 4.2. *Let C be a shortest cycle enclosing φ_t. Then there exists a ξ_i-cycle of the same length.*

Proof. The proof follows immediately from the fact that Π is a shortest (ξ^s, ξ^t) path and therefore a subpath of Π between ξ_i and any ξ_j is a shortest (ξ_i, ξ_j) path. Moreover, every cycle enclosing φ_t must intersect Π. Q.E.D.

(This argument does not work in directed graphs.)

The previous lemma implies that in order to find a minimum cycle enclosing φ_t we may find for each $i = 1, \cdots, k$ a minimum ξ_i cycle. The shortest of these k cycles is a minimum cycle enclosing φ_t. In order to find minimum ξ_i-cycle we use the following construction.

Let $\vec{G}^d$ be the directed graph obtained from G^d in the following manner: Every Π-left edge $\xi_i - \eta$ is directed from ξ_i to η. Every Π-right edge $\xi_i - \eta$ is directed from η to ξ_i. All the other edges $\xi - \eta$ are replaced by two edges $\xi \to \eta$ and $\eta \to \xi$.

LEMMA 4.3. *Let $\xi_i \in \Pi$. If ξ_i is a shortest simple nontrivial directed path from ξ_i to itself in $\vec{G}^d$, then the corresponding undirected edges in G^d form a shortest ξ_i-cycle.*

Proof. It can be easily verified from the definition of $\vec{G}^d$ that if a directed path from ξ_i to itself uses more than one Π-left edge or more than one Π-right edge, then it crosses itself and therefore it is not a shortest ξ_i-cycle. Q.E.D.

Finding a minimum ξ_i-path for a given i is therefore equivalent to finding a shortest nontrivial (ξ_i, ξ_i)-path in $\vec{G}^d$. This can be done in $O(m \log n) = O(n \log n)$ time and therefore the entire algorithm requires at most $O(n^2 \log n)$ time.

5. Conclusions. We have presented an $O(n \log n)$ algorithm to find a maximum flow in an (s, t) planar network. The algorithm was programmed and compared on (s, t) planar networks with Berge's and Dinic's algorithms. On networks which exhibit Dinic's $O(n^3)$ behavior, the special purpose algorithms (Berge's and ours) were superior.

The tests were also conducted on random data. Since it was unclear how random (s, t) planar graphs can be algorithmically constructed, the algorithm was tested on several (s, t) planar graph with random capacities. For these networks the results were less clear cut. The performance of our algorithm and Dinic's were about the same; however there were differences on different networks. Berge's algorithm was superior to both.

This behavior is explained by two observations:

i) The number of augmenting paths found by Dinic's algorithm was much less than the upper bound.

ii) The priority queue involves considerable overhead.

In the general case the value of a maximum flow is equal to that of the minimum cut. We have presented an $O(n^2 \log n)$ algorithm to find the minimum cut in an undirected planar network. Using this algorithm and Algorithm G a maximum flow in an undirected planar network may be found in $O(n^2 \log n)$ time.

We have not found an $O(n^2 \log n)$ method to find the value of the minimum cut for the directed case. However, since Algorithm G indicates whether D is less than or equal to the value of the maximum flow, if the capacities are integers it may be used to find the maximum flow. However, this method requires $\log \sum_{e \in E} c(e)$ iterations of Algorithm G, and hence its complexity is a function of the size of the capacities, as well as the number of vertices. Nevertheless, if the capacities are all small integers the method is superior to the existing algorithms.

Appendix. A validity proof of Berge's algorithm. Let f be a flow in $N = (G, s, t, c)$, $G = (V, E)$; then the graph G_f is defined by:

$$G_f = (V, E_f), \qquad E_f = \{e: e \in E \text{ and } f(e) > 0\}.$$

Let $P = (s = v_0, \cdots, v_k = t)$ be the uppermost path of G, and $e_h = v_h \to v_{h+1}$ for $h = 0, \cdots, k-1$. Let f be a maximum flow such that

$$\text{(A.1)} \qquad \sum_{h=1}^{k} f(e_h) \geqq \sum_{h=1}^{k} f'(e_h) \quad \text{for any maximum flow } f'.$$

LEMMA A.1. *Let e^B be the bottleneck of P then $f(e_h) \geqq c(e^B)$, $(h = 0, \cdots, k-1)$.*

Proof. Assume to the contrary that r is the first index such that $f(e_r) < c(e^B)$. Then

$$\text{(A.2)} \qquad f(e_h) < c(e^B) \quad \text{for } h = r, r+1, \cdots, k-1.$$

We prove (A.2) by induction on h. By hypothesis it is true for $h = r$. Assume it holds for $h = r, r+1, \cdots, j-1$.

If $f(e_j) \geqq c(e^B)$ then $f(e_j) > f(e_r)$ and therefore there exists an (s, v_j) path P_1 in G_f, which does not pass through e_r. Since OUT $(f, v_r) > f(e_r)$ there exists a (v_r, t) path P_2 in G_f, which does not pass through e_r. By Lemma 2.4, P_1 crosses P_2; let x be their common vertex (see Fig. 13).

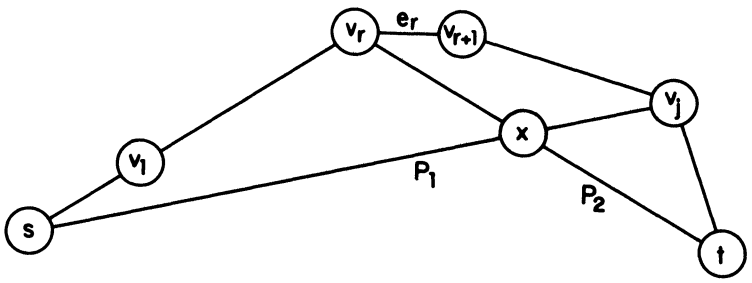


FIG. 13.

Let P_3 be the path in G_f constructed from the subpath of P_2 from v_r to x and the subpath of P_1 from x to v_j. P_3 is a (v_r, v_j) path in G_f. Let P' denote the subpath of P from v_r to v_j. The edge e_r belongs to P' but not to P_3, therefore, $P' \neq P_3$. By the induction hypothesis the edges of P' are not saturated. Thus, we may divert flow from P_3 to P'. The resultant flow f' violates (A.1), this completing the proof of (A.2).

To complete the proof of the lemma, let P_2 be a (v_r, t)-path in G_f; then by diverting flow from P_2 to P, (A.1) is violated. Q.E.D.

THEOREM A.1. *Berge's algorithm finds a maximum flow.*

Proof. By induction on the number of edges:

i) The claim is obvious if the network contains only one edge.

ii) For $m>1$ edges, let e^B be the bottleneck of P, define the flow network $\bar{N}=(G, s, t, \bar{c})$ as follows:

$$\bar{c}(e)=\begin{cases} c(e) & \text{if } e\in E-P, \\ c(e)-c(e^B) & \text{if } e\in P. \end{cases}$$

Let

$$\bar{f}(e)=\begin{cases} f(e) & \text{if } e\in E-P, \\ f(e)-c(e^B) & \text{if } e\in P. \end{cases}$$

By Lemma A.1 $\bar{f}(e)\geqq 0\ \forall e\in P$, and therefore $\bar{f}$ is a legal flow. Obviously, $\bar{f}$ is a maximum flow in $\bar{N}$ and $|\bar{f}|=|f|-c(e^B)$.

In Berge's algorithm we push $c(e^B)$ units of flow through P and then apply the same process on the resultant network—$\bar{N}$ which has at least one edge (e^B) less than N. By the induction hypothesis—the algorithm, applied to $\bar{N}$ finds maximum flow of value $|f|-c(e^B)$.

Consequently, the algorithm applied to N finds a flow of value $(|f|-c(e^B))+c(e^B)=|f|$. That is, Berge's algorithm finds a maximum flow. Q.E.D.

Note added in proof. It was brought to our attention by Professor T. C. Hu that what we call "Berge's Algorithm" was originated by L. R. Ford and D. R. Fulkerson in their paper *Maximal flow through a network*, Canad. J. Math., 8 (1956), pp. 399–404.

REFERENCES

[1] A. E. BARATZ, *Construction and analysis of network flow problem which forces Karzanov algorithm to $O(N^3)$ running time*, MIT Laboratory for Computer Science Report MIT/LCS/TM-83, Mass. Inst. of Tech., Cambridge, 1977.

[2] C. BERGE AND A. GHOUILA-HOURI, *Programming, Games and Transportation Networks*, Methuen, Agincourt, Ontario.

[3] E. A. DINIC, *Algorithm for solution of a problem of maximal flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[4] S. EVEN AND R. TARJAN, *Network flow and testing graph connectivity*, this Journal, 4 (1975), pp. 507–518.

[5] J. EDMONDS AND R. M. KARP, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19 (1972), pp. 248–264.

[6] C. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[7] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[8] T. C. HU, *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

[9] A. V. KARZANOV, *Determining the maximal flow in a network by the method of the preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[10] D. E. KNUTH, *The Art of Computer Programming*, vol. 3, Addison-Wesley, Reading, MA, 1973.

[11] O. ORE, *The Four Color Problem*, Academic Press, New York, 1967.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0005 $01.00/0

PROVABLY DIFFICULT COMBINATORIAL GAMES*

LARRY J. STOCKMEYER† AND ASHOK K. CHANDRA†

Abstract. For a number of two-person combinatorial games, the problem of determining the outcome of optimal play from a given starting position (that is, of determining which player, if either, has a forced win) is shown to be complete in exponential time with respect to logspace-reducibility. As consequences of this property, it is shown that (1) any algorithm which determines the outcome of optimal play for one of these games must infinitely often use a number of steps which grows exponentially as a function of the size of the starting position given as input; and (2) these games are "universal games" in the sense that, if G denotes one of these games and R denotes any member of a large class of combinatorial games (including Chess, Go, and many other games of popular or mathematical interest), then the problem of determining the outcome of R is reducible in polynomial time to the problem of determining the outcome of G.

Key words. computational complexity, combinatorial game, completeness in exponential time

1. Introduction. For many combinatorial games of perfect information (for example, Chess, Go, Kayles, and Nim) it is known that there are algorithms which determine whether or not the player moving first has a "forced win" from a given starting position. We say that such an algorithm *decides* the game. For a game such as Go (generalized to boards of arbitrary size) a position is essentially specified by a placement of stones on a board together with an indication of whose turn it is; a position in Nim is a sequence of nonnegative integers represented in, say, binary notation which specifies the number of sticks in each heap. We are interested primarily in the running times of decision algorithms where the time is measured as a function of the size of the starting position given as input. For example, it would be reasonable to define the size of a position in Go to be the number of squares on the board, and the size of a position in Nim to be the sum of the lengths of the binary representations comprising the sequence of heap sizes.

For both Go and Nim, the number of positions which could conceivably be reached from a given position π by one or more moves of the game grows roughly as an exponential function of the size of π. Therefore, Go and Nim can be decided in exponential time by algorithms which list all positions reachable from the input π and then determine the value of each listed position by the methods of classical game theory [24, § 15]. An exponential running time, while prohibitive in practice for all but very small initial positions, does provide a rough upper bound on the time that is sufficient to decide many examples of games.

For certain games this exponential running time can be substantially improved. The known analysis of Nim [3], [5], [11] yields a decision algorithm for Nim whose running time is a polynomial of low degree. The applications of Grundy-Sprague theory [11] and other clever analyses (see, for example, [5]) have produced nonobvious and efficient decision algorithms for a number of games.

However, other games have resisted analysis. It is not known, for example, if there is a decision algorithm for Go whose running time is bounded above by a polynomial in the board size, and it is possible that no such algorithm exists. Recently it has been proved that the decision problems for Go and Checkers are polynomial-space-hard [10], [18]; this provides evidence (but, as yet, not proof) that these games cannot be decided in polynomial time. The main purpose of this paper is to prove that the decision

* Received by the editors February 6, 1978.
† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

problems for certain simply-defined combinatorial games are complete in exponential time with respect to efficient reducibility (cf. [1], [22]) and, therefore, that these games require exponential time to decide, at least on some infinite sequence of starting positions. Since we have not been able to prove this for existing games such as Go, we have defined several games for the purpose of illustrating the proof methods.

In § 3 we consider games played on propositional formulas. In these games, a starting position is a propositional formula (or formulas) together with an assignment of truth values to the propositional variables in the formula(s). Two players alternate moves. A player moves by changing the truth values of certain variables subject to the rules of the particular game, and the winner is, for example, the player who first makes the formula *true*. Some of these games have more appealing representations. The following game of Peek is equivalent to one of our formula games and illustrates the kinds of results that are contained herein. A starting position in Peek is a box containing a finite number of horizontal plates which can be pushed in or pulled partially out; each movable plate has exactly two positions, "in" or "out", and we may assume that all plates are initially "in". There is also one immobile plate. Some of the movable plates "belong" to player I and the rest "belong" to player II. The plates have holes cut into them at various places and the locations of all holes are known to both players; see Fig. 1. The two players alternate moves with I moving first. A player moves by either passing, pulling one of his plates out, or by pushing one of his plates in. The game ends at the point when a hole appears through the entire stack of plates, and the winner is the player who made the last move which caused the hole to appear. Define the *size* of a position to be the number of plates. We place no a priori bound on the number of plates, but we assume that there is a fixed constant d such that the number of holes in each plate is at most the multiple d of the number of plates. By encoding each starting position as a string of symbols suitable as input to some formal machine model such as a Turing machine, the set of encodings of starting positions from which player I has a forced win is a set of strings which we denote W(Peek). For example, the encoding could contain, for each plate, a list of pairs of positive integers represented in radix notation which specifies the coordinates of all holes in that plate. The key result is that W(Peek) is complete in exponential time with respect to logspace-reducibility. Briefly, this means

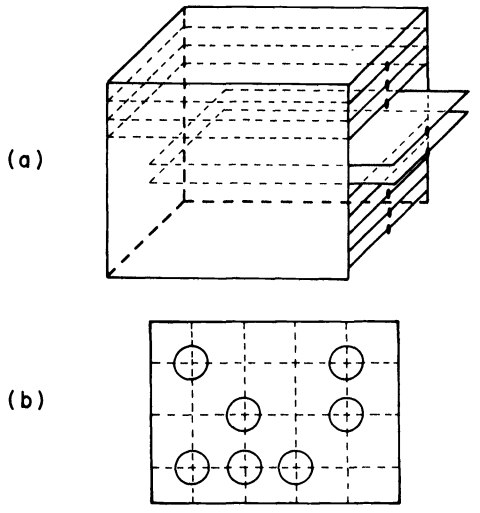


FIG. 1. (a) *A box with ten movable plates, eight "in" and two "out". The top plate is immobile.* (b) *A plate with holes.*

that (i) W(Peek) can be recognized in exponential time, and (ii) if A is any set of strings which can be recognized in exponential time, and (ii) if A is any set of strings which can be recognized in exponential time then A is logspace-reducible to W(Peek)—that is, there is a function f such that $w \in A$ iff $f(w) \in W$(Peek) for all strings w, and f can be computed by a Turing machine within logarithmic space (and, therefore, within polynomial time).

There are two interesting consequences of the fact that W(Peek) is complete in exponential time. First, we are able to derive an exponential lower bound on the time required (infinitely often) to decide Peek. Precisely, there is a constant $c > 1$, such that if M is a deterministic Turing machine which recognizes W(Peek), then there are infinitely many Peek positions π such that M runs for at least $c^{\text{size}(\pi)}$ steps when started on the encoding of π. It should be pointed out that we use Turing machines as our model of algorithm purely for technical convenience in proofs, and that this exponential lower bound (possibly with a different constant $c > 1$) holds for more realistic models such as random access register machines [1], [7]. This is true because there are sufficiently efficient simulations of random access machines by Turing machines.

The second consequence is that Peek is a "universal game" in the sense that the problem of deciding any reasonable game is logspace-reducible to the problem of deciding Peek. Informally, a game is "reasonable" if (i) the number of positions reachable from a given position π within an arbitrary number of moves is bounded above by an exponential function of the size of π, and (ii) it is not onerously difficult to recognize whether a move is allowed by the rules of the game. Many common games such as Chess and Go generalized in any number of ways to arbitrarily large (possibly multidimensional) boards are reasonable in this sense. The formal definition of "reasonable" precedes the statement of Corollary 3.2.

Besides serving as examples of exponential-time-complete games, the formula games of § 3 might be useful in showing that other games are complete in exponential time, in much the same way that the Boolean satisfiability problem [6] was used to show that certain problems are *NP*-complete [16], and the quantified Boolean formula problem [21], [22] was used to show that certain games are complete in polynomial space [8], [20]. To illustrate this, in § 4 we define a type of blocking game played by moving markers on a graph, prove that one of the formula games is logspace-reducible to the blocking game, and conclude that the blocking game is complete in exponential time.

It is instructive to view the results of this paper in the context of previous research [8], [15], [20] concerning the computational complexity of deciding games. One can identify three different types of games corresponding to three levels of complexity. Given a game G and a position π of the game, let *reach*(π) be the number of positions which can possibly be reached from π within an arbitrary number of moves, and let *reach of* G be that function which maps each positive integer s to the maximum of reach(π) taken over all positions π of size s. Games of the first type are those whose reach is bounded above by a polynomial. For example, if a game is played on a graph by moving a single marker from node to node, then the number of reachable positions is at most the number of nodes in the graph. Assuming that the legal moves of the game can be recognized in polynomial time, the straightforward decision algorithm described in the second paragraph of this section shows that any game of the first type can be decided in polynomial time. Jones and Laaser [15] describe a particular game of this type which is *complete* in deterministic polynomial time. Games of the second type are those like Hex and Dots-and-Boxes where players make permanent marks on a board. The distinguishing feature of games of the second type is that if the game is played from a

starting position π, then the game is assured to end after a number of moves which is bounded above by some polynomial in the size of π. However, there is a game of the second type which is not of the first type because its reach grows exponentially (even though the number of positions visited during a *particular* line of play is at most polynomial). By exhaustively examining all possible lines of play from a given starting position, it can be seen that any game of the second type can be decided by an algorithm which uses space (i.e., memory) bounded above by a polynomial in the size of the starting position (assuming again that the legal moves can be recognized in polynomial time). Even and Tarjan [8] and Schaefer [20] exhibit games of the second type which are *complete* in polynomial space. It follows that these polynomial-space-complete games can be decided in polynomial time if and only if any set recognizable in polynomial space is recognizable in polynomial time; this is viewed as providing evidence that these games cannot be decided in polynomial time. Games of the third type are those like Chess and Go where players can move, place, and remove pieces on a board. Games of the third type are those with exponentially bounded reach. There is a game of the third type which is not of the second type because its play lasts an exponential number of moves. Any game of the third type can be decided in exponential time by the straightforward position-listing algorithm. The purpose of this paper is to exhibit games of the third type which are *complete* in exponential time.

The relationship between the three types of games and their decision algorithms is summarized in Fig. 2, where we define the *space* of a game to be the logarithm of its reach (which measures, as a function of the size of π, the number of bits which is sufficient to assign a distinct binary string to each position reachable from π). These are but three instances of a general relationship between time (space) bounded games and space (time) bounded algorithms which is formalized in [4], [17] and outlined in the next section.

TYPE	GAME	ALGORITHM
1	LOGARITHMIC SPACE	POLYNOMIAL TIME
2	POLYNOMIAL TIME	POLYNOMIAL SPACE
3	LINEAR SPACE	EXPONENTIAL TIME

FIG. 2. *The relationship between resource bounds for games and algorithms.*

2. Games, algorithms, and completeness. For a finite alphabet Σ, Σ^* denotes the set of words (i.e., finite strings of symbols) over Σ including the empty word ε; $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. A *language* is a subset of Σ^+ for some finite Σ. For a word $w \in \Sigma^*$, $|w|$ denotes the length of w. $\mathbf{N}$ denotes the nonnegative integers, and $\mathbf{R}$ denotes the real numbers. For a finite set S, card(S) denotes the cardinality of S.

It is convenient to adopt the following definition of game.

DEFINITION. A (*two-person perfect-information*) *game* is a triple (P_1, P_2, R) where P_1 and P_2 are sets, $P_1 \cap P_2 = \varnothing$, and $R \subseteq P_1 \times P_2 \cup P_2 \times P_1$.

In other words, P_1 (P_2) is the set of positions in which player I (II) has the initiative, and R is the set of allowable moves—if $(\pi, \pi') \in R$ and $\pi \in P_1$ ($\pi \in P_2$), then player I (II)

can move from position π to position π' in one move. By convention, a player who is unable to move is declared the loser.

DEFINITION. Let $G = (P_1, P_2, R)$ be a game. Let $W_{-1}(G) = \varnothing$ and for integer $i \geqq 0$ let

$$W_i(G) = W_{i-1}(G) \cup \{\pi \in P_1 | (\exists \pi' \in P_2)[(\pi, \pi') \in R \text{ and } \pi' \in W_{i-1}(G)]\}$$
$$\cup \{\pi \in P_2 | (\forall \pi' \in P_1)[(\pi, \pi') \in R \text{ implies } \pi' \in W_{i-1}(G)]\}.$$

Define

$$W(G) = \bigcup_{i \geqq 0} W_i(G).$$

$W(G)$ is the set of positions from which player I has a "forced win". In particular, $W_0(G)$ is the set of $\pi \in P_2$ from which player II cannot move, and $W_i(G)$ is the set of positions from which I has a forced win in no more than i moves. For certain games G, our objective is to establish bounds on the computational complexity of recognizing the set $W(G)$.

In measuring the computational complexity of sets, our model of computation is the deterministic one-tape Turing machine [1], [13]. We first define a more general device, the alternating Turing machine, which is a useful technical tool in the proofs of our results. The definition of an alternating Turing machine is very similar to that of a nondeterministic Turing machine (cf. [1], [13]) except that some subset of its states are referred to as *universal* states and the rest as *existential* states. Alternating Turing machines are discussed in more detail in [4], [17]. We state here a somewhat simplified version of the definition which is sufficient for the purposes of this paper.

DEFINITION. A one-tape *alternating Turing machine* (ATM) is a seven-tuple $M = (Q, \Gamma, \Sigma, \#, \delta, q_0, U)$ where:

Q	is the set of states;
Γ	is the tape alphabet;
Σ	is the input alphabet, $\Sigma \subseteq \Gamma$;
$\#$	is the blank tape symbol, $\# \in \Gamma - \Sigma$;
δ	$\subseteq (Q \times \Gamma) \times (Q \times (\Gamma - \{\#\}) \times \{\text{L, R, S}\})$ is the next-move relation;
q_0	is the initial state;
U	is the set of universal states, $U \subseteq Q$;
$Q - U$	is the set of existential states.

The tape is assumed to be one-way infinite to the right, and we assume that the head never moves off the left end of the tape.

A *configuration* of M is a triple of the form (q, γ, j) where $q \in Q$ is the current state, $\gamma \in (\Gamma - \{\#\})^*$ denotes the nonblank portion of the tape, and $j \geqq 1$ is an integer which indicates that the jth tape cell from the left end is currently being scanned; let $\mathscr{C}_M$ denote the set of all such configurations. For an input $w \in \Sigma^+$, the *initial configuration on* w is $(q_0, w, 1)$. If M is in state q scanning the symbol $u \in \Gamma$, and if $((q, u), (q', u', d)) \in \delta$, then M can in one step enter state q', print u' on the tape, and shift the head in direction d (Left, Right, or Stationary). For configurations C and C' we write $C \vdash_M C'$ iff C can reach C' in one step as just described; $\vdash_M^*$ denotes the reflexive transitive closure of $\vdash_M$. The configuration (q, γ, j) is a *universal* (*existential*) *configuration* if q is a universal (existential) state.

Several equivalent definitions of acceptance for ATM's are discussed in [4], [9], [17]. The following definition was suggested by M. Fischer and R. Ladner [9].

DEFINITION. Let M be an ATM. A *trace of* M is a set $\mathrm{Tr} \subseteq \mathscr{C}_M \times \mathbf{N}$ such that:

1) if $(C, k) \in \mathrm{Tr}$ and C is a universal configuration, then $(C', k-1) \in \mathrm{Tr}$ for all C' such that $C \vdash_M C'$; and
2) if $(C, k) \in \mathrm{Tr}$ and C is an existential configuration, then there exists a C' such that $C \vdash_M C'$ and $(C', k-1) \in \mathrm{Tr}$.

M accepts $w \in \Sigma^+$ iff there is a trace Tr of M and a $k \in \mathbf{N}$ such that $((q_0, w, 1), k) \in \mathrm{Tr}$; in this case, Tr is said to be an *accepting trace for w*.

Let $L(M)$ denote that subset of Σ^+ which M accepts.

Note. A configuration C is said to be *halting* if there is no C' such that $C \vdash_M C'$. Universal halting configurations serve as "accepting configurations" since such configurations can belong to any trace. Existential halting configurations serve as "rejecting configurations" since such configurations belong to no trace. Thus, the definition of ATM given above need not mention accepting and rejecting states explicitly.

Let $t, s \in \mathbf{N}$. The trace Tr *uses time at most* t iff $k \leqq t$ for all $(C, k) \in \mathrm{Tr}$. The trace Tr uses *space at most* s iff $j \leqq s$ for all $((q, \gamma, j), k) \in \mathrm{Tr}$.

DEFINITION. Let $F: \mathbf{N} \to \mathbf{R}$ and let M be an ATM. *M operates within time (space)* $F(n)$ iff for each $w \in L(M)$ there is a trace Tr of M such that Tr is an accepting trace for w and Tr uses time (space) at most $F(|w|)$. Define

$$\mathrm{ATIME}(F(n))(\mathrm{ASPACE}(F(n))) = \{L(M) \mid M \text{ is an ATM which operates within time (space) } F(n)\}.$$

A *deterministic Turing machine* (DTM) is an ATM M such that for any configuration C of M there is at most one C' such that $C \vdash_M C'$. When restricted to DTM's, the above definitions of time and space bounded acceptance of languages are identical to the usual definitions [1], [13]. (Although nondeterministic Turing machines play no role in this paper, it might aid the reader's intuition to note that one could define a nondeterministic Turing machine to be an ATM with the restriction that every universal configuration is halting.) Define

$$\mathrm{DTIME}(F(n))\,(\mathrm{DSPACE}(F(n))) = \{L(M) \mid M \text{ is a DTM which operates within time (space) } F(n)\}.$$

Also let

$$\mathscr{P}\text{-TIME} = \bigcup_{c,k \geqq 1} \mathrm{DTIME}(cn^k), \qquad \mathscr{P}\text{-SPACE} = \bigcup_{c,k \geqq 1} \mathrm{DSPACE}(cn^k),$$

$$\mathscr{E}\text{-TIME} = \bigcup_{c \geqq 1} \mathrm{DTIME}(c^n).$$

The connection between space-bounded ATM's and time-bounded DTM's is embodied in the following result.

THEOREM 2.1 (Chandra, Kozen, Stockmeyer [4], [17]). *Let* $F(n) \geqq n+1$.

$$\mathrm{ASPACE}(F(n)) = \bigcup_{c \geqq 1} \mathrm{DTIME}(c^{F(n)}).$$

To be completely precise, an ATM is defined in [4], [17] to have a separate input tape, and Theorem 2.1 is proved for all $F(n) \geqq \log n$; the case $F(n) = \log n$ is implicit in Jones and Laaser [15, Thm. 13]. In the case that $F(n) \geqq n+1$, the presence or absence of an input tape is immaterial, so Theorem 2.1 follows trivially from [4], [17]. We are interested primarily in the following corollary of Theorem 2.1.

COROLLARY 2.1. $\text{ASPACE}(n+1) = \mathscr{E}\text{-TIME}$.

In § 3 we prove that the sets $W(G)$ for certain games G are complete in $\mathscr{E}$-TIME by exploiting the natural connection between ATM's and games. Briefly, the existential (universal) configurations of the ATM correspond to positions from which player I (player II) has the initiative to move, and the universal halting configurations correspond to immediate losing positions for II.

Remark. The papers [4], [17] also characterize time-bounded ATM's in terms of space-bounded DTM's. In particular,

$$\bigcup_{c,k\geqq 1} \text{ATIME}(cn^k) = \mathscr{P}\text{-SPACE}.$$

This equality embodies the connection between $\mathscr{P}$-SPACE and "polynomial-time bounded games" (games of the second type in § 1) which is exploited by Even and Tarjan [8] and Schaefer [20] in proving that certain games are complete in $\mathscr{P}$-SPACE.

Finally we define the notion of a language being *complete* in a class of languages. Let $\log n$ denote the base two logarithm for $n \geqq 2$, and $\log 0 = \log 1 = 1$. Let Σ and Δ be finite alphabets. The function $f\colon \Sigma^+ \to \Delta^+$ is *logspace-computable* (cf. [14], [15], [22]) if there is a deterministic Turing machine with a separate two-way read-only input tape, a read/write work tape, and a one-way output tape such that, when started with any word $w \in \Sigma^+$ on the input tape, the machine eventually halts with $f(w)$ on the output tape while having visited at most $\log |w|$ squares on the work tape. Let $l\colon \mathbf{N} \to \mathbf{R}$. The function f is *length $l(n)$ bounded* iff $|f(w)| \leqq l(|w|)$ for all $w \in \Sigma^+$.

Let $A \subseteq \Sigma^+$ and $B \subseteq \Delta^+$. *A transforms to B within logspace via f* ($A \leqq_{\log} B$ *via* f) iff f is a logspace-computable function, $f\colon \Sigma^+ \to \Delta^+$, such that $w \in A \leftrightarrow f(w) \in B$ for all $w \in \Sigma^+$. We remark that the class of logspace-computable functions is closed under composition [14], [22], so that $\leqq_{\log}$ is a transitive relation on languages.

Let B be a language and let $\mathscr{L}$ be a class of languages. Then

$$\mathscr{L} \leqq_{\log} B \quad \text{iff} \quad A \leqq_{\log} B \quad \text{for all } A \in \mathscr{L}.$$

Furthermore, $\mathscr{L} \leqq_{\log} B$ *via length order* $l(n)$ ($l\colon \mathbf{N} \to \mathbf{R}$) provided that for each $A \in \mathscr{L}$ there is a function f and a constant $b \in \mathbf{N}$ such that $A \leqq_{\log} B$ via f and f is length $b \cdot l(n)$ bounded.

The language B is *log-complete in* $\mathscr{L}$ iff both $B \in \mathscr{L}$ and $\mathscr{L} \leqq_{\log} B$.

3. Games on propositional formulas. In this section we describe six games which are played on propositional formulas and prove that their sets of winning positions are log-complete in $\mathscr{E}$-TIME. By *formula* we mean a well-formed parenthesized expression involving variable symbols (which are denoted in the text by (subscripted) letters t, u, v, x, y, z), the binary connectives $\wedge$ (conjunction), $\vee$ (disjunction) and $\oplus$ (exclusive- or), the unary connective $\sim$ (negation), and parentheses. We define the class of formulas and simultaneously define $V(F)$, the set of variable symbols in F, and size(F), the number of occurrences of variable symbols in F.

DEFINITION. 1) If x denotes a variable symbol, then x is a *formula*, $V(x) = \{x\}$, and $\text{size}(x) = 1$;

2) if $F = (G \mathbin{@} H)$ where G and H are formulas and @ denotes a binary connective, then F is a *formula*, $V(F) = V(G) \cup V(H)$, and $\text{size}(F) = \text{size}(G) + \text{size}(H)$;

3) if $F = \sim H$ where H is a formula, then F is a *formula*, $V(F) = V(H)$ and $\text{size}(F) = \text{size}(H)$.

When writing formulas in the text, parentheses are deleted when not needed to determine the precedence of operations. If $X_1, \cdots, X_m$ denote disjoint sets of variable

symbols, we let $F(X_1, \cdots, X_m)$, $H(X_1, \cdots, X_m)$, etc., denote formulas containing only variables in $X_1 \cup \cdots \cup X_m$. For a set S of variable symbols, an *S-assignment* is a function from S to $\{0, 1\}$, where 0 and 1 in this context denote Boolean values *false* and *true*, respectively. A formula F defines, in the obvious way, a function mapping $V(F)$-assignments to $\{0, 1\}$. A *literal* is either x or $\sim x$ where x denotes a variable symbol. A formula is in *conjunctive normal form* (CNF) iff it is a conjunction of disjunctions of literals. A formula is in *disjunctive normal form* (DNF) iff it is a disjunction of conjunctions of literals. For positive integer k, let kDNF denote the set of formulas in DNF which are of the form $C_1 \vee C_2 \vee \cdots \vee C_m$ where each C_i $(1 \leqq i \leqq m)$ is a conjunction of at most k literals; kCNF is defined dually.

We now describe games $G_k = (P_{k1}, P_{k2}, R_k)$ for $1 \leqq k \leqq 6$. We prefer to describe the move-relations R_k informally, and in several cases we indicate the formal definition as well; it should be obvious how to translate these descriptions into complete formal definitions of the R_k as subsets of pairs of positions. In these games, each position contains a symbol $\tau \in \{1, 2\}$ which serves only to differentiate the positions in P_{k1} from those in P_{k2}.

G_1: A position is a triple $(\tau, F(X, Y, \{t\}), \alpha)$ where $\tau \in \{1, 2\}$, F is a formula in 4CNF whose variables have been partitioned into disjoint sets X, Y, and $\{t\}$, and α is a $V(F)$-assignment. Player I moves by setting t to 1 (*true*) and setting the variables in X to any values; player II moves by setting t to 0 (*false*) and setting the variables in Y to any values. A player loses if the formula F is *false* after his move.

G_2: A position is a 4-tuple $(\tau, \text{I-WIN}(X, Y), \text{II-WIN}(X, Y), \alpha)$ where $\tau \in \{1, 2\}$, I-WIN and II-WIN are formulas in 12DNF, and α is an $(X \cup Y)$-assignment. Player I (II) moves by changing the value assigned to at most one variable in X (Y); either player may pass since changing no variable amounts to a "pass". Player I (II) wins if the formula I-WIN (II-WIN) is *true* after some move of player I (II). More precisely, player I can move from (1, I-WIN, II-WIN, α) to (2, I-WIN, II-WIN, α') in one move iff α' differs from α in the assignment given to at most one variable in X and II-WIN is *false* under the assignment α; the moves of player II are defined symmetrically.

G_3: A position is a 4-tuple $(\tau, \text{I-LOSE}(X, Y), \text{II-LOSE}(X, Y), \alpha)$ where $\tau \in \{1, 2\}$, I-LOSE and II-LOSE are formulas in 12DNF, and α is an $(X \cup Y)$-assignment. Player I (II) moves by changing the value assigned to *exactly* one variable in X (Y) (i.e., passing is not allowed). Player I (II) loses if the formula I-LOSE (II-LOSE) is *true* after some move of player I (II). More precisely, player I can move from (1, I-LOSE, II-LOSE, α) to (2, I-LOSE, II-LOSE, α') iff α and α' differ in the assignment to exactly one variable in X and I-LOSE is *false* under the assignment α'.

G_4: A position is a triple $(\tau, F(X, Y), \alpha)$ where F is a formula in 13DNF and τ and α are as in game G_2. Player I (II) moves by changing at most one variable in X (Y); passing is allowed. The game ends at the point when F first becomes *true* and the winner is the player who made the last move which caused F to become *true*. In other words, a player has no legal move from a position in which F is *true*.

G_5: A position is a triple $(\tau, F(X, Y), \alpha)$ where F is a formula and τ and α are as in G_2. Player I (II) moves by changing at most one variable in X (Y); passing is allowed. Player I wins if the formula F ever becomes *true*. In other words, player II cannot move from $(2, F, \alpha)$ to $(1, F, \alpha')$ if F is *true* under α, but I can always move from $(1, F, \alpha)$ to $(2, F, \alpha')$ provided only that α and α' differ in the assignment to at most one variable in X.

G_6: Game G_6 is identical to G_5 except that F is restricted to be in CNF.

Note that the game of Peek described in the Introduction is merely a restatement of that game which is identical to G_4 except that F can be any formula in DNF. Among

these six games G_4 and G_6 stand out because they are played on one formula which is in restricted form (i.e., DNF or CNF) and the artificial "turn variable" t of G_1 is not involved. The games G_1, G_2, and G_5 are included since they arise as useful intermediate steps toward the proofs that G_4 and G_6 are $\mathscr{E}$-TIME-complete. The game G_3, a minor variant of G_2, is used in § 4.

In order to discuss the complexity of the sets $W(G_k)$, we must first encode the positions of these games as words over some fixed finite alphabet. The details of this encoding are for the most part immaterial, and we do not define the encoding formally. We do assume, however, that variable symbols are encoded as words over a finite alphabet by writing subscripts in binary notation; for example, $x_{5,6}$ would be encoded as $x101¢110$. This encoding can be extended in a natural way to give encodings of formulas and assignments (cf. [1], [6], [21]) and ultimately of positions. Since subscripts are written in binary, we assume that there is a constant $e > 0$ such that, if $|F|$ denotes the length of the encoding of the formula F, then

$$|F| \leqq e \cdot \text{size}(F) \cdot \log(\text{size}(F)); \tag{3.1}$$

and

$$\text{card}(V(F)) \leqq e \cdot |F|/\log(|F|); \tag{3.2}$$

and the length of the encoding of an S-assignment is at most $e \cdot \text{card}(S) \cdot \log(\text{card}(S))$. Fix some encoding with these properties, and let $EW(G_k)$ denote the set of encodings of positions in $W(G_k)$.

THEOREM 3.1. *For $1 \leqq k \leqq 6$, $EW(G_k)$ is log-complete in $\mathscr{E}$-TIME.*

Theorem 3.1 is immediate from Lemmas 3.1 and 3.2 below. The first lemma shows that each $EW(G_k)$ belongs to $\mathscr{E}$-TIME.

LEMMA 3.1. *There is a constant $d > 1$ such that*

$$EW(G_k) \in \text{DTIME}(d^{n/\log n}) \quad \textit{for } 1 \leqq k \leqq 6.$$

Proof. We consider the case $k = 1$; the proof in the other cases is virtually identical. Let w be a given input which encodes the position $\pi = (\tau, F, \alpha)$ and let $n = |w|$. Let

$$P = \{(\tau, F, \beta) \mid \tau \in \{1, 2\} \text{ and } \beta \text{ is a } V(F)\text{-assignment}\}.$$

By the convention (3.2) concerning encodings,

$$\text{card}(P) \leqq p = \lceil 2 \cdot 2^{en/\log n} \rceil.$$

The DTM which accepts $EW(G_1)$ first constructs the sets $W_i(G_1) \cap P$ for $0 \leqq i \leqq p$ using repeated application of the inductive definition of W_i, and then checks whether or not $\pi \in W_p(G_1) \cap P$. The time to carry out this procedure is clearly bounded above by some polynomial in np, and the conclusion follows. □

LEMMA 3.2.

$$\mathscr{E}\text{-TIME} \leqq_{\log} EW(G_k) \quad \textit{via length order } n \log n, \quad \textit{for } 1 \leqq k \leqq 5;$$

$$\mathscr{E}\text{-TIME} \leqq_{\log} EW(G_6) \quad \textit{via length order } n^3 \log n.$$

Lemma 3.2 is proved below. Several remarks and corollaries precede the exposition of this proof.

Lemma 3.2 combines with the hierarchy theorem for deterministic time complexity, proved by Hartmanis and Stearns [12], to yield an exponential lower bound on the time required (infinitely often) to accept $EW(G_k)$.

COROLLARY 3.1. *Let $1 \leqq k \leqq 5$. There is a rational constant $c > 1$, such that if a deterministic Turing machine accepts $EW(G_k)$ and operates within time $T(n)$, then $T(n) > c^{n/\log n}$ for infinitely many n.*

Proof. From [12] there is a language A such that $A \in \mathscr{E}$-TIME, and if a DTM accepts A and operates within time $T(n)$ then $T(n) \geqq 2^n$ for infinitely many n. By Lemma 3.2, $A \leqq_{\log} EW(G_k)$ via f where f is length $bn \log n$ bounded for some constant b; say that $b \geqq 1$. Choose $c > 1$ so that $c^b < 2$. Assume for contradiction that there is a DTM which accepts $EW(G_k)$ within time $T(n)$ where $T(n) \leqq c^{n/\log n}$ for almost all n. Since f is computable in polynomial time, it follows that there is a DTM which accepts A within time $T'(n)$ where

$$T'(n) \leqq c^{bn(\log n)/\log(bn \log n)} + p(n)$$

for almost all n, where $p(n)$ is a polynomial. By our choice of c, $T'(n) < 2^n$ for almost all n, contradicting one condition that A was chosen to satisfy. □

By a virtually identical proof one shows that $EW(G_6)$ requires time $c^{(n/\log n)^{1/3}}$ infinitely often.

The following definition and corollary formalize the assertion made in the Introduction that the formula games G_k are "universal games".

DEFINITION. The game $G = (P_1, P_2, R)$ is *reasonable* if:

1) there is a finite alphabet Σ such that $P_1, P_2 \subseteq \Sigma^+$; and

2) for all $\pi, \pi' \in P_1 \cup P_2$, if $(\pi, \pi') \in R$ then $|\pi| = |\pi'|$; and

3) for some symbol $\$ \notin \Sigma$, the language $\{\pi \$ \pi' \mid (\pi, \pi') \in R\}$ belongs to $\mathscr{P}$-TIME.

The condition 1) is a convenience to dispose of the issue of encoding positions as words. The intent of 2) is that each instance of the game be played on a "board" of fixed (but possibly arbitrary) size; for example, in generalized Go (cf. § 1), once a board size has been chosen the players are not permitted to enlarge the board during the course of play. The condition 3) ensures that the legal moves can be recognized in polynomial time.

COROLLARY 3.2. *If the game G is reasonable, then*

$$W(G) \leqq_{\log} EW(G_k) \quad \textit{for } 1 \leqq k \leqq 6.$$

Proof. By the definition of reasonable it is easy to see that $W(G) \in \mathscr{E}$-TIME using the method described in the proof of Lemma 3.1. Now the conclusion is immediate from Lemma 3.2. □

Our aim in this section is simply to illustrate the kinds of games on formulas which are complete in $\mathscr{E}$-TIME rather than to give a full analysis of the various combinations of rules, CNF versus DNF formulas, etc. Two points are worth mention, however. First, the games G_k remain complete in $\mathscr{E}$-TIME if formulas are not restricted to CNF or DNF, or if passing is disallowed, or both. If arbitrary formulas appear in positions of these games, the winning positions can still be accepted within time $d^{n/\log n}$ as the proof of Lemma 3.1 demonstrates. If passing is disallowed, we can give each player an additional variable upon which the formulas do not depend. Secondly, the "turn variable" t appears to be essential to the $\mathscr{E}$-TIME-completeness of games like G_1 where a player can change the assignment to all of his variables in one move. For example, if we define the game G_4' like G_4 except that player I (II) can change the assignment of the entire set $X(Y)$ in one move, then a position $(1, F(X, Y), \alpha)$ belongs to $W(G_4')$ iff F is *false* under α and $(\exists X)[F'(X)]$ where $F'(X)$ is obtained from $F(X, Y)$ by setting the variables in Y according to the assignment α. This problem is trivial for DNF formulas and log-complete in NP [6], [1] for CNF or arbitrary formulas.

If player II has the first move, a position $(2, F(X, Y), \alpha)$ belongs to $W(G_4')$ iff either F is *true* under α or

$$(\forall Y)[\sim F''(Y) \text{ and } (\exists X)[F(X, Y)]]$$

where $F''(Y)$ is obtained from F by setting X according to α. This problem is log-complete in co-*NP* for DNF formulas and log-complete in Π_2^p [21] for CNF or arbitrary formulas.

The remainder of § 3 is devoted to the proof of Lemma 3.2. In this proof it is technically convenient to deal with ATM's of a special type described next. A *standard linear ATM* is an ATM $M = (Q, \Gamma, \Sigma, \not\#, \delta, q_0, U)$ with the properties that: (i) for all $w \in \Sigma^+$, when started on input w, M can reach no configuration in which tape cell $|w|+2$ is being scanned (formally, there do not exist $q \in Q$ and $\gamma \in \Gamma^*$ such that $(q_0, w, 1) \vdash_M^* (q, \gamma, |w|+2)$); (ii) the initial state q_0 is existential; and (iii) if C and C' are configurations of M with $C \vdash_M C'$, then C is existential if and only if C' is universal. Because of the following lemma, we can restrict attention to standard linear ATM's in the sequel.

LEMMA 3.3. $\mathscr{E}$-TIME $= \{L(M) \mid M \text{ is a standard linear ATM}\}$.

Proof. In light of Corollary 2.1 it suffices to observe that if A is accepted by an ATM M which operates within space $n+1$, then M can be modified to satisfy the necessary constraints (i), (ii) and (iii), and still accept A. The constraint (i) can be met by modifying M so that it enters a halting existential (i.e., rejecting) configuration whenever the original M would attempt to shift the head to cell $|w|+2$. Now (ii) and (iii) can be met by introducing new states. Say, if $((p, u), (q, u', d)) \in \delta$ where both p and q are existential states, then remove this element from δ and introduce a new universal state p' such that both $((p, u), (p', u', \text{Stationary}))$ and $((p', u'), (q, u', d))$ belong to δ. □

With each standard linear ATM M we associate a game $G_M = (P_{M1}, P_{M2}, R_M)$ as follows: P_{M1} (P_{M2}) is the set of existential (universal) configurations of M, and R_M is the next-move relation $\vdash_M$.

LEMMA 3.4. *Let M be a standard linear ATM and let G_M be the game associated with M.*

$$L(M) = \{w \in \Sigma^+ \mid (q_0, w, 1) \in W(G_M)\}.$$

Proof. The proof follows easily from the definitions of $W(G_M)$ and of acceptance for ATM's, together with the conventions concerning standard linear ATM's. On the one hand, if Tr is a trace of M then one proves by induction on i that

$$\{C \mid (C, i) \in \text{Tr}\} \subseteq W_i(G_M) \quad \text{for all } i \in \mathbf{N}.$$

On the other hand, it also follows from definitions that

$$\{(C, i) \mid i \in \mathbf{N} \text{ and } C \in W_i(G_M)\}$$

is a trace of M. □

Lemmas 3.3 and 3.4 provide the necessary link between $\mathscr{E}$-TIME and games. Note, in particular, the similarity between G_M and G_1. The proof that $\mathscr{E}$-TIME $\leqq_{\log} EW(G_1)$ follows easily from known methods of expressing a Turing machine's next-move relation $\vdash_M$ as a propositional formula [1], [6], [21]. We next formalize this method.

First, we need a convention for representing a configuration by an assignment to a set of Boolean variables. Let M be a standard linear ATM with states Q and tape

alphabet Γ. With each configuration $C = (q, \gamma_1\gamma_2 \cdots \gamma_l, j)$ where $1 \leqq j \leqq l+1$ and $\gamma_i \in \Gamma - \{\#\}$ for all i, we associate the word

$$\omega(C) = \gamma_1\gamma_2 \cdots \gamma_{j-1} q \gamma_j \gamma_{j+1} \cdots \gamma_l.$$

Let $s = s(M) \overset{\text{def}}{=} \lceil \log(\text{card}(Q \cup \Gamma)) \rceil$. Fix a one-to-one map $h_M: (Q \cup \Gamma) \to \{0, 1\}^s$ such that $h_M(\#) = 0^s$ and extend h_M to a map from $(Q \cup \Gamma)^*$ to $\{0, 1\}^*$ in the obvious way. Let $U = \{u_1, \cdots, u_p\}$ be a set of variable symbols where $p \geqq s \cdot (l+1)$. The U-assignment α *represents* C iff

$$\alpha(u_1)\alpha(u_2) \cdots \alpha(u_{s(l+1)}) = h_M(\omega(C)),$$

and $\alpha(u_i) = 0$ for $s(l+1) < i \leqq p$. It is important to note that, once M and h_M have been fixed, each U-assignment represents at most one configuration.

LEMMA 3.5. *Let M be a standard linear ATM. For each $w \in \Sigma^+$ there is a formula* $\text{NEXT}_{M,w}(U, V)$ *involving the variables $U = \{u_1, \cdots, u_p\}$ and $V = \{v_1, \cdots, v_p\}$ where $p = s(M) \cdot (|w|+2)$ with the following properties. If α_U is a U-assignment and α_V is a V-assignment such that α_U represents a configuration C such that $(q_0, w, 1) \vdash_M^* C$, then* $\text{NEXT}_{M,w}$ *is* true *under α_U and α_V iff α_V represents a configuration C' such that $C \vdash_M C'$. Moreover,*

$$\text{size}(\text{NEXT}_{M,w}) \leqq c_M |w|$$

for some constant c_M depending only on M, and the function which maps w to the encoding of $\text{NEXT}_{M,w}$ *is logspace-computable.*

The proof of Lemma 3.5 is not difficult, and the details (in a slightly different context) can be found in [21, Lemma 6.3]. Briefly, the formula $\text{NEXT}_{M,w}$ is a conjunction of $|w|$ subformulas; the ith subformula checks that the ith, $(i+1)$th, $(i+2)$th symbols of $h^{-1}(\alpha_U)$ and $h^{-1}(\alpha_V)$ are consistent with a legal move of M. This can be done in such a way that the size of each subformula is a constant depending only on M.

The next lemma permits us in certain cases to replace arbitrary formulas by formulas in 3CNF while incurring only a constant factor dilation in formula size. The proof, which is not repeated here, is based on a method of Tseitin [23], (cf. [2], [21]) for converting an arbitrary formula to a formula in 3CNF while preserving satisfiability.

LEMMA 3.6. *There is a constant a, such that for any formula $F(S)$, there is a formula $H(S, Z)$ in 3CNF where Z is a set of variables disjoint from S, such that*

1) $\text{size}(H) \leqq a \cdot \text{size}(F)$, *and*
2) *for any S-assignment α, $F(S)$ is* true *under α iff there exists a Z-assignment β such that $H(S, Z)$ is* true *under the combined assignments α and β.*

Moreover, there is a logspace-computable function which maps each F to an H satisfying 1) *and* 2).

We have now collected the technical machinery to be used in the proof of Lemma 3.2. In each case $1 \leqq k \leqq 6$, given a standard linear ATM M and an input w, we describe a position π_w such that M accepts w iff $\pi_w \in W(G_k)$. We also note how the length of the encoding of π_w depends on $|w|$ (with M fixed). If $g_1, g_2: \mathbf{N} \to \mathbf{R}$, then we write $g_1 = O_M(g_2)$ to assert that there is a constant b_M depending only on M such that $g_1(n) \leqq b_M \cdot g_2(n)$ for all $n \geqq 1$. In each case it is not difficult to see that the function mapping w to the encoding of π_w is logspace-computable (given that the functions described in Lemmas 3.5 and 3.6 are logspace-computable) and we let the reader convince himself that these functions are indeed logspace-computable.

Proof of Lemma 3.2.

1. Let M be a standard linear ATM, let w be an input, and let $n = |w|$. Let p, U, V, and $\mathrm{NEXT}_{M,w}(U, V)$ be as in Lemma 3.5 for this M and w. Let $\mathrm{NEXT}(U, V, Z)$ be the formula obtained by applying Lemma 3.6 with $S = U \cup V$ and $F = \mathrm{NEXT}_{M,w}$. Recall that $\mathrm{NEXT}(U, V, Z)$ is in 3CNF and its size is $O_M(n)$. Say that $Z = \{z_1, \cdots, z_k\}$.

Now we describe a formula $F_1(X_1, Y_1, \{t\})$ where $X_1 = \{x_1, \cdots, x_m\}$, $Y_1 = \{y_1, \cdots, y_m\}$, and $m = p + k$. Let $\mathrm{MOVE}_1(Y_1, X_1)$ denote the formula $\mathrm{NEXT}(U, V, Z)$ after substituting y_i for u_i, x_i for v_i $(1 \leqq i \leqq p)$, and x_{p+j} for z_j $(1 \leqq j \leqq k)$. Let $\mathrm{MOVE}'_1(X_1, Y_1)$ denote $\mathrm{NEXT}(U, V, Z)$ after substituting x_i for u_i, y_i for v_i $(1 \leqq i \leqq p)$ and y_{p+j} for z_j $(1 \leqq j \leqq k)$. Let F'_1 denote the formula

$$(\sim t \vee \mathrm{MOVE}_1(Y_1, X_1)) \wedge (t \vee \mathrm{MOVE}'_1(X_1, Y_1)).$$

F'_1 is easily transformed via the distributive laws to an equivalent formula F_1 in 4CNF with $\mathrm{size}(F_1) = O_M(n)$. Let α_1 be an $(X_1 \cup Y_1)$-assignment such that the restriction of α_1 to $\{y_1, \cdots, y_p\}$ represents the initial configuration of M on input w; the assignment to the other variables can be arbitrary. The position π_w is $(1, F_1, \alpha_1)$. Recalling Lemma 3.4, and noting that the legal moves of G_1 mimic the legal moves of G_M, it should be obvious that $\pi_w \in W(G_1)$ iff M accepts w. For example, say that player I is about to move, so that t must be set to 1. To avoid losing, player I must set the variables in X_1 so that $\mathrm{MOVE}_1(Y_1, X_1)$ assumes the value 1. If C is the configuration of M currently represented by the assignment to $\{y_1, \cdots, y_p\}$ then, by Lemmas 3.5 and 3.6, I must choose the X_1-assignment so that the assignment to $\{x_1, \cdots, x_p\}$ represents a configuration C' with $C \vdash_M C'$. The reasoning is similar in the case that II is about to move, except that $\sim t$ is 1 so $\mathrm{MOVE}'_1(\mathrm{X}_1, \mathrm{Y}_1)$ is enabled.

Since $\mathrm{size}(F_1) = O_M(n)$, it follows from the convention (3.1) concerning encodings that the function mapping w to the encoding of π_w is length $b_M n \log n$ bounded for some constant b_M. Since M was arbitrary, we conclude that $\mathscr{E}\text{-TIME} \leqq_{\log} EW(G_1)$ via length order $n \log n$.

2. We introduce some new terminology which will be useful in this part of the proof. Let $G_2 = (P_1, P_2, R)$, and let $\perp \notin P_1 \cup P_2$. A I-strategy (II-strategy) is a total function $\sigma\colon P_1 \cup \{\perp\} \to P_2 \cup \{\perp\}$ ($\sigma\colon P_2 \cup \{\perp\} \to P_1 \cup \{\perp\}$) such that $\sigma(\perp) = \perp$, and, for all $\pi \in P_1$ ($\pi \in P_2$), if there exists a π' such that $R(\pi, \pi')$ then $R(\pi, \sigma(\pi))$, and if there does not exist a π' such that $R(\pi, \pi')$ then $\sigma(\pi) = \perp$. For a position $\pi_1 \in P_1$, a I-strategy σ_1 and a II-strategy σ_2, define play $(\pi_1, \sigma_1, \sigma_2)$ to be the infinite sequence $\pi_1, \pi_2, \pi_3, \cdots$ where $\pi_{i+1} = \sigma_1(\pi_i)$ for i odd and $\pi_{i+1} = \sigma_2(\pi_i)$ for i even. We say that play$(\pi_1, \sigma_1, \sigma_2)$ *ends* if $\pi_i = \perp$ for some i, and in this case we let last$(\pi_1, \sigma_1, \sigma_2)$ denote that position $\pi_j \neq \perp$ with largest subscript j. For games of perfect information, there is no loss of generality in assuming that players choose their moves by a "strategy" as just defined; for example, it is not difficult to see that, for any game G, the inductive definition of $W(G)$ yields an optimal I-strategy and an optimal II-strategy for G.

Let M be a standard linear ATM and let w be an input. We construct a pair of formulas I-WIN(X_2, Y_2) and II-WIN(X_2, Y_2) and an $(X_2 \cup Y_2)$-assignment α_2 such that the position $\pi_w = (1, \text{I-WIN}, \text{II-WIN}, \alpha_2)$ has the following properties (3.3) and (3.4). If $H(X_2, Y_2)$ is a formula, $\pi = (\tau, \text{I-WIN}, \text{II-WIN}, \alpha)$ is a position, and $b \in \{0, 1\}$, we say that π *satisfies* $H = b$ iff H assumes the value b under the assignment α.

(3.3) If M accepts w then there is a I-strategy σ_1, such that for all II-strategies σ_2:
(a) play$(\pi_w, \sigma_1, \sigma_2)$ ends with last$(\pi_w, \sigma_1, \sigma_2) \in P_2$ (i.e., I wins), and last$(\pi_w, \sigma_1, \sigma_2)$ satisfies I-WIN = 1 and II-WIN = 0; and

(b) if player II passes on some move, then I next moves to a position which satisfies I-WIN = 1 and II-WIN = 0.

(3.4) If M does not accept w then there is a II-strategy σ_2, such that for all I-strategies σ_1:

(a) if play$(\pi_w, \sigma_1, \sigma_2)$ ends then last$(\pi_w, \sigma_1, \sigma_2) \in P_1$ (i.e., II wins), and last$(\pi_w, \sigma_1, \sigma_2)$ satisfies I-WIN = 0 and II-WIN = 1; and

(b) if player I passes on some move, then II next moves to a position which satisfies I-WIN = 0 and II-WIN = 1.

In particular, (3.3)(a) and (3.4)(a) imply that M accepts w iff $\pi_w \in W(G_2)$. The properties (3.3) and (3.4) will be useful in proving cases $k = 4, 5$ of Lemma 3.2 where we construct formulas using I-WIN and II-WIN as subformulas. We there use the fact that the game never ends with both I-WIN = 1 and II-WIN = 1, and that if one player passes then the other player wins immediately on the next move.

Let m be as in part 1 for this M and w. I-WIN and II-WIN contain variables $X_2 = \{x_{i,j}\}$ and $Y_2 = \{y_{i,j}\}$ where $1 \leqq i \leqq 2m+2$ and $j = 1, 2$. The sets of variables $X_c = \{x_{i,1} \mid 1 \leqq i \leqq m\}$ and $Y_c = \{y_{i,1} \mid m+2 \leqq i \leqq 2m+1\}$ play the roles of X_1 and Y_1, respectively, in the previous part. However, since the rules of G_2 allow only one variable to be changed in one move, we must constrain the play so that, for example, while I is changing variables in X_c, player II can only change variables not belonging to Y_c. Similar to the proofs in [8], [20], we describe a *legitimate play* such that if both players play legitimately then it is obvious that (3.3) and (3.4) hold, and a player who departs from legitimate play loses after the next move of the other player. When we say that a player *plays a variable* x we mean that the player changes the truth value of x. *Legitimate play* is described as follows:

$i \leftarrow 1$;
loop: I plays exactly one of $x_{i,1}$ or $x_{i,2}$;
II plays exactly one of $y_{i,1}$ or $y_{i,2}$;
$i \leftarrow$ (if $i < 2m+2$ then $i+1$ else 1);
go to loop.

If we assume legitimate play, then as play progresses from $i = 1$ to $i = m$, player I can assign any values to variables in X_c; then as play progresses from $i = m+2$ to $i = 2m+1$, player II can assign any values to variables in Y_c; and so on.

We next describe formulas I-ILL and II-ILL which punish players I and II, respectively, for illegitimate play. We use the symbols a_i, b_i, a_i', and b_i' to denote certain subformulas of I-ILL and II-ILL. For $1 \leqq i \leqq 2m+2$ define

$$a_i = x_{i,1} \oplus x_{i,2},$$

$$b_i = y_{i,1} \oplus y_{i,2}.$$

If we choose the initial assignment so that the a_i and b_i are all 0 initially, then during legitimate play we always have $a_1 a_2 \cdots a_{2m+2}$ and $b_1 b_2 \cdots b_{2m+2}$ in $0^*1^* \cup 1^*0^*$. The formulas a_i' and b_i' detect the boundaries between the blocks of 0's and 1's.

$$a_i' = \begin{cases} a_{i-1} \oplus a_i & \text{if } 2 \leqq i \leqq 2m+2, \\ \sim(a_{2m+2} \oplus a_1) & \text{if } i = 1. \end{cases}$$

The b_i' are defined similarly in terms of the b_i. If we assume legitimate play then: just before a move of I there is a j such that $a_j' = b_j' = 1$ and $a_i' = b_i' = 0$ for all $i \neq j$; and just before a move of II there is a j such that $a_{j+1}' = b_j' = 1$ and $a_{i+1}' = b_i' = 0$ for all $i \neq j$.

(Here and subsequently, subscripts are evaluated modulo $2m+2$ to lie in the range from 1 to $2m+2$.) Define

$$\text{I-ILL} = \bigvee_{1 \leqq i \leqq 2m+2} ((a'_i \wedge a'_{i+1}) \vee (a'_i \wedge b'_{i+1} \wedge \sim b'_{i-1})),$$

$$\text{II-ILL} = \bigvee_{1 \leqq i \leqq 2m+2} ((b'_i \wedge b'_{i+1}) \vee (b'_i \wedge a'_{i+2} \wedge \sim a'_i)).$$

Note that during legitimate play, both I-ILL and II-ILL remain 0. In addition, these formulas satisfy the following properties.

(3.5) Suppose that both players have played legitimately to a position where II is about to move. Then:
- (a) player II cannot in one move reach a position which satisfies I-ILL = 1; and
- (b) any illegitimate nonpassing move of player II reaches a position which satisfies I-ILL = 0 and II-ILL = 1; and
- (c) if II passes, then I can in one move reach a position which satisfies I-ILL = 0 and II-ILL = 1.

To verify (3.5), consult Fig. 3 which depicts a typical situation where II is about to move; in this example, $a'_{j+1} = b'_j = 1$. First note that in order to reach a position which satisfies I-ILL = 1, the move must change both b'_j to 0 and b'_{j+2} to 1; this is impossible in one move, so (a) is true. If the next move of player II changes b_j, then this move is legitimate. If II changes b_{j-1} then the term $(b'_{j-1} \wedge a'_{j+1} \wedge \sim a'_{j-1})$ of II-ILL becomes 1. If II changes b_l with $l \neq j$ and $l \neq j-1$, then the term $(b'_l \wedge b'_{l+1})$ of II-ILL becomes 1. If II passes, then I changes a_{j+1} from 1 to 0 on the next move, so that the term $(b'_j \wedge a'_{j+2} \wedge \sim a'_j)$ of II-ILL becomes 1. In a completely analogous fashion, one verifies the symmetric version of (3.5) where the players I and II are interchanged and the formulas I-ILL and II-ILL are interchanged.

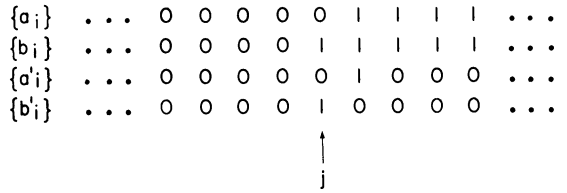


FIG. 3. *A typical situation in G_2 when player* II *is about to move.*

Now let $\text{MOVE}_2(Y_c, X_c)$ and $\text{MOVE}'_2(X_c, Y_c)$ denote the formulas $\text{MOVE}_1(Y_1, X_1)$ and $\text{MOVE}'_1(X_1, Y_1)$ of part 1, after substituting $x_{i,1}$ for x_i and $y_{m+i+1,1}$ for y_i $(1 \leqq i \leqq m)$ in both. The formulas MOVE_2 and MOVE'_2 check that configurations are chosen correctly; these formulas are enabled only at the proper times during play. Let

$$\text{I-WIN}' = (\text{II-ILL} \vee (a'_{2m+2} \wedge b'_{2m+1} \wedge \sim \text{MOVE}'_2(X_c, Y_c))),$$

$$\text{II-WIN}' = (\text{I-ILL} \vee (a'_{m+1} \wedge b'_{m+1} \wedge \sim \text{MOVE}_2(Y_c, X_c))).$$

These two formulas can be transformed to equivalent formulas I-WIN and II-WIN in 12DNF with size(I-WIN) = size(II-WIN) = $O_M(n)$. For example, the terms $(a'_i \wedge a'_{i+1})$ and $(a'_i \wedge b'_{i+1} \wedge \sim b'_{i-1})$ of I-ILL are formulas which involve at most twelve variables (after the abbreviations a'_i and b'_i have been replaced by their definitions). Each term is, therefore, equivalent to a formula in 12DNF, so I-ILL is in 12DNF. Also, $\sim\text{MOVE}_2$

and $\sim\text{MOVE}'_2$ are equivalent to formulas in 3DNF by DeMorgan's laws. Let α_2 be an $(X_2 \cup Y_2)$-assignment such that the assignment to Y_c represents the initial configuration on input w, and $a_i = b_i = 0$ for all i.

The verification of (3.3) and (3.4) is a combination of (3.5) (and its symmetric version) with the fact that legitimate play mimics G_M. Say that M accepts w. Player I's strategy is to play legitimately and play a "side game" of G_M to determine, each time play progresses from $i = 1$ to $i = m$, which configuration to represent by X_c. If II plays legitimately, then just as in part 1, eventually the game will reach a position which satisfies $a'_{2m+2} = b'_{2m+1} = 1$ and $\text{MOVE}'_2(X_c, Y_c) = 0$. Therefore, we need only consider the case that II makes an illegitimate move. If II's first illegitimate move is a pass, then by (3.5) (c), player I can next move to a position (2, I-WIN, II-WIN, α) which satisfies II-ILL = 1 and I-ILL = 0. Since I has been playing legitimately and II passed, we also have that a'_{m+1} and b'_{m+1} are not both 1 under α, so the position satisfies I-WIN = 1 and II-WIN = 0. Say then that II's first illegitimate move is a nonpassing move from position π to π'. By (3.5) (b), the position π' satisfies II-ILL = 1 (and, therefore, I-WIN = 1) and I-ILL = 0. We must check that π' satisfies

$$(a'_{m+1} \wedge b'_{m+1} \wedge \sim\text{MOVE}_2(Y_c, X_c)) = 0.$$

If π satisfies $a'_{m+1} = 0$ we are done. If π satisfies $a'_{m+1} = 1$ then, since both players have been playing legitimately up to π, π satisfies $b'_{m+1} = 0$. Since I has chosen the current assignment to X_c using a winning strategy for G_M, π satisfies $\sim\text{MOVE}_2 = 0$. Since the formula b'_{m+1} contains no variable in Y_c, we conclude that either π' satisfies $b'_{m+1} = 0$ or π' satisfies $\sim\text{MOVE}_2(Y_c, X_c) = 0$. This completes the verification of (3.3). The verification of (3.4) is completely analogous, using the symmetric version of (3.5) (i.e., interchanging I and II), and is left to the reader.

Having noted above that the sizes of I-WIN and II-WIN are $O_M(n)$, it follows that $\mathscr{E}$-TIME $\leqq_{\log} W(G_2)$ via length order $n \log n$.

3. Define

$$\text{I-LOSE}' = (\text{I-ILL} \vee (a'_{m+1} \wedge \sim\text{MOVE}_2(Y_c, X_c))),$$

$$\text{II-LOSE}' = (\text{II-ILL} \vee (b'_{2m+2} \wedge \sim\text{MOVE}'_2(X_c, Y_c))).$$

As in part 2, these formulas are equivalent to formulas I-LOSE and II-LOSE in 12DNF of size $O_M(n)$. It is easy to see that M accepts w iff $(1, \text{I-LOSE}, \text{II-LOSE}, \alpha_2) \in W(G_3)$. First recall that, by the rules of G_3, neither player can pass and player I (II) cannot move to a position which satisfies I-LOSE = 1 (II-LOSE = 1). This forces both players to play legitimately, so it should be obvious that this starting position has the property claimed.

4. Let $\text{I-WIN}(X_2, Y_2)$ and $\text{II-WIN}(X_2, Y_2)$ be the formulas described in part 2. We construct a formula $F_4(X_4, Y_4)$ where

$$X_4 = X_2 \cup \{x_1, x_2, x_3, x_4, x_5\} \quad \text{and} \quad Y_4 = Y_2 \cup \{y_1, y_2, y_3, y_4, y_5\}.$$

Let

$$F'_4 = ((y_1 \vee \text{I-WIN}) \wedge (x_2 \vee y_3)) \vee (x_4 \wedge x_5 \wedge \sim y_3) \vee ((x_1 \vee \text{II-WIN}) \wedge (y_2 \vee x_3)) \vee (y_4 \wedge y_5 \wedge \sim x_3).$$

Since I-WIN and II-WIN are formulas in 12DNF of size $O_M(n)$, F'_4 is equivalent to a formula F_4 in 13DNF of size $O_M(n)$. Let α_4 be an assignment that assigns X_2 and Y_2 as in part 2 and assigns x_i and y_i to 0 for $1 \leqq i \leqq 5$. We claim that M accepts w iff $(1, F_4, \alpha_4) \in W(G_4)$.

Say that M accepts w. We describe a winning strategy for player I. As long as II plays only variables in Y_2, I plays a side game of G_2 using the strategy of (3.3) to determine his plays in X_2. Before each of his moves, I switches strategy if one of the following two conditions are met:

1) if II has just played one of the y_i, then I switches to one of the strategies 1a)–1d);
2) if I has a play (possibly a pass) which moves the side game of G_2 to a position which satisfies I-WIN = 1 and II-WIN = 0, then I switches to one of 2a) or 2b).

Note that (3.3) ensures that either 1) or 2) will eventually occur. If 1) and 2) are met simultaneously then 1) takes precedence.

In describing the strategies 1a)–1d) we can assume that none of the x_i have been played and that exactly one of the y_i has been played. Moreover, the current position satisfies I-WIN = 0 and II-WIN = 0 since condition 2) was not met before I's most recent previous move.

1a) If II has just set y_1 to 1, then I sets x_2 to 1 and wins.

1b) If II has just set y_2 to 1, then I sets x_1 to 1 and wins.

1c) If II has just set y_3 to 1, then I views this as a pass by II in the side game; by (3.3) (b), I has a play in X_2 which sets I-WIN = 1, and I wins by making this play.

1d) If II has just set either y_4 or y_5 to 1, then I sets x_3 to 1. Since I has been playing the strategy (3.3) up to this point, II cannot reach a position which satisfies II-WIN = 1 on his next move. Since also y_1, x_2, y_3 and $\sim x_3$ are 0, II cannot set F_4 to 1 in one move. Therefore, player I can set x_1 to 1 on his next move and win.

In describing the strategies 2a) and 2b) we can assume that none of the x_i or y_i have yet been played.

2a) If the current position satisfies I-WIN = 1, then I sets x_2 to 1 and wins.

2b) If the current position satisfies I-WIN = II-WIN = 0, but I can reach a position which satisfies I-WIN = 1 on his next move, then I sets x_4 to 1. Since the variables x_i and y_i for $1 \leqq i \leqq 5$ other than x_4 are all 0, it is easy to check that II cannot reach a position which satisfies $F_4 = 1$ in one move. Now if II does not set y_3 to 1, then I sets x_5 to 1 and wins. If II does set y_3 to 1, then I makes the play in X_2 that sets I-WIN = 1.

This completes the proof that if M accepts w then $(1, F_4, \alpha_4) \in W(G_4)$. The proof of the converse is symmetric utilizing the symmetry between (3.3) and (3.4) and the symmetry in the definition of F_4'.

5. Let the formulas I-WIN and II-WIN be as in part 2. Let $Y_2' = \{y' \mid y \in Y_2\}$. We describe a formula $F_5(X_5, Y_5)$ where $X_5 = X_2 \cup \{x_0\}$ and $Y_5 = Y_2 \cup Y_2' \cup \{y_0, y_1\}$. Let I-WIN$_5$ and II-WIN$_5$ be the formulas I-WIN and II-WIN, respectively, after substituting $(y \oplus y')$ for all occurrences of y for each $y \in Y_2$. Let T denote a formula which is the *exclusive-or* of the variables in $Y_2' \cup \{y_1\}$. In what follows, it is useful to imagine that T is a "variable" and that I-WIN$_5$ and II-WIN$_5$ contain variables in Y_2 just as in part 2. The effect is that in one move player II can either play one $y \in Y_2$ while leaving T fixed, or play T while leaving all $y \in Y_2$ fixed, or *simultaneously* play T and one $y \in Y_2$. Let

$$F_5 = ((T \wedge \text{I-WIN}_5) \vee (\sim T \wedge x_0)) \wedge \sim \text{II-WIN}_5 \wedge (T \vee y_0).$$

Note that size$(F_5) = O_M(n)$. Let α_5 be an assignment which assigns X_2 and Y_2 as in part 2, assigns x_0 to 0, y_0 to 1 and y_1 to 1, and assigns all $y' \in Y_2'$ to 0; note that the "variable" T assumes the value 1 under α_5. We claim that M accepts w iff $(1, F_5, \alpha_5) \in W(G_5)$; furthermore, if M accepts w then I has a winning strategy such that if II passes then I wins immediately on his next move.

Say that M accepts w. As long as II doesn't play T or y_0, I plays a side game of G_2 using the strategy (3.3) to determine his plays in X_2. Eventually the side game will reach a position which satisfies I-WIN = 1 and II-WIN = 0, so this position satisfies $F_5 = 1$ and I wins. If II either passes or plays y_0, then I views this as a pass by II in the side game, and I next moves to a position which satisfies I-WIN = 1 and II-WIN = 0. If II sets T to 0 (possibly in parallel with a move in the side game), then I sets x_0 to 1 and wins.

Say now that M does not accept w. As long as I doesn't play x_0, II plays a side game of G_2 using (3.4) to determine his plays in Y_2. The "variable" T is left fixed at 1 unless II sees that his next play in the side game reaches a position which satisfies II-WIN = 1. In this case, II makes this play while simultaneously setting T to 0. Now I needs at least two moves to reach a position which satisfies $F_5 = 1$ since he must set x_0 to 1 and make some play in X_2 which sets II-WIN to 0. Therefore, before I can win, II can set y_0 to 0 and I cannot win thereafter. If I sets x_0 to 1 before the side game ends in II's favor, then II sets y_0 to 0. Since II has been playing the strategy (3.4) up to this point, I cannot win on his next move, and II sets T to 0 on his next move. If I passes then II passes.

6. Let $F_5(X_5, Y_5)$ be the formula just described in part 5. By invoking Lemma 3.6 with $S = X_5 \cup Y_5$ and $F = F_5$, there is a formula $H(X_5, Y_5, Z)$ in 3CNF with size$(H) = O_M(n)$ such that, for all assignments to X_5 and Y_5,

$$F_5(X_5, Y_5) = 1 \quad \text{iff} \quad (\exists Z)[H(X_5, Y_5, Z) = 1]. \tag{3.6}$$

Say that $Z = \{z_1, \cdots, z_k\}$. Let H' denote the formula H after substituting $(z_i \vee z_i')$ for z_i for $1 \leqq i \leqq k$. Let

$$X_6 = X_5 \cup \{z_i, z_i' \mid 1 \leqq i \leqq k+1\}, \qquad Y_6 = Y_5 \cup \{u_i, u_i' \mid 1 \leqq i \leqq k\},$$

and

$$F_6' = H' \vee (z_1 \wedge \cdots \wedge z_k \wedge z_{k+1} \wedge (u_1 \vee \cdots \vee u_k)) \vee (z_1' \wedge \cdots \wedge z_k' \wedge z_{k+1}' \wedge (u_1' \vee \cdots \vee u_k')).$$

By the distributive laws, F_6' is equivalent to a formula F_6 in CNF with size$(F_6) = O_M(n^3)$. Let α_6 be an assignment that assigns X_5 and Y_5 as in part 5, assigns z_i and z_i' to 0 for all i and assigns u_i and u_i' to 1 for all i. We show that M accepts w iff $(1, F_6, \alpha_6) \in W(G_6)$.

Say that M accepts w. As long as II plays only variables in Y_5, I determines his plays in X_5 by playing a side game of G_5 starting on position $(1, F_5, \alpha_5)$ using the strategy described in part 5. At some point, the side game will reach a position $\pi = (1, F_5, \alpha)$ such that I can move to a position $\pi' = (2, F_5, \alpha')$ where π' satisfies $F_5 = 1$. At the point where such a π occurs, I begins a strategy we call the *end strategy*. By (3.6), there is a Z-assignment ζ such that H assumes the value 1 under the combined assignments α' and ζ. Let $\mathscr{Z} = \{i \mid \zeta(z_i) = 1\}$. To play the end strategy, I does not immediately make the play in X_5 which moves π to π', but rather I first sets z_i to 1 for all $i \in \mathscr{Z}$ on his next card($\mathscr{Z}$) moves. Each time I sets some z_i to 1, II must respond by setting some u_i to 0; for otherwise I can set all the z_i to 1 before II can set all the u_i to 0, and I wins G_6. (We are still assuming that II played only variables in Y_5 up to the point where the side game reached π.) After I has set z_i to 1 for all $i \in \mathscr{Z}$ and II has responded by setting some u_i to 0, I makes the play in X_5 which moves the side game from π to π'; the new position satisfies $H = 1$ and I wins. If II passes (before playing some u_i or u_i') then, as was noted in part 5, the side game is at a position π as above (i.e., I can win G_5 on his next move). Now I plays the end strategy.

If II plays some u_i' before I begins the end strategy, then I views this as a pass by II in the side game, and I plays the end strategy. If II plays some u_i, then again I views this as a pass and plays the end strategy except that the variables z_i' for $i \in \mathscr{Z}$ are set to 1.

Say now that M does not accept w. Player II plays a side game of G_5 to determine his responses to plays of I in X_5. If I plays some z_i (z_i'), then II changes some u_i (u_i') from 1 to 0 if possible, or II passes otherwise. If I passes then II passes.

This completes the proof of Lemma 3.2.

4. Games on graphs. In the previous section we have exhibited several games on propositional formulas which are log-complete in $\mathscr{E}$-TIME. It is possible that these games will be useful as starting points for reductions to other games, in the same way that the quantified Boolean formula "game" [21], [22] has been used to show that $\mathscr{P}$-SPACE is reducible to certain games [8], [10], [18], [20]. Since $\leqq_{\log}$ is a transitive relation, to show that $\mathscr{E}$-TIME $\leqq_{\log} W(G)$ for some game G, it suffices to show that $EW(G_k) \leqq_{\log} W(G)$ where G_k is one of the formula games. The main purpose of this section is to illustrate, for a particular game G on graphs, how a reduction $EW(G_3) \leqq_{\log} W(G)$ can be performed.

Before presenting this example in detail, we remark that by combining Theorem 3.1 with Schaefer's notion of a *pseudoformula* [19] it is easy to devise $\mathscr{E}$-TIME-complete games which are based on known NP-complete problems. For example, the following game HAM is obtained by combining G_5 with the NP-complete Hamiltonian circuit problem for undirected graphs [16].

HAM: A position in HAM is a tuple $(\tau, V, E, E_1, E_2, \alpha_1, \alpha_2)$ where $\tau \in \{1, 2\}$, V is a finite set (the vertices of the graph), $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ (the edges of the graph), $E_1, E_2 \subseteq E$, $E_1 \cap E_2 = \varnothing$, and $\alpha_i: E_i \to \{\text{in, out}\}$ for $i = 1, 2$. Player I (II) moves by either passing or changing the status of one edge in E_1 (E_2) from "in" to "out" or vice versa. Player I wins if, after some move of either player, there is a Hamiltonian circuit in the graph (V, E) (that is, a circuit which contains each vertex exactly once) such that all of the edges currently declared "in" belong to the circuit and none of the edges currently declared "out" belong to the circuit.

For any of the standard methods of encoding graphs as strings of symbols, EW(HAM) is log-complete in $\mathscr{E}$-TIME. By [19, Fact 3.1] it is immediate that $EW(G_5) \leqq_{\log} EW$(HAM), so Theorem 3.1 and the transitivity of $\leqq_{\log}$ imply that $\mathscr{E}$-TIME $\leqq_{\log} EW$(HAM). Also, EW(HAM) $\in \mathscr{E}$-TIME by the method of Lemma 3.1.

We now describe another game, BLOCK, for which the reduction from $W(G_k)$ to W(BLOCK) is more involved. The portion of a position of BLOCK which remains fixed throughout play of the game is referred to as a *board.*

A *board* is a tuple (V, E, ν, W_1, W_2) where:
V is a finite set;
$E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$;
$\nu: E \to \{1, 2, 3\}$; and
$W_1, W_2 \subseteq V$.

V and E should be thought of as the vertices and edges of an undirected graph.

A position of BLOCK is a tuple (τ, B, M_1, M_2) where:
$\tau \in \{1, 2\}$ signifies whose turn it is;
$B = (V, E, \nu, W_1, W_2)$ is a board; and
$M_1, M_2 \subseteq V$ and $M_1 \cap M_2 = \varnothing$.

M_1 and M_2 represent the movable part of a position; M_1 (M_2) should be thought of as a set of markers which belong to player I (II) and are placed on vertices of the board. A player moves by choosing one of his markers and moving it to a new unoccupied vertex by traversing edges of the graph subject to the restrictions that (i) all traversed edges are given the same value by ν, and (ii) no traversed vertex is occupied by a marker of either player. The players are not permitted to pass. The function ν gives a "direction" to each edge; the restriction (i) corresponds, for example, to the situation in Chess that a queen can move in any of four directions but cannot change direction during a single move. Player I (II) wins by placing one of his markers on a vertex in W_1 (W_2). Formally, $(1, B, M_1, M_2)$ can reach $(2, B, M_1', M_2')$ in one move iff $M_2 = M_2'$, $M_2 \cap W_2 = \varnothing$, and there exist $u_1, \cdots, u_k \in V$ and $c \in \{1, 2, 3\}$ such that $u_1 \neq u_k$, $u_1 \in M_1$, $M_1' = (M_1 - \{u_1\}) \cup \{u_k\}$, $u_i \notin M_1 \cup M_2$ for $2 \leqq i \leqq k$, and $\{u_i, u_{i+1}\} \in E$ and $\nu(\{u_i, u_{i+1}\}) = c$ for $1 \leqq i < k$. The legal moves of player II are defined symmetrically.

To encode positions as words over a finite alphabet, associate each element $u \in V$ with a distinct binary word $\omega(u)$ of length roughly $\log(\text{card}(V))$, and list the various elements of a position in some natural way; for example, letting $e = \text{card}(E)$, list $E = \{\{u_1, v_1\}, \cdots, \{u_e, v_e\}\}$ as $\omega(u_1)\$\omega(v_1)\$ \cdots \$\omega(u_e)\$\omega(v_e)$. Let $EW(\text{BLOCK})$ denote the set of encodings of positions in $W(\text{BLOCK})$.

THEOREM 4.1. *$EW(\text{BLOCK})$ is log-complete in $\mathscr{E}$-TIME.*

Proof. By the algorithm of Lemma 3.1 and the method of encoding positions, one finds that there is a constant d such that

$$EW(\text{BLOCK}) \in \text{DTIME}(d^{n/\log n}).$$

To show that $\mathscr{E}\text{-TIME} \leqq_{\log} EW(\text{BLOCK})$ it suffices to prove

$$EW(G_3) \leqq_{\log} EW(\text{BLOCK}). \tag{4.1}$$

Let

$$\pi = (\tau, \text{I-LOSE}(X, Y), \text{II-LOSE}(X, Y), \alpha)$$

be a given position of G_3. We describe a position

$$\pi' = (\tau, (V, E, \nu, W_1, W_2), M_1, M_2)$$

in BLOCK such that $\pi \in W(G_3)$ iff $\pi' \in W(\text{BLOCK})$.

Say that $X = \{x_i \mid 1 \leqq i \leqq m_1\}$ and $Y = \{y_i \mid 1 \leqq i \leqq m_2\}$. Let $\bar{x}_i$ denote the literal $\sim x_i$ and let $\bar{y}_i$ denote $\sim y_i$. For each variable x_i and y_i the graph (V, E) contains the subgraph depicted in Figs. 4(a) and 4(b), respectively. In these figures, vertices are labeled by subscripted lower case Roman letters. The value of each edge under ν is indicated by drawing the edge as either solid, dashed, or dotted. Vertices which belong to W_1 (W_2) are indicated by open (solid) stars. The markers of player I (II) are shown as open (solid) circles. The dashed edges in these figures connect these subgraphs to the remainder of the graph, and these are the only such connections. The rest of the graph is constructed in such a way that (i) neither y_i nor $\bar{y}_i$ (resp., x_i nor $\bar{x}_i$) is connected to a vertex in W_2 (resp., W_1) by a path of dashed edges, and (ii) any attempt by either player to move a marker from outside one of these subgraphs to a vertex x_i, $\bar{x}_i$, y_i, or $\bar{y}_i$ results in an immediate win on the next move for the other player.

Consider Fig. 4(a). The marker currently shown on vertex x_i is termed the *i*th *value marker* (*of player* I) and is free to move between x_i and $\bar{x}_i$. The position of this marker is associated with a truth value of the variable x_i as follows: if the marker is on x_i ($\bar{x}_i$), then x_i has value 0 (1). The markers on d_{1i} and c_{1i} are termed *guard markers*. If I moves his *i*th value marker to a vertex other than x_i or $\bar{x}_i$, then II wins on the next move by moving

his guard marker to either a_{1i} or b_{1i}, one of which must be unoccupied. However, if II moves his guard to either $\bar{x}_i$, e_{1i}, or f_{1i}, then I wins immediately by moving his guard marker from c_{1i} to either e_{1i} or f_{1i}. The terminology and behavior associated with Fig. 4(b) is analogous. Therefore, at each move player I (II) must either move his ith value marker from x_i to $\bar{x}_i$ or vice versa (resp., from y_i to $\bar{y}_i$ or vice versa) for some i, or move some marker other than a value marker or a guard marker.

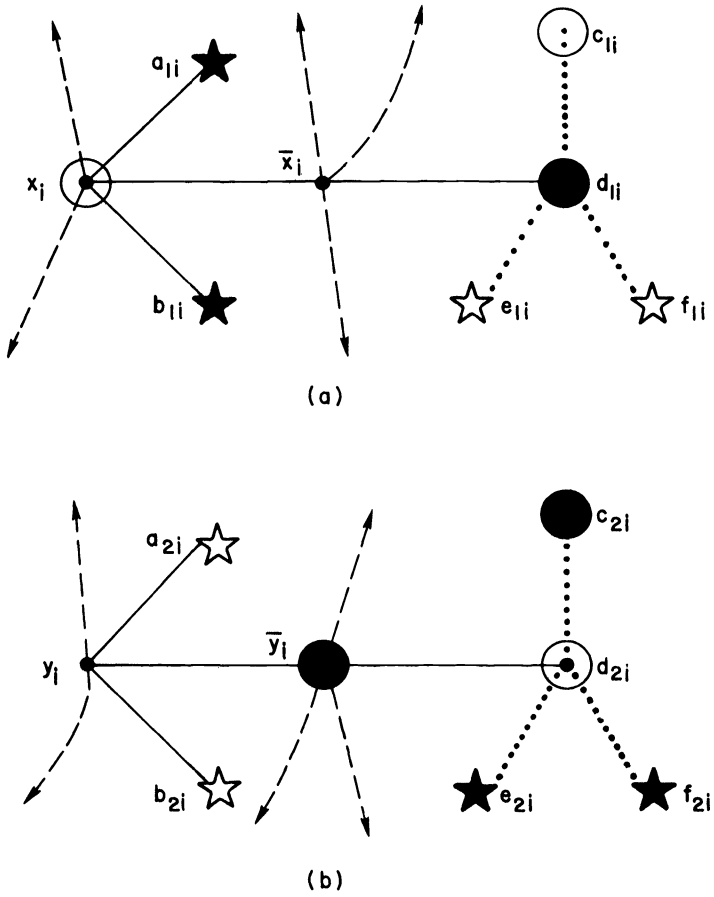


FIG. 4. *Subgraphs which represent the truth value of x_i and y_i.*

Write I-LOSE as $C_{11} \vee \cdots \vee C_{1k_1}$ and write II-LOSE as $C_{21} \vee \cdots \vee C_{2k_2}$ where each C_{ij} is a conjunction of literals. Another portion of the graph (V, E) is constructed for each C_{ij}. For example, say that C_{2j} is $(x_3 \wedge \bar{y}_5)$. Then the graph would contain the subgraph shown in Fig. 5. (With the exception of x_3 and $\bar{y}_5$, each vertex label in Fig. 5 actually has two additional subscripts, 2 and j, which have been repressed for readability.) This subgraph has two relevant properties. First, if it is player I's turn to move and both x_3 and $\bar{y}_5$ are unoccupied (corresponding to an assignment for which $x_3 \wedge \bar{y}_5$ is *true*) then I has a forced win. The strategy which achieves the win is termed the *end strategy*. To play the end strategy, I first moves the marker from w to s_1. Now II is forced to place a marker on r_1, and the marker shown currently on v is the only one that can reach r_1 in one move. Now I is forced to place a marker on u_1, and he does this by moving the marker from s_1 to u_1. Now II is forced to move his marker from r_1 to t_1, I is

forced to move from u_1 to s_2, and so on. Finally, I is able to move a marker from u_2 to z, and I wins. Note that as soon as I moves the marker from w to s_1, the moves of both players are forced. This yields the second relevant property of this subgraph: if I moves the marker from w to s_1 at a time when either x_3 or $\bar{y}_5$ is occupied, then II has a forced win. For example, suppose that x_3 is occupied and I moves from w to s_1. Then II moves from v to r_1. The only marker of player I which can reach u_1 in one move is the one currently on x_3. However, if I moves this marker from x_3 to u_1, then II moves his guard marker from d_{13} to a_{13} and wins (see Fig. 4(a) where $i = 3$).

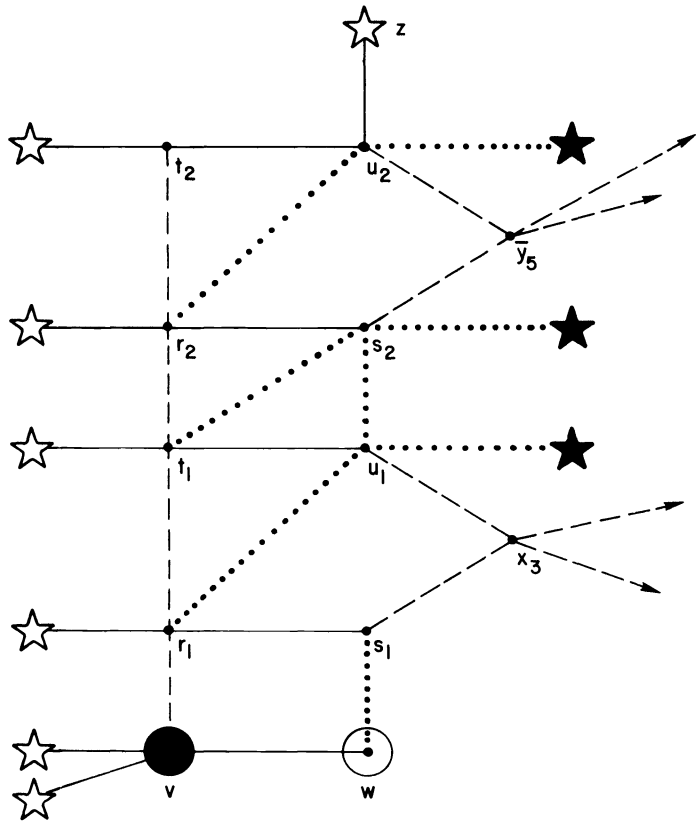


FIG. 5. *The subgraph which corresponds to the clause* $(x_3 \wedge \bar{y}_5)$ *in* II-LOSE.

It should be obvious how to generalize the graph of Fig. 5 for conjunctions with more than two literals. For conjunctions C_{1j} in I-LOSE, the construction is similar except that the markers on vertices v and w are interchanged and the vertices in W_1 (W_2) become vertices in W_2 (W_1). The position π' consists of the subgraphs of Fig. 4(a) and 4(b) for each variable together with the subgraph of Fig. 5 for each conjunction in I-LOSE and II-LOSE. The initial placement of value markers is specified by the assignment α. Observe that there is a constant b such that

$$\text{card}(V)+\text{card}(E) \leqq b \cdot (\text{size(I-LOSE)}+\text{size(II-LOSE)}). \tag{4.2}$$

It is not difficult to see that $\pi \in W(G_3)$ iff $\pi' \in W(\text{BLOCK})$. Suppose that $\pi \in W(G_3)$. Player I plays a side game of G_3 starting on π to determine his movement of value markers as long as II moves only value markers. At some point, player II must

move the side game to a position (1, I-LOSE, II-LOSE, β) where II-LOSE = 1 under β, so $C_{2j} = 1$ for some j. Now I can successfully play the end strategy on the subgraph corresponding to C_{2j}. The only other possibility is that II moves a marker from w_{1j} to s_{11j} (cf. Fig. 5), for some j, before II-LOSE = 1 in the side game. But since I has been playing a winning strategy in G_3, some literal in C_{1j} is 0 (i.e., the vertex corresponding to that literal is covered by a marker). Now II's moves are forced, and I wins as discussed above. The argument that $\pi \notin W(G_3)$ implies $\pi' \notin W(\text{BLOCK})$ is symmetric. This completes the proof of (4.1) and, therefore, that of Theorem 4.1. □

Moreover, from the proofs of Lemma 3.2 and (4.1), the inequality (4.2), and the method of encoding positions of BLOCK, it can be seen that the following is true.

COROLLARY 4.1. 1) $\mathscr{E}$-TIME $\leqq_{\log} EW(\text{BLOCK})$ *via length order* $n \log n$.

2) *There is a constant* $c > 0$, *such that if a* DTM *accepts* $EW(\text{BLOCK})$ *within time* $T(n)$, *then* $T(n) > c^{n/\log n}$ *for infinitely many* n.

REFERENCES

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

[2] M. BAUER, D. BRAND, M. FISCHER, A. MEYER AND M. PATERSON, *A note on disjunctive form tautologies*, SIGACT News, 5 (April 1973), pp. 17–20.

[3] C. L. BOUTON, *Nim, a game with a complete mathematical theory*, Ann. Math. Princeton, 3 (1902), pp. 35–39.

[4] A. K. CHANDRA AND L. J. STOCKMEYER, *Alternation*, Proc. 17th IEEE Symp. on Foundations of Computer Science, 1976, IEEE, New York, 1976, pp. 98–108.

[5] J. H. CONWAY, *On Numbers and Games*, Academic Press, New York, 1976.

[6] S. A. COOK, *The complexity of theorem proving procedures*, Proc. Third ACM Symp. on Theory of Computing, 1971, Assoc. for Comput. Mach., New York, 1971, pp. 151–158.

[7] S. A. COOK AND R. A. RECKHOW, *Time bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[8] S. EVEN AND R. E. TARJAN, *A combinatorial problem which is complete in polynomial space*, J. Assoc. Comput. Mach., 23 (1976), pp. 710–719.

[9] M. J. FISCHER AND R. E. LADNER, *Propositional modal logic of programs*, Proc. Ninth ACM Symp. on Theory of Computing, 1977, Assoc. for Comput. Mach., New York, 1977, pp. 286–294.

[10] A. S. FRAENKEL, M. R. GAREY, D. S. JOHNSON, T. SCHAEFER AND Y. YESHA, *The complexity of Checkers on an* $N \times N$ *board–preliminary report*, Proc. 19th IEEE Symp. on Foundations of Computer Science, 1978, IEEE, New York, pp. 55–64.

[11] R. K. GUY AND C. A. B. SMITH, *The G-values of various games*, Proc. Cambridge Philos. Soc., 52 (1956), pp. 514–526.

[12] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[13] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[14] N. D. JONES, *Space-bounded reducibility among combinatorial problems*, J. Comput. System Sci., 11 (1975), pp. 68–85.

[15] N. D. JONES AND W. T. LAASER, *Complete problems for deterministic polynomial time*, Theoretical Computer Science, 3 (1977), pp. 105–117.

[16] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[17] D. KOZEN, *On parallelism in Turing machines*, Proc. 17th IEEE Symp. on Foundations of Computer Science, 1976, IEEE, New York, 1976, pp. 89–97.

[18] D. LICHTENSTEIN AND M. SIPSER, *Go is Pspace hard*, Proc. 19th IEEE Symp. on Foundations of Computer Science. 1978, IEEE, New York, pp. 48–54.

[19] T. J. SCHAEFER, *Complexity of decision problems based on finite two-person perfect-information games*, Proc. Eighth ACM Symp. on Theory of Computing, 1976, Assoc. for Comput. Mach., New York, 1976, pp. 41–49.

[20] ———, *On the complexity of some two-person perfect information games*, J. Comput. System Sci., 16 (1978), pp. 185–225.

[21] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoretical Computer Science, 3 (1977), pp. 1–22.
[22] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time: preliminary report*, Proc. Fifth ACM Symp. on Theory of Computing, 1973, Assoc. for Comput. Mach., New York, 1973, pp. 1–9.
[23] G. S. TSEITIN, *On the complexity of derivation in propositional calculus*, Studies in Constructive Mathematics and Mathematical Logic, Part II, A. O. Slisenko, ed., Steklov Math. Institute, Leningrad, 1968.
[24] J. VON NEUMANN AND O. MORGENSTERN, *Theory of Games and Economic Behavior*, 3rd ed., Princeton University Press, Princeton, NJ, 1953.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0006$01.00/0

DYNAMIC BINARY SEARCH*

KURT MEHLHORN†

Abstract. We consider search trees under time-varying access probabilities. Let $S=\{B_1, \cdots, B_n\}$ and let p_i^t be the number of accesses to object B_i up to time t, $W^t=\sum p_i^t$. We introduce D-trees with the following properties.

1) A search for $X=B_i$ at time t takes time $O(\log W^t/p_i^t)$. This is nearly optimal.

2) Update time after a search is at most proportional to search time, i.e. the overhead for administration is small.

Key words. searching, binary trees, time-varying access probabilities, TRIES

1. Introduction. "One of the popular methods for retrieving information by its 'name' is to store the names in a binary tree. We are given n names $B_1, B_2, \cdots, B_n$ and $2n+1$ frequencies $\beta_1, \cdots, \beta_n, \alpha_0, \cdots, \alpha_n$ with $\sum \beta_i + \sum \alpha_j = 1$. Here β_j is the frequency of encountering name B_i and α_j is the frequency of encountering a name which lies between B_j and B_{j+1}, α_0 and α_n have obvious interpretations." [13].

A binary search tree T is a tree with n interior nodes (nodes having two sons), which we denote by circles, and $n+1$ leaves, which we denote by squares. The interior nodes are labeled by the B_i in increasing order from left to right and the leaves are labeled by the intervals (B_j, B_{j+1}) in increasing order from left to right. Let b_i be the distance of interior node B_i from the root and let a_i be the distance of leaf (B_j, B_{j+1}) from the root. To retrieve a name X, b_i+1 comparisons are needed if $X=B_i$ and a_i comparisons are required if $B_j<X<B_{j+1}$. Therefore we define the weighted path length of tree T as

$$P=\sum_{i=1}^{n} \beta_i(b_i+1)+\sum_{j=0}^{n} \alpha_j a_j.$$

A large number of papers have been written on the subject of constructing optimal or nearly optimal binary search trees [3], [7], [8], [9], [10], [11], [12], [13], [15], [16], [17], [21]. We quote two results:

It is possible to construct a tree T, in time linear in the number of nodes, such that

$$b_i \leqq \log 1/\beta_i$$

and

$$a_j \leqq \log 1/\alpha_j + 2$$

[7], [17], [18]. Furthermore these bounds are almost sharp for most nodes and leaves [10], [18]. More precisely, for any $d \in \mathbb{R}$ and $h>0$ let

$$L_h=\{j;\ a_j \leqq (\log(1/\alpha_j)-h)/\log(2+2^{-d})\}$$

and

$$N_h=\{j;\ b_i+1 \leqq (\log(1/\beta_i)-h-d)/\log(2+2^{-d})\}.$$

* Received by the editors January 24, 1978.

† Angewandte Mathematik und Informatik der Universität Saarlandes, D-6600 Saarbrücken, Germany.

Then

$$\sum_{j \in L_c} \alpha_j + \sum_{i \in N_c} \beta_i \leqq 2^{-h};$$

i.e. only a small percentage of the nodes can be considerably higher in the tree than stated in the upper bound. These results show that the best we can expect from binary search trees is "logarithmic" behavior.

In many applications the access frequencies are (a) not known in advance (b) changing over time and therefore (nearly) optimal binary trees are not readily applicable. In this paper we introduce D-trees (dynamic-trees) in an attempt to resolve this difficulty and thus answer a challenge of Knuth [13]: "A harder problem, but perhaps solvable, is to devise an algorithm which keeps its frequency counts empirically, maintaining the tree in optimum form depending on the past history of the searches. Names occurring most frequently gradually move towards the root, etc.".

We suggest the following model. With every node B_i and leaf (B_j, B_{j+1}) we associate its frequency count p_i^t and q_j^t respectively. Here p_i^t is the number of searches for $X = B_i$ performed up to time t and q_j^t is the number of searches performed for $X \in (B_j, B_{j+1})$ performed up to time t. We use $W^t = \sum p_i^t + \sum q_j^t$ for the total number of accesses up to time t. Then $\beta_i^t = p_i^t / W^t$ ($\alpha_j^t = q_j^t / W^t$) is the relative access frequency of node B_i (of leaf (B_j, B_{j+1})) at time t. A search for $X \in B_i$ ($X \in (B_j, B_{j+1})$) at time t increases p_i^t (q_j^t) by one. We drop the upper index t when it is clear from the context. Our tree structure exhibits the following behavior:

1. The tree is always nearly optimal, i.e. a search for $X = B_i$ ($X \in (B_j, B_{j+1})$) can be carried out in time $O(\log 1/\beta_i^t)$ ($O(\log 1/\alpha_j^t)$).

2. The time needed to update the tree structure is at most proportional to search time. This is achieved by restricting updating to the path from the root to the node (leaf) searched for.

3. New names can be inserted in time $O(\min(n, \log W))$.

In § 2 we review some facts about weight-balanced trees [20]. In § 3 we introduce D-trees and show properties 1 and 2 above. In § 4 we give an alternate definition and work out some of its properties. Section 5 is dedicated to compact D-trees, and in § 6 we give some extensions. Finally, we apply D-trees to TRIES.

Note. A preliminary version of this paper was presented at the 4th Colloquium on Automata, Languages and Programming, Turku, 1977, Springer-Verlag Lecture Notes vol. 52, pp. 323–336, Springer-Verlag, Berlin.

2. Preliminaries: Weight balanced trees. Nievergelt and Reingold introduced weight balanced trees (cf. [20] and [18]). We review some of their definitions and adapt them for our purposes. In a binary tree every node has either two sons or no son at all. Nodes with no sons are called leaves.

DEFINITION. Let T be a binary tree. If T is a single leaf then the *root-balance* $\rho(T)$ is $1/2$; otherwise we define $\rho(T) = |T_l|/|T|$, where $|T_l|$ is the number of leaves in the left subtree of T and $|T|$ is the number of leaves in tree T.

DEFINITION. A binary tree T is said to be of *bounded balance* α, or in the set BB[α], for $0 \leqq \alpha \leqq 1/2$, if and only if

1. $\alpha \leqq \rho(T) \leqq 1 - \alpha$,

2. T is a single leaf or both subtrees are of bounded balance α.

The depth of a tree T of bounded balance α is $O(\log |T|)$. We add a leaf to a tree T by replacing a leaf by a tree consisting of one node and two leaves. "If upon the addition of a leaf to a tree in BB[α] the tree becomes unbalanced relative to α, that is, some subtree of T has root-balance outside the range $[\alpha, 1-\alpha]$ then that subtree can be

rebalanced by a rotation or a double rotation. In Fig. 1 we have used squares to represent nodes, and triangles to represent subtrees; the root-balance is given beside each node"[20]. Symmetrical variants of the operations exist.

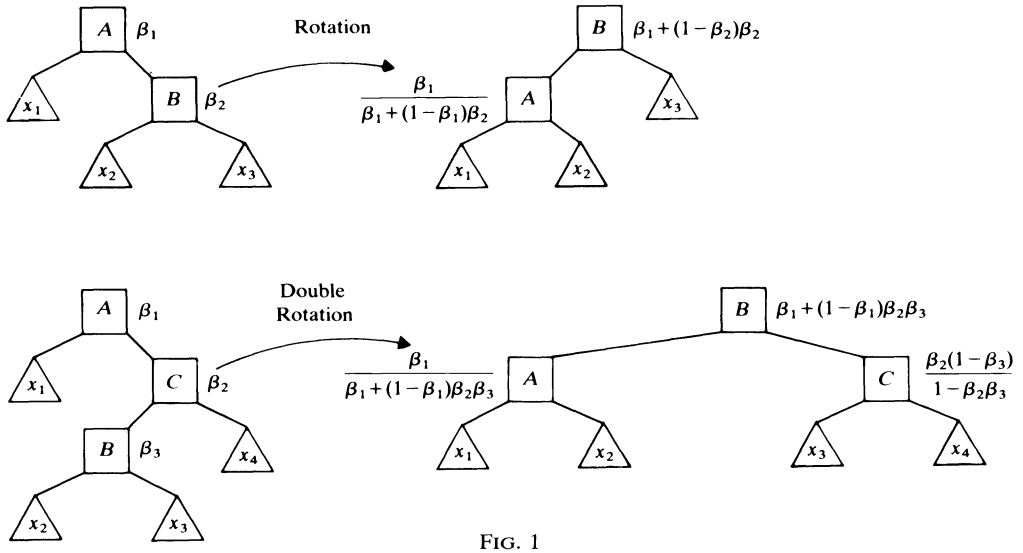


FIG. 1

Fact 1 (Nievergelt and Reingold). If $\alpha \leqq 1-\sqrt{2}/2$ and the insertion of a leaf in a tree in BB[α] causes a subtree T of that tree to have root-balance less than α, T can be rebalanced by performing one of the two transformations shown above. More precisely let β_2 denote the balance of the right subtree of T after the insertion has been done. If $\beta_2 < (1-2\alpha)/(1-\alpha)$ then a rotation will rebalance T, otherwise a double rotation will rebalance T.

The search time in weight-balanced trees is proportional to the logarithm of the number of leaves. Updating the structure upon insertion (or deletion) of a leaf can be done in time proportional to the search time. It takes constant time on the average [24]. In the next section we adapt weight-balanced trees to binary search trees.

3. D-Trees: The basic scheme. In this section we restrict the discussion to the case in which only searches for the leaves of a binary search tree are performed. Let q_j be the number of searches for some $X \in (B_j, B_{j+1})$, $0 \leqq j \leqq n$, performed up to now and let $W = \sum q_j$ be the total number of searches performed so far. We assume $q_0 = q_n = 1$ to avoid some technical difficulties. This can always be achieved by adding two extra names. From now on α is fixed, $0 < \alpha \leqq 1-\sqrt{2}/2$.

Let T be a tree in BB[α] with W leaves. The leaves of T are labeled from left to right according to the following rule. The first q_0 leaves are labeled with $(\cdot\,, B_1)$, the next q_1 leaves are labeled with $(B_1, B_2), \cdots$. The idea of duplicating leaves appears implicitly in [17] and explicitly in [15].

DEFINITION. (a) A node v of T is a *j-node*, $0 \leqq j \leqq n$, if all leaves in the subtree with root v are labeled with (B_j, B_{j+1}) and v's father does not have this property.

(b) A node v of T is the *j-joint*, if all leaves labeled with (B_j, B_{j+1}) are descendants of v and neither of v's sons has the property.

In general, the j-joint is not a j-node. If it is, then there is just one j-node. Figure 2 shows the relative position of j-nodes and the j-joint.

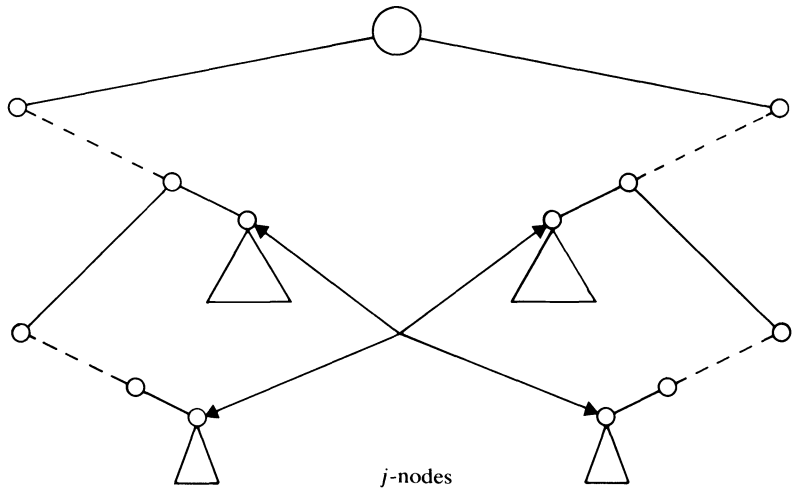


FIG. 2. *Dotted lines denote zero or more tree edges.*

DEFINITION. (a) Consider the j-joint v. q'_j of the leaves labeled with (B_j, B_{j+1}) are left of v and q''_j are right of v. If $q'_j \geqq q''_j$ then the j-node of minimal depth to the left of v is *active*; otherwise the j-node of minimal depth to the right of v is active.

(b) The thickness th (v) of a node is the number of leaves in the subtree with root v.

LEMMA 1. *Let a_j be the depth of the active j-node in tree T. Then $a_j \leqq c_1 \log 1/\alpha_j + c_2$, where $c_1 = 1/\log(1/(1-\alpha))$, $c_2 = 1 + c_1$ and $\alpha_j = q_j/W$.*

Proof. Let v be the active j-node, a_j the depth of v and let w be the father of v. We show th $(w) \geqq q_j/2$. If v is the j-joint then th $(v) = q_j$ and we are done. Otherwise v is a left or right descendant of the j-joint. Suppose v is a left descendant. Then at least $q_j/2$ of the leaves labeled with (B_j, B_{j+1}) are in the left subtree of the j-joint. All of them are descendants of w. Hence th $(w) \geqq q_j/2$. w has depth $a_j - 1$. Since the tree T is of bounded balance α

$$\text{th}(w) \leqq (1-\alpha)^{a_j - 1} \cdot W$$

and hence

$$\alpha_j \leqq 2 \cdot (1-\alpha)^{a_j - 1}.$$

Taking logarithms yields the result. □

Example. Let $\alpha = 1 - \sqrt{2}/2$. Then $c_1 = 2$, $c_3 = 3$ and hence $a_j \leqq 2 \log 1/\alpha_j + 3$. No analogue to Lemma 1 exists if one takes height-balanced trees instead of weight-balanced trees as the underlying tree structure.

Next we have to assign queries to the nodes of tree T. The queries are of the form

"**if** $X < B_j$ **then** go left **else** go right".

We assign queries in such a way as to direct a search for $X \in (B_j, B_{j+1})$ to the active j-node. Then Lemma 1 assures us that search time is logarithmic and thus nearly optimal.

Let v be any node of T. Let j be maximal with: the active j-node is left of v. Then we assign the query "**if** $X < B_{j+1}$ **then** left **else** right" to v. This rule assigns queries to all nodes of v. It is apparent that a search for $X \in (B_j, B_{j+1})$ is directed to the active j-node. Figure 3 is Fig. 2 redrawn; this time the queries are shown.

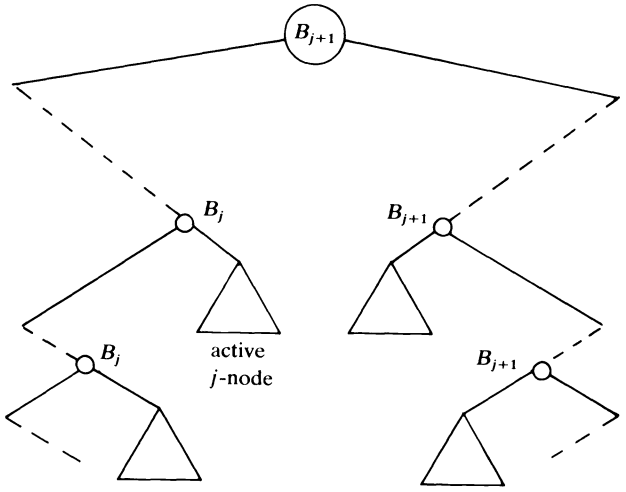
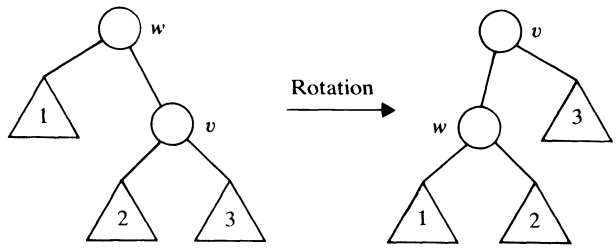


FIG. 3

Before we describe the searching in and updating of our tree structure we have to say more about the information stored in the nodes.

1. All proper descendants of j-nodes are pruned.
2. In each remaining node of the underlying tree of bounded balance α we store
(a) the type of the node: joint node or j-node or neither of above,
(b) its thickness,
(c) in the case of a joint node the number of j-leaves in its left and right subtree,
(d) in the case of a j-node a pointer to the element B_j of a linear list containing the names $B_1, \cdots, B_n$ in that order. An array will do if the insertion of new names is not required (cf. § 6).

Suppose now that we search for some $X \in (B_j, B_{j+1})$. We descend the tree as directed by the queries and end up in the active j-node. As we descend, the thickness of every node encountered during the descent is increased by one. Then we ascend and rebalance the trees as in the case of trees of bounded balance α. Three new problems arise. Assume that we reach node w from its right son v. Then we searched below v. The thickness of w and v were both increased by 1. If the root balance $\rho(w) = (\text{th}(w) - \text{th}(v))/\text{th}(w)$ is less than α then we have to rebalance the tree. We treat the case of a rotation and leave the case of a double rotation for the reader.



Problem 1. v is a j-node for some j (see Fig. 4). Then trees $\triangle_2$ and $\triangle_3$ do not exist explicitly. We recreate them by splitting v into 2 j-nodes of thickness $\lfloor \text{th}(v)/2 \rfloor$ and $\lceil \text{th}(v)/2 \rceil$ respectively. Then $1/3 \leqq \rho(v) = \lfloor \text{th}(v)/2 \rfloor / \text{th}(v) \leqq 1/2$ because of $\text{th}(v) \geqq 2$. A rotation will rebalance the tree. Note further that the j-node v is to the left of the

j-joint (cf. Fig. 2). Hence the query assigned to v after the rotation should be "if $X < B_j \cdots$". The name B_j can be found using the pointer into the linear list containing all names.

Problem 2. w is a j-node after the rotation. Then we have to combine the two trees $\triangle_1$ and $\triangle_2$ into a single node.

Problem 3. Queries have to be changed. This can only happen if the active nodes change. This can only be the case if the active node is split (Problem 1) or combined (Problem 2) or if the distribution of the leaves labeled with (B_j, B_{j+1}) with respect to the j-joint changes. The first two cases were treated already. The distribution of the leaves labeled with (B_j, B_{j+1}) with respect to the j-joint can only change if the j-joint is one of the nodes involved in the transformation, i.e. is either node w or v in Fig. 4. If v is the j-joint then the distribution does not change. If w is the j-joint then we have to distinguish two cases.

Case 1. The tree $\triangle_3$ contains no j-node. Then all j-nodes to the right of w (the j-joint) are elements of the subtree $\triangle_2$. This can be checked by comparing the thickness of the root of $\triangle_2$ with the number stored in the j-joint w. In this case nothing has to be changed.

Case 2. The tree $\triangle_3$ contains a j-node. Then $\triangle_2$ is a j-node and its thickness is strictly smaller than the number stored in w. Then w ceases to be the j-joint, v becomes the j-joint. The query assigned to w after the rotation is "if $X < B_j$ then $\cdots$". The distribution of the j-leaves with respect to the new j-joint v can be computed from the thickness of the j-node $\triangle_2$ and the distribution stored in the old j-joint w. The distribution is stored in v and the appropriate query is assigned to v. Again the names B_j and B_{j+1} can be found using the pointer stored in the j-node $\triangle_2$.

Problem 4. Nodes change their type.

Case 1. A joint node can only change its type if it is involved in the rotation. Hence this case was treated already in Problem 3.

Case 2. A j-node can only change its type if it is involved in the rotation. Hence this case reduces to Problems 1 and 2.

We summarize the discussion. We use trees of bounded balance in order to implement search trees. The depth of the active j-node is always less than $c_1 \log 1/\alpha_j^t + c_2$ for some small constants c_1 and c_2. Here, α_j^t is the relative frequency of leaf (B_j, B_{j+1}) at time t. Updating is restricted to the path from the root to the active j-node. In each node of the path a constant amount of work is necessary. Thus searching and updating the structure can be performed in time $O(\log 1/\alpha_j)$.

THEOREM 1. *Consider a D-tree based on a* BB[α]*-tree with* $0 < \alpha \leqq 1 - \sqrt{2}/2$.

(a) *Let q_j^t be the number of searches for $X \in (B_j, B_{j+1})$, $0 \leqq j < n$, performed up to time t and let $W^t = \sum q_j^t$. Then at time t a search for $X \in (B_j, B_{j+1})$ can be executed in time $O(c_1 \log (W^t/q_j^t) + c_2)$ where $c_1 = 1/\log (1/(1-\alpha))$ and $c_2 = 1 + c_1$. The time needed to update the tree structure is proportional to the search time.*

(b) *Let $\alpha_j^t = q_j^t/W^t$, $H = H(\alpha_0^t, \alpha_1, \cdots, \alpha_n^t) = -\sum \alpha_j^t \log \alpha_j^t$, let P_{opt} be the weighted path length of an optimal search tree for the distribution and let P be the weighted path length of the D-tree at time t. Then*

$$P \leqq c_1 H + c_2$$
$$\leqq c_1 P_{\text{opt}} + c_2$$

where c_1, c_2 are defined as in (a).

(c) *The number of nodes in a D-tree is $O(n(1 + \log W))$.*

Proof. (a) This is immediate from Lemma 1 and the preceding discussion.

(b) Let a_j^t be the depth of the active j-node at time t. Then $a_j^t \leqq c_1 \log 1/\alpha_j^t + c_2$ by Lemma 1. Hence

$$\begin{aligned} P &= \sum a_j^t \alpha_j^t \leqq \sum \alpha_j^t (-c_1 \log 1/\alpha_j^t) + c_2 \\ &\leqq c_1 \cdot H + c_2 \\ &\leqq c_1 \cdot P_{\text{opt}} + c_2 \end{aligned}$$

since $H \leqq P_{\text{opt}}$ by Gilbert and Moore [8], [14], [18].

(c) Suppose there are k j-nodes $v_1, \cdots, v_k$ of thickness $q^{(1)}, \cdots, q^{(k)}$ respectively. k_1 of these j-nodes are to the left of the j-joint. Among these v_1 has maximal depth and v_{k_1} has minimal depth. Let w be the father of v_{k_1}. Then

$$\text{th}\,(v_{k_1})/\text{th}\,(w) \geqq \alpha$$

and hence

$$\text{th}\,(w) \leqq q^{(k_1)}/\alpha \leqq q_j/\alpha.$$

Furthermore $1 \leqq q^{(1)} = \text{th}\,(v_1)$. The depth of node v_1 in the tree with root w is bounded above by $c_1 \log (\text{th}\,(w)/\text{th}\,(v_1)) + c_2$. (Lemma 1). v_1 has depth $\geqq k_1$. Thus

$$\begin{aligned} k_1 &\leqq c_1 \log \text{th}\,(w) + c_2 \\ &\leqq c_1 \log q_j/\alpha + c_2. \end{aligned}$$

From this we conclude that the number of j-nodes is $\leqq d_1 \log q_j + d_2$ for suitable constants d_1 and d_2. Hence the total number of j-nodes, $0 \leqq j \leqq n$, is

$$\begin{aligned} &\leqq d_1 \cdot \sum_{j=0}^{n} \log q_j + d_2(n+1) \\ &\leqq d_1 \cdot n \cdot \log W + d_2(n+1). \end{aligned}$$

Since the j-nodes are the leaves of the D-tree the total number of nodes is $O(n(1 + \log W))$. □

Example. Figure 5 shows a D-tree for the distribution $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 8, 2)$ based on a tree in BB[1/4]. The j-nodes are indicated by square boxes, the active j-nodes are underlined, the thickness of the j-nodes is written on top of them and finally the distribution of the j-leaves with respect to the j-joints is written on top of the joint nodes.

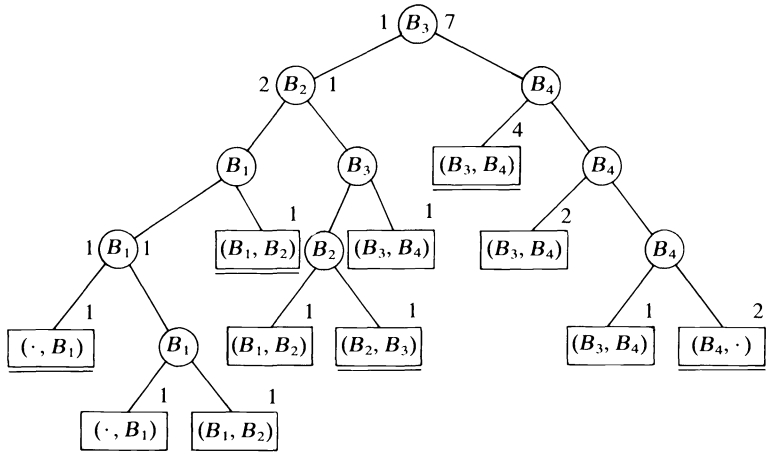


FIG. 5

Suppose we search for $X < B_1$. The search is directed towards the active 0-node and destroys the balance in the node between the 0-joint and the 1-joint. A rotation about that node rebalances the tree. One gets Fig. 6.

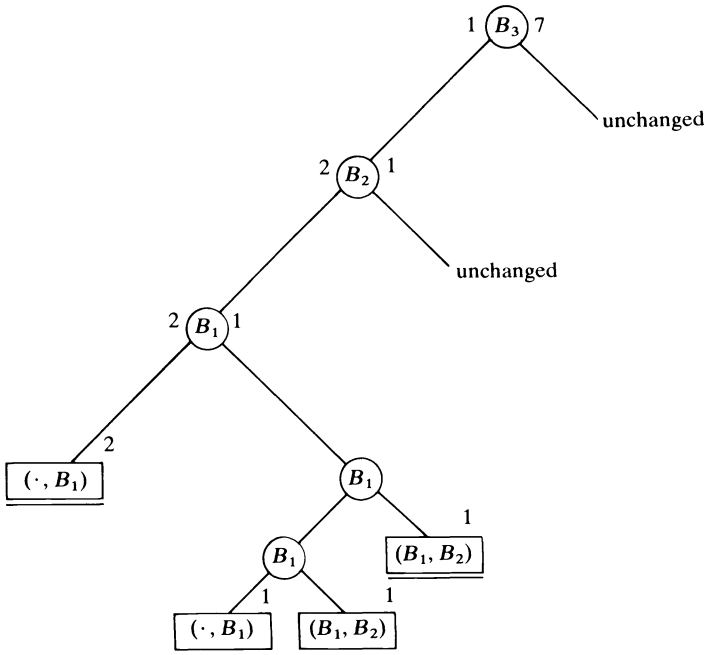


FIG. 6

4. A modified definition. Recall that the active j-node is the minimal depth j-node in that subtree of the j-joint which contains at least one half of the j-leaves. Hence the active j-node is not necessarily a j-node of minimal depth. An example is shown in Fig. 7.

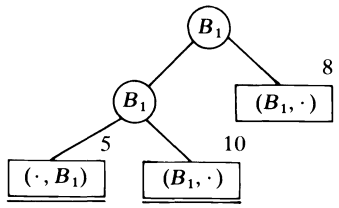


FIG. 7. *A D-tree for* $q_0 = 5$, $q_1 = 18$.

Of course, search time could be improved by ensuring that the active j-node is always a j-node of minimal depth. This leads to the following alternate definition of active j-node.

DEFINITION (alternate definition of active j-node). Exactly one of the j-nodes is active. The active j-node is a j-node of minimal depth.

Lemma 1 and hence Theorem 1(b) is obviously true for the alternate definition of active j-node. However, the bound for the weighted path length can be improved considerably.

THEOREM 2. (Average search time in D-trees with the alternate definition of active j-node).

$$P \leqq (1/H(\alpha, 1-\alpha)) \cdot H(\alpha_0, \cdots, \alpha_n) + c$$

where $c = 1 + 1/\alpha$.

Example. Let $\alpha = 1 - \sqrt{2}/2$. Then $1/H(\alpha, 1-\alpha) = 1.09$ and $c = 3 + \sqrt{2} \approx 4.41$. Because of the fact that $P \geqq H(\alpha_0, \cdots, \alpha_n)$ ([8], [14], [18]) always, average search time is at most 9% above the optimum.

Proof of Theorem 2. Suppose there are m_j j-nodes $v_1, \cdots, v_{m_j}$ with thickness $q_{j_1}, \cdots, q_{jm_j}$ and depth $a_{j_1}, \cdots, a_{jm_j}$ respectively. Then

$$q_j = q_{j_1} + \cdots + q_{jm_j}$$

and

$$a_j = \min (a_{j_1}, \cdots, a_{jm_j}).$$

Thus

$$P \leqq \hat{P} = \sum_{j=0}^{n} \sum_{i=1}^{m_j} (q_{j_i}/W) \cdot a_{j_i}.$$

An easy induction argument on the height of the D-tree shows

$$\hat{P} \leqq d \cdot H(\alpha_{01}, \alpha_{02}, \cdots, \alpha_{0m_0}, \alpha_{11}, \cdots, \alpha_{1m_1}, \cdots)$$

where $d = 1/H(\alpha, 1-\alpha)$ and $\alpha_{j_i} = q_{j_i}/W$. By the grouping axiom

$$H(\alpha_{01}, \alpha_{02}, \cdots, \alpha_{0m_0}, \alpha_{11}, \cdots, \alpha_{1m_1}, \cdots) = H(\alpha_0, \cdots, \alpha_n) + \sum_{j=0}^{n} \alpha_j \cdot H\left(\frac{\alpha_{j_1}}{\alpha_j}, \cdots, \frac{\alpha_{jm_j}}{\alpha_j}\right).$$

Choose K such that $v_1, \cdots, v_K$, are left of the j-joint and $v_{K+1}, \cdots, v_{m_j}$ are right of the j-joint. Let $\alpha'_j = \alpha_{j_1} + \cdots + \alpha_{j_K}$ and $\alpha''_j = \alpha_j - \alpha'_j$. Again, by the grouping axiom

$$H\left(\frac{\alpha_{j_1}}{\alpha_j}, \cdots, \frac{\alpha_{jm_j}}{\alpha_j}\right) \leqq H\left(\frac{\alpha'_j}{\alpha_j}, \frac{\alpha''_j}{\alpha_j}\right) + \frac{\alpha'_j}{\alpha_j} H\left(\frac{\alpha_{j_1}}{\alpha'_j}, \cdots, \frac{\alpha_{j_K}}{\alpha'_j}\right) + \frac{\alpha''_j}{\alpha_j} \cdot H\left(\frac{\alpha_{j_{K+1}}}{\alpha''_j}, \cdots, \frac{\alpha_{jm_j}}{\alpha_j}\right).$$

Consider nodes $v_1, \cdots, v_K$. Among these, v_1 has maximal depth and v_k has minimal depth (cf. Fig. 2). For $2 \leqq l \leqq k$ let w_l be the father of v_l and let z_l be the other son of w_l. Then

$$\text{th}(w_l) = \text{th}(v_1) + \cdots + \text{th}(v_l) + x,$$
$$\text{th}(z_l) = \text{th}(v_1) + \cdots + \text{th}(v_{l-1}) + x$$

for some number x. Furthermore $\text{th}(z_l) \leqq (1-\alpha)\,\text{th}(w_l)$ since the underlying tree is in BB[α]. Hence

$$\text{th}(v_1) + \cdots + \text{th}(v_{l-1}) \leqq (1-\alpha)(\text{th}(v_1) + \cdots + \text{th}(v_l))$$

for $2 \leqq l \leqq k$. By repeated application of the grouping axiom

$$\begin{aligned}
&\alpha_j' H\left(\frac{\alpha_{j_1}}{\alpha_j'}, \cdots, \frac{\alpha_{j_k}}{\alpha_j'}\right) \\
&\quad = \sum_{l=2}^{k} (\alpha_{j_1} + \cdots + \alpha_{j_l}) \cdot H\left(\frac{\alpha_{j_l}}{\alpha_{j_1} + \cdots + \alpha_{j_l}}, \frac{\alpha_{j_1} + \cdots + \alpha_{j_{l-1}}}{\alpha_{j_1} + \cdots + \alpha_{j_l}}\right) \\
&\quad \leqq \sum_{l=2}^{k} (\alpha_{j_1} + \cdots + \alpha_{j_l}) \\
&\quad \leqq \sum_{l=0}^{k-2} (1-\alpha)^l \cdot (\alpha_{j_1} + \cdots + \alpha_{j_k}) \\
&\quad \leqq (1/\alpha) \cdot (\alpha_{j_1} + \cdots + \alpha_{j_k}).
\end{aligned}$$

Hence

$$H\left(\frac{\alpha_{j_1}}{\alpha_j}, \cdots, \frac{\alpha_{jm_j}}{\alpha_j}\right) \leqq 1 + 1/\alpha$$

and

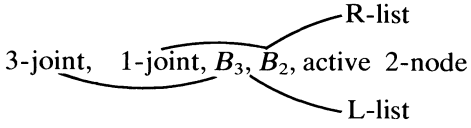
$$P \leqq (1/H(\alpha, 1-\alpha)) \cdot [H(\alpha_0, \cdots, \alpha_n) + 1 + 1/\alpha]. \quad \square$$

The implementation of D-trees with the alternate definition of active nodes is considerably more difficult than the one suggested in § 3. This comes from the following fact: With the old definition of active j-node a different j-node can become active after a transformation only if the distribution of the j-leaves with respect to the j-joint changes. This can only be the case if the j-joint node is involved in the rotation or double rotation. However, this is no longer true for the modified definition of active j-node. A different j-node may become active if the distance of some j-node to the j-joint changes. This can happen even if the j-joint is not involved in the tree transformation but rather the transformation takes place far below the joint node. This forces us to include some additional information in a D-tree.

For every j $(0 \leqq j \leqq n)$: The j-nodes and the j-joint are kept in a doubly linked list in symmetric order. With each link of the list we associate the distance of the j-node (if the j-node is of minimal depth) or the distance to the father of the j-node above. Figure 8 shows the D-tree of Fig. 5 with the additional data structure.

It remains to describe the search and update process. Assume we search for some $X \in (B_j, B_{j+1})$. We descend the tree as directed by the queries and end up in the active j-node. The nodes passed during the descent are stacked. In addition, the nodes are entered into two linked lists, the R-list and the L-list. A node is stored on the R-list if it is left via its right link; the L-list contains all nodes which are left via their left links.

Example. Assume that we search for $X \in (B_2, B_3)$ in the tree of Fig. 8. Then the stack contains



with the R- and L-lists as shown. Furthermore the thickness of every node (including the active j-node) encountered during the descent is increased by one. Then we ascend (using the stack). Suppose we reach node v from node w. We may assume without loss of generality that w is v's right son.

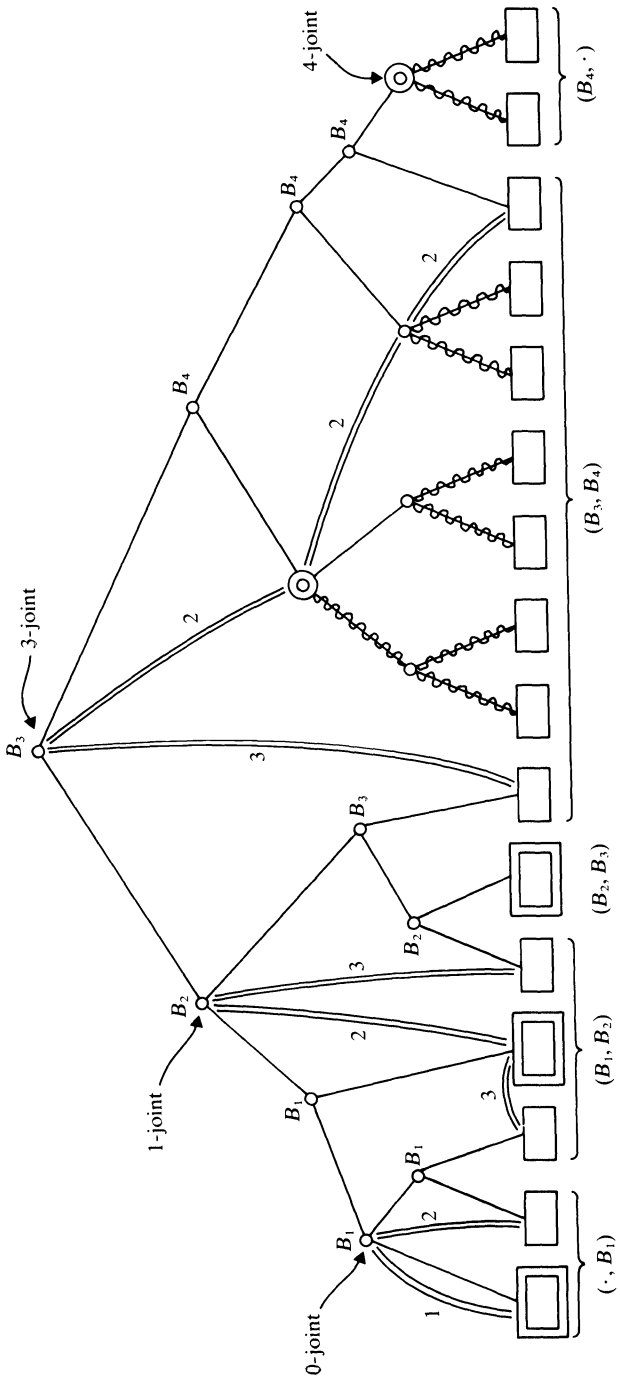


FIG. 8. *The D-tree of Fig. 5 redrawn for the modified definition of active node.*

Case 1. th $(w) \leqq (1-\alpha)$ th (v). Then everything is fine. We delete v from the stack and the R- and L-list (whichever it is on) and ascend one more level.

Case 2. th $(w) > (1-\alpha)$ th (v). We distinguish two cases.

Case 2.1. w is a j-node for some j (see Fig. 9). Then w is the j-node with $X \in (B_j, B_{j+1})$. We split w into 2 j-nodes w_1, w_2 of thickness $\lfloor \text{th}(w)/2 \rfloor$ and $\lfloor \text{th}(w)/2 \rfloor$ respectively. Then $\alpha \leqq \rho(W) \leqq 1/2 \leqq (1-2\alpha)/(1-\alpha)$. Remember that $\alpha \leqq 1-\sqrt{2}/2$ and th $(w) \geqq 2$. By Fact 1 a rotation will rebalance the tree. We obtain Fig. 10.

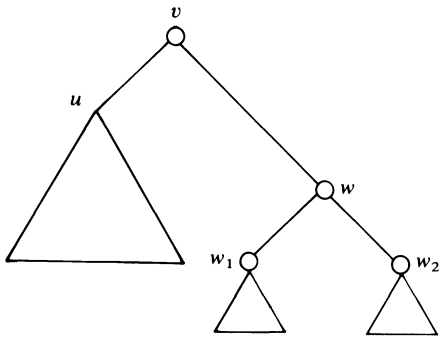


FIG. 9

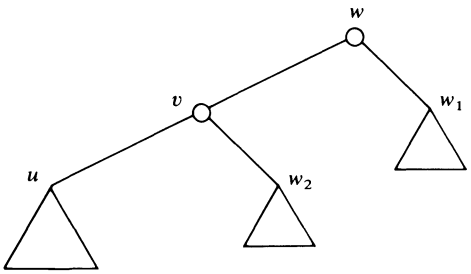


FIG. 10

If either v or w was the j-joint before the rotation then w is the j-joint afterwards. In either case we insert w_1, w_2 (and maybe w) into the proper place of the doubly linked list of j-nodes. The rotation demotes u by one level. This might cause a node below u to become inactive. Let (B_j, B_{j+1}) be the leftmost leaf below u and let x be the first node on the R-list. Three cases may occur (see Fig. 11).

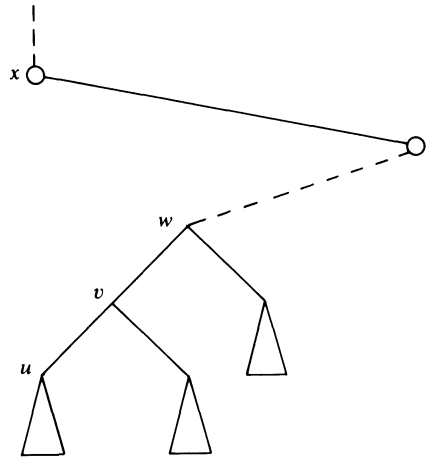


FIG. 11

(a) x is the j-joint. Then we increase the number associated with the link from x to the j-node below u by 1. Then we compare the number with the number associated with the link to the left and change the query assigned to x if necessary. Note that this will direct future accesses to $X \in (B_j, B_{j+1})$ to the newly active j-node.

(b) The other subtree of x is an j-node. Then we increase the distance from this j-node to the j-node below u by 1.

(c) Neither of above. Then the active j-node is also below u and we have nothing to do. This finishes Case 2.1. We delete v from the stack and the R- and L-list (whichever it is on) and ascend.

Note that w is not a j-node for any j after the rotation is performed. This shows that Case 2.1 occurs only in the first step of the ascent.

Case 2.2. w is not a j-node. We have Fig. 12. If $\rho(w) \leqq (1-2\alpha)/(1-\alpha)$ then we perform a rotation, otherwise a double rotation. Since a double rotation is just two rotations we only have to treat the case of a rotation. A rotation about v demotes subtree a one level and promotes subtree b one level. Hence we have to go through the routine described above for the left- and rightmost leaves of subtrees a and b. We also have to distinguish whether u is a j-node for some j or not. The details are left to the reader.

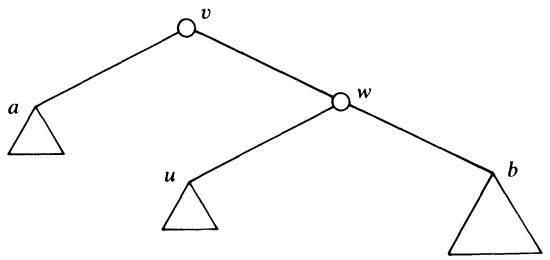


FIG. 12

We summarize the discussion. Modifications of the tree structure are limited to the path of search. In each node of that path a constant amount of work is required. Hence update time is proportional to search time. Theorem 1 is true even for the modified definition of active nodes.

5. Compact D-trees. The tree structure of § 3 achieves one main goal: search time is logarithmically bounded and update time is proportional to search time. However, our solution might use $O(n \cdot \log W)$ storage cells. In addition, the depth of a node, though being bounded by $O(\log W^t/q_j^t)$, can be arbitrarily large compared to n. Consider the case that $q_j^t = 1$ for all t and W^t becomes large.

However, most of the nodes are only present to make rebalancing and bookkeeping easy to explain. In this section we propose a compact version of the tree structure. It exhibits the same search time and update behavior as the basic structure of § 3; in addition, it requires only $O(n)$ storage cells and permits a linear time construction. Also the depth of the active j-node is bounded by $O(\min(n, \log W^t/q_j^t))$.

We obtain a compact tree from a D-tree in the sense of § 3, which we call an extended tree from now on, by node deletion and path compression. The compact tree is formed by the query nodes which contain active nodes in both subtrees, the joint nodes and the active nodes. All other nodes are deleted. Applying this process to the tree of Fig. 5 yields Fig. 13.

We remember the deleted nodes by storing expressions of the form [number, number] along the compressed edges. For example between the node B_4 and the active 4-node we deleted two left subtrees representing a total number of 3 leaves labeled (B_3, B_4) and no leaves labeled $(B_4, \cdot)$. We denote this by the expression [3, 0] on the

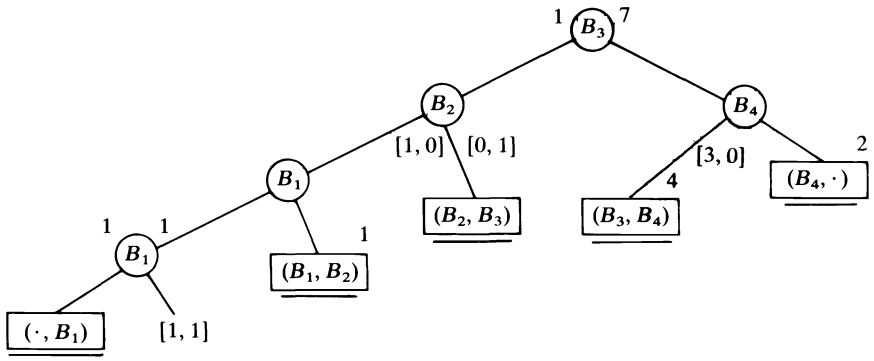


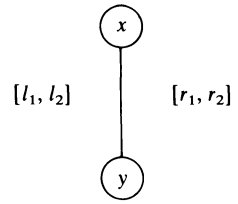
FIG. 13. *The compact version of the tree in Fig. 5.*

left side of the edge joining B_4 and the active 4-node. The right subtree of the 1-joint was deleted completely. It contained one 0-leaf and one 1-leaf. This is denoted by the expression [1, 1].

More formally, the edge labels are assigned as follows. Consider any node x of the extended tree which is not deleted and any edge emanating from it.

Case 1. x has no right (left) descendant which is not deleted. Then the right (left) subtree of x contains no active node and hence at most two kinds of leaves. The right (left) subtree is replaced by the expression $[n_1, n_2]$ where n_1 (n_2) denotes the number of the first (second) kind of leaves. If the query assigned to x is "if $B_j \cdots$ " and the edge is right emanating then n_1 denotes the number of (B_{j-1}, B_j)-leaves and n_2 denotes the number of (B_j, B_{j+1})-leaves. An analogous statement holds if the edge is left emanating.

Case 2. x has a right (left) descendant which is not deleted. Let y be such a descendant of minimal depth. Then all proper descendants of x which are not descendants of y were deleted. (Otherwise y is not minimal depth.) Hence the path from x to y is compacted to a single edge.



The left (right) subtrees along the path from x to y contain no active node and hence at most two kinds of leaves. Their respective numbers are stored in the expressions $[l_1, l_2]$, $[r_1, r_2]$. The above comment about the meaning of n_1, n_2 holds analogously.

Note also that more than one extended tree may be represented by the same compact tree. For example, the compact tree of Fig. 13 above also represents the tree whose right subtree looks like Fig. 14.

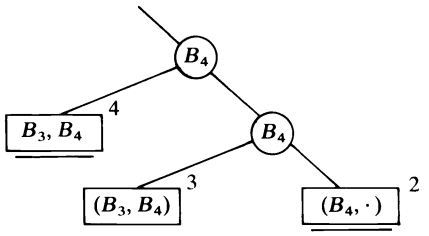
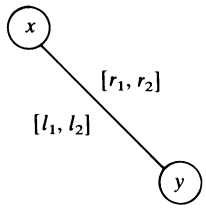


FIG. 14

So every compact D-tree represents a whole class of extended D-trees. We will operate on a compact D-tree as if we were operating on one of the extended D-trees represented by that compact D-tree. Hence we need a method to (locally) construct an extended tree from a compact tree. The method is based on the following facts about the edge labels in compact trees.

LEMMA 2. *Let T be an extended D-tree and let T^c be the compact D-tree constructed from it.*

1) *Consider any edge*



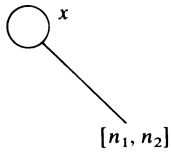
in T^c. Let $l = l_1 + l_2$ and $r = r_1 + r_2$ and $\gamma = \alpha/(1-\alpha)$ where α is the balancing parameter. Then either

(a) $l = r = 0$ *or*

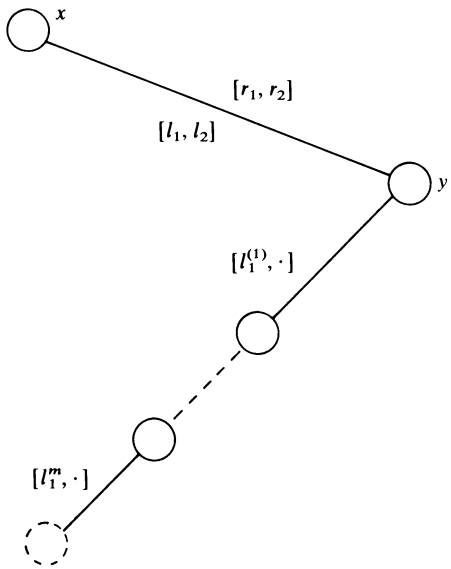
(b) $l \geqq \beta t$ *and* ($r \geqq \gamma(l+t)$ *or* $r = 0$) *or*

(c) $r \geqq \beta t$ *and* ($l \geqq \gamma(r+t)$ *or* $l = 0$).

2) *Consider any node x in the compact tree and an edge right emanating from it. Then either*



and x is a joint node, say the j-joint, and $n_1 \leqq q_j/2$ or



Then y has at least one active descendant. Let the active $(j+1)$-node be the leftmost active descendant of y and let the active $(k-1)$-node be the rightmost active descendant of y. Let $t = \text{th}(y)$.

2.1) *If* $l_2 \neq 0$ *then* $j+1 = k-1$, *y is the active* $(j+1)$*-node,* $r_2 = r_1 = 0$, $l_2 + 1 \leqq t/\gamma$ *and either* $l_1 + l_2 \leqq t/\gamma$ *or* $(t + l_2 + 1) \leqq (l_1 - 1)/\gamma$.

2.2) *If* $r_1 \neq 0$ *then* $j+1 = k-1$, *y is the active* $(j+1)$*-node,* $l_1 = l_2 = 0$, $r_1 + 1 \leqq t/\gamma$ *and either* $r_1 + r_2 \leqq t/\gamma$ *or* $(t + r_1 + 1) \leqq (r_2 - 1)/\gamma$.

2.3) *If* $l_1 \neq 0$ *then x is a descendant of the j-joint; if* $l_2 \neq 0$ *then x is a descendant of the* $(j+1)$*-joint.*

2.4) *If* $r_2 \neq 0$ *then x is a descendant of the* $(k-1)$*-joint; if* $r_1 \neq 0$ *then x is a descendant of the* $(k-1)$*-joint.*

2.5) *If x is the j-joint then let* $[l_1^{(1)}, \cdot], \cdots, [l_1^{(m)}, \cdot]$ *be the edge labels on the left frontier of the subtree with root y. Then* $0 < l_1 + \sum_{p=1}^{m} l_i^{(p)} \leqq q_j/2$.

Proof. 1) The path from x to y in the extended D-tree has length k, $k \geqq 1$. If $k = 1$ then y is a son of x, no subtrees were deleted and hence $l = r = 0$. Suppose $k > 1$. Let $w \neq y$ be the son of x on the path from x to y. Then th $(w) = l + r + t$. Let c_1 (c_2) be the thickness of that subtree of w which contains y (does not contain y). Then either $c_2 \leqq r$ or $c_2 \leqq l$ and $c_1 + c_2 = \text{th}\,(w)$. Furthermore,

$$c_2 \geqq \alpha \cdot \text{th}\,(w),$$

$$c_1 \leqq (1-\alpha)\,\text{th}\,(w).$$

Hence $c_2 \geqq \gamma c_1$ and therefore either $l \geqq \gamma(t+r)$ or $r \geqq \gamma(l+t)$. If y is in the left (right) subtree of w then the same argument applied to a node v (if it exists) on the path from x to y such that y is a right (left) descendant of v finishes the proof.

2) If x has an edge right emanating from it but no right descendant then x does not have active nodes in both of its subtrees. Hence x must be a joint node by the definition of compact D-tree. Say node x is the j-joint. Then n_1 is the number of j-leaves to the right of x and hence $n_1 \leqq q_j/2$ since the active j-node is a left descendant of x.

Suppose x has a right descendant, say y. Since every node in a compact D-tree has at least one active descendant (not necessarily proper), so does y. Let the active $j+1$ be the left-most active descendant of y and let the active $k-1$ node be the rightmost active descendant of y. Suppose $l_2 \neq 0$. l_2 is the number of $(j+1)$-leaves which are members of the left subtrees of the path from x to y in the extended tree. Hence the $(j+1)$-joint w is a proper ancestor of y. w is either to the left of y or to the right of y in tree T. If w is to the left of y then $l_1 = 0$ and the left subtrees of the path from x to y in the extended tree are $(j+1)$-nodes. But then the active $(j+1)$-node is not of minimal depth. Contradiction.

Suppose that the $(j+1)$-joint is to the right of y. Then $k - 1 = j + 1$ and there is only one active descendant of y. Hence y is either the $(j+1)$-joint or the active $(j+1)$-node. In the first case we have a contradiction since w was supposed to be the $(j+1)$-joint. In the latter case $r_2 = 0$ and hence $r_1 = 0$. In the extended tree the path from x to y has $k \geqq 1$ left subtrees. All of them have to contain j-leaves because otherwise y would not be a $(j+1)$-node. The thickness of the left subtree of the father of y must be $\leqq t/\gamma$ where $t = \text{th}\,(y)$ since the underlying tree is in BB[α]. This subtree contains l_2 $(j+1)$-leaves and at least 1 j-leaf. Hence $l_2 + 1 \leqq t/\gamma$. If $k = 1$ then even $l_1 + l_2 \leqq t/\gamma$. If $k > 1$ let $s = l_1' + l_2 > l_2$ be the thickness of the left subtree of the father of y and let l_1'' be the thickness of the left subtree of the grandfather of y. Then $s \geqq l_2 + 1$ and $l_1 \geqq l_1' + l_1'' \geqq 1 + \gamma(t+s) \geqq 1 + \gamma(t + l_2 + 1)$. This proves 2.1). A similar argument proves 2.2).

Suppose $l_1 \neq 0$. Hence x has j-leaves below it in the extended tree and is a descendant of the j-joint. This shows 2.3). The same argument proves 2.4).

Finally 2.5) follows from the definition of active j-node and the fact that the active j-node is a proper left descendant of node x. □

Note as a consequence of Lemma 2 that at most two of the 4 numbers stored in the labels of an edge can be $\neq 0$. This reduces the space requirement of compact trees somewhat. The crucial observation is that the conditions of Lemma 2 are strong enough to characterize compact D-trees. Let T^c be a compact D-tree. In particular, the edge labels of T^c satisfy Lemma 2. We want to construct an extended D-tree T such that compacting T gives T^c. The construction will only use the properties of the edge labels stated in Lemma 2.

Let x be any node in T^c and let B_j be the query assigned to node x. Consider any edge right emanating from x. (Left emanating edges are treated similarly.) If the edge is dangling, i.e., x has no right descendant, then let $[n_1, n_2]$ be the edge label. We have to connect the dangling edge to a subtree with n_1 j-leaves and n_2 $(j+1)$-leaves. If either $n_1 = 0$ or $n_2 = 0$ then we construct a j- or $(j+1)$-node of the appropriate thickness. Otherwise, suppose $n_1 \leqq n_2$ (the symmetric case is treated analogously). Then there exists a $p \geqq -1$ with

$$3 \cdot 2^p \cdot n_1 \leqq n_1 + n_2 \leqq 3 \cdot 2^{p+1} \cdot n_1.$$

In this case we construct Fig. 15. It is easy to see that this tree is weight-balanced.

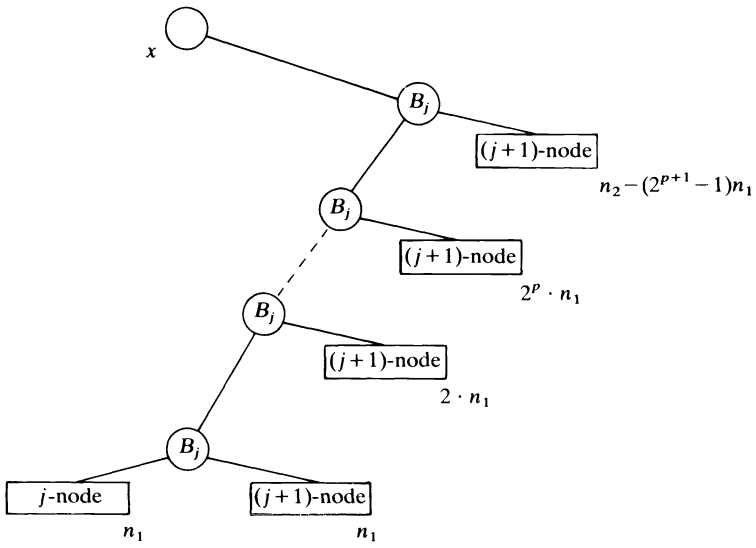


FIG. 15

It remains to consider the case that x has a right son y in the compact tree. Let $[l_1, l_2]$, $[r_1, r_2]$ be the labels of the edge from x to y. If $l_1 + l_2 = r_1 + r_2 = 0$ then there is nothing to do. Assume otherwise. We treat the case $l_1 + l_2 = 0 \neq r_1 + r_2$, the other cases being similar. Let $r = r_1 + r_2$. Then $r \geqq \gamma t$ where $t = \text{th}(y)$ by Lemma 2. In the extended tree there is a path from x to y such that the right subtrees along that path contain r_1 $(k-1)$-leaves and r_2 k-leaves for some k. We show later how to determine k. If $r \leqq t/\gamma$ then we construct Fig. 16.

Note that the newly constructed node satisfies the balance criterion. Suppose $r > t/\gamma$. Let $s = \max(r_1 + 1, t)$. Choose p, $p \geqq -1$, such that

$$2^{(p+1)}(s+t) \leqq (1-\alpha)(t+r) \leqq 2^{(p+2)}(s+t).$$

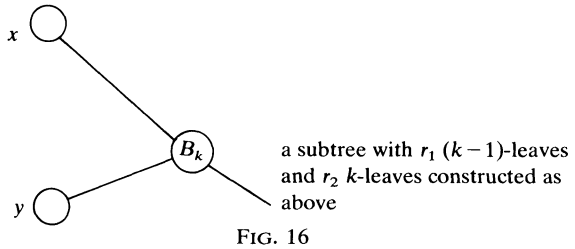


FIG. 16

If $s = r_1 + 1 > t$ and hence $r_1 \neq 0$, then p exists since

$$\begin{aligned}(1-\alpha)(t+r) &= (1-\alpha)(t+r_1+r_2)\\ &\geqq (1-\alpha)(t+r_1+\gamma(t+r_1+1)+1)\\ &= t+r_1+1 = 2^{-1+1}(s+t).\end{aligned}$$

If $s = t$ then p exists since

$$\begin{aligned}(1-\alpha)(t+r) &\geqq (1-\alpha)(t+t/\gamma)\\ &= t(1-\alpha)\cdot(1+1/\gamma)\\ &= t/\gamma > 2\cdot t\end{aligned}$$

since $\gamma = \alpha/(1-\alpha) \leqq 0.43$ and hence $1/\gamma \geqq 2.3$.

We construct the tree shown in Fig. 17. All newly constructed nodes satisfy the balance criterion: The father of y satisfies it since $r_1 + 1 \leqq t/\gamma$; the son of x satisfies it since its thickness is $t+r$, the thickness of its left subtree is $2^{p+1}\cdot(t+s)$ and the choice of p (remember that $\alpha \leqq 1-\sqrt{2}/2$ and hence $(1-\alpha)/2 \geqq \alpha$). The other nodes trivially satisfy the balance criterion. This ends the description of the expansion process.

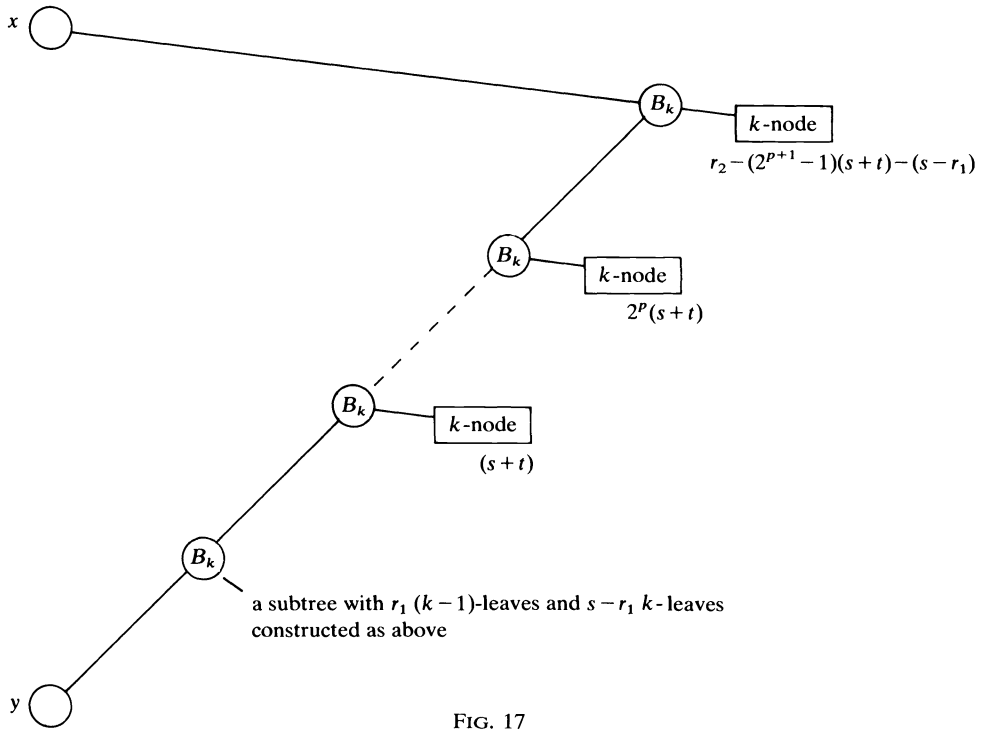
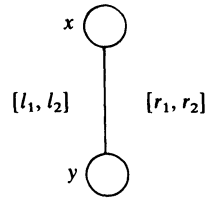


FIG. 17

It remains to be seen that the extended tree obtained in this way is actually a D-tree. The newly constructed nodes satisfy the balance criterion; the other nodes satisfied the balance criterion as nodes of the compact tree and hence satisfy it as nodes of the extended tree.

It is also easy to see that the joint nodes of the compact tree are still joint nodes of the extended tree. Let x be the j-joint in the compact tree. It follows from 2.3) and 2.4) of Lemma 2 that the j-joint in the extended tree cannot be below x. Hence x is the j-joint in the extended tree.

Finally, consider the active j-node z in the compact tree. We want to show that z is the active j-node in the extended tree obtained by the expansion process. Certainly z is on that side of the j-joint which contains $\geqq q_j/2$ of the j-leaves (by 2.5)). Hence if z were not active in the extended tree then there must be a j-node of smaller depth on the same side of the j-joint. Since this j-node does not exist in the compact tree, it was constructed during the expansion process. Hence there must be an edge x



in the compact tree such that the active j-node is the leftmost active descendant of y and $l_2 \neq 0$ or it is the rightmost active descendant of y and $r_1 \neq 0$. In either case y is the active j-node (2.1) or 2.2) of Lemma 2) and either $r_1 + r_2 = 0$ or $l_1 + l_2 = 0$. We treated the case $l_1 + l_2 = 0$, $r_1 \neq 0$ of the expansion process in detail above. It does not introduce any j-nodes of smaller depth than y.

LEMMA 3. *Let T^c be a compact D-tree. The expansion process applied to T^c yields an extended D-tree T.*

Proof. Proof is by the discussion above.

Another property of the expansion process is useful in the sequel. Expansions can be done locally, i.e., knowing the edge label and the thickness of the end point permits proper expansion of an edge. Furthermore, it is possible to expand an edge partially, say, only to generate father and grandfather of the end point y and son and grandson of the starting point x.

We are now able to describe the searching process in compact D-trees. Assume we search for some $X \in (B_j, B_{j+1})$. We descend the tree as directed by the queries and end up in the active j-node. The nodes passed during the descent are stacked, their thickness is increased by one and they are entered into an R-list and L-list as described in § 4. Then we ascend. Suppose without loss of generality we reach node x from its right son y. Two things can happen which require an action. Either the edge label of the edge from x to y does not satisfy Lemma 2 any more or node x has gone out of balance or both.

Case 1. The label of the edge from x to y does not satisfy Lemma 2 any longer. Then either condition 1) or 2.1) or 2.2) is violated. Violations of conditions 2.1) and 2.2) are treated analogously. So suppose 2.2) is violated. Then y is the active $(j+1)$-node for some $j+1$. Let t be the thickness of y before the search. Then $(t + r_1 + 1) \leqq (r_2 - 1)/\gamma < (t + r_1 + 2)$. Furthermore $r_1 + 1 \leqq t/\gamma$. Hence $r = r_1 + r_2 \leqq c \cdot t$ for some constant c dependent only on α. Hence the expansion of the edge from x to y yields a path of bounded length, the bound only depending on α. So we can afford to expand the right subtree of x completely and operate on it as we did in the case of extended D-trees. This shows how to handle violations of 2.1) or 2.2). Suppose now that condition 1) is

violated. Then either $0<l<\gamma(t+1)$ or $0<r<\gamma(t+1)$ (or $r<\gamma(r+t+1)$ and $l<\gamma(r+t+1)$ and $l\neq 0\neq r$). Here t denotes the weight of the right subtree of x before the search. In the first two cases we have to expand the edge from x to y only near y (generation of y's father suffices and in the third case we can afford to expand the edge completely. Note also that using the L- and R-lists it is possible to determine what kind of leaves have to be constructed. In either case we reduced the operations on compact trees to the corresponding operations on expanded trees.

Case 2. Node x has gone out of balance. A rotation or double rotation will rebalance the tree. Performing the transformation may require partially expanding the edges emanating from x. This causes no problems.

We summarize the discussion in

THEOREM 3. *Consider a compact D-tree based on a* BB[α]*-tree with* $0<\alpha\leqq 1-\sqrt{2}/2$.

(a) *Let* q_j^t *be the number searches for* $X\in(B_j, B_{j+1})$, $0\leqq j<n$, *performed up to time* t *and let* $W^t=\sum q_j^t$. *Then at time* t *a search for* $X\in(B_j, B_{j+1})$ *can be executed in time* $O(c_1\log W^t/q_j^t+c_2)$ *where* $c_1=1/\log(1/(1-\alpha))$ *and* $c_2=1+c_1$. *The time needed to update the tree structure is proportional to the search time.*

(b) *A compact D-tree has* $O(n)$ *nodes and edges and thus requires storage space* $O(n)$.

(c) *Given a distribution* $(q_0, q_1, \cdots, q_n)$ *it is possible to construct a compact D-tree for it in linear time* $O(n)$.

Proof. (a) This was proved by the discussion above.

(b) A compact D-tree has $\leqq 2n$ interior nodes and hence $\leqq 4n$ edges. In each node and edge only a fixed constant number of fields are required.

(c) In [17] (cf. also [18]) an algorithm for constructing nearly optimal binary search trees was introduced. It essentially constructs a compact-tree except for the labels of the edges. The labeling process is easily incorporated in that procedure. The details are left to the reader.

6. Extensions.

6.1. General search trees. So far, we considered only searches for elements not in the name set, i.e. $X\notin\{B_1, \cdots, B_n\}$. We drop that restriction and return to the model described in the Introduction. Let p_i (q_j) be the number of searches conducted for $X=B_i(X\in(B_j, B_{j+1}))$ up to now and let

$$W=\sum_{i=1}^{n} p_i+\sum_{i=0}^{n} q_j$$

be the total number of searches conducted so far. Then $\beta_i=p_i/W$ $(\alpha_j=q_j/W)$ is the relative access frequency of $X=B_i$ $(X\in(B_j, B_{j+1}))$ at this point of time.

Define new frequencies q_j', $0\leqq j\leqq n$, by $q_0'=q_0$ and $q_j'=q_j+p_j$ for $1\leqq j\leqq n$, i.e. we change the open intervals (B_j, B_{j+1}) into the half-open intervals $[B_j, B_{j+1})$, and construct the tree structure of § 3 (the compact tree of § 5) for the new set of frequencies. A search for a name X is carried out as above. With search argument $X\in[B_j, B_{j+1})$ we will reach the j-active node. (Note that we assigned queries of the form "**if** $X<B_i$ **then** left **else** right".) In the j-active node we will distinguish between $X=B_j$ and $X\in(B_j, B_{j+1})$ by one more comparison. A search for $X\in[B_j, B_{j+1})$ will take time $O(\log W/q_j')=O(\log W/p_j)$ $(=O(\log W/q_j))$, i.e. we still have logarithmic behavior. The trick used here is due to D. E. Knuth [14, § 6.2.2, exercise 36].

6.2. Insertions. Suppose we want to insert a new name $B\notin\{B_1, \cdots, B_n\}$ into the name set, say $B\in(B_j, B_{j+1})$. A search for B will end in the active j-node representing the

half open interval $[B_j, B_{j+1})$. We have to split the interval $[B_j, B_{j+1})$ and the associated frequency $p_j + q_j$ into two intervals $[B_j, B)$ and $[B, B_{j+1})$ with frequencies $p_j + q_j'$ and $1 + q_j''$ respectively (1 is the frequency of name B and $q_j = q_j' + q_j''$). The splitting of q_j into q_j' and q_j'' may be prescribed arbitrarily.

Using the distribution of j-leaves with respect to the j-joint (stored in the j-joint) and the thickness of the various j-nodes we identify that j-node v such that the j-nodes to the left (right) of v contain $\leqq q_j'$ ($\leqq 1 + q_j''$) j-leaves. We split this j-node into two nodes of type $[B_j, B)$ and $[B, B_{j+1})$. Then we perform the required changes to the tree structure. The j-joint gets new distribution numbers, the active j-node may change and we need to create a new $[B, B_{j+1})$-joint (the father of node v). It is easy to see that these changes can be carried out in time proportional to the depth of the newly created nodes of type $[B_j, B)$ and $[B, B_{j+1})$. This depth is bounded by $O(\log \max (w/q_j', w/q_j''))$ in the case of extended D-trees and by $O(\min (n, \max (w/q_j', w/q_j'')))$ in the case of compact trees.

7. Experimental results. A node of a compact D-tree is represented by six components, the query, the pointers to the two sons, the thickness, and the labels of the incoming edge. The distribution of j-leaves with respect to the j-joint is stored in the son pointers of the active j-node. The type of a node is stored in the sign bits. Since a compact D-tree has between $2n$ and $4n$ nodes and leaves the storage requirements are between $12n$ and $24n$ storage cells. Some additional savings are possible. Lemma 2 of § 5 states that the label of an edge has only one nonzero component except in two special cases. This observation can be used to reduce the space requirement somewhat. A more promising approach is to delete all interior nodes which do not have active nodes in both subtrees. The updating is more complicated in this case but space requirement goes down to $12n$.

Experiments of the following form were carried out. Starting with an arbitrary tree, searches were performed according to a fixed probability distribution. After some number of searches the weighted path length P_D of the D-tree was computed and compared with the weighted path length P_{opt} of an optimal tree for the fixed underlying probability distribution. The optimal tree was constructed by means of the Hu and Tucker algorithm. Two probability distributions were used.

Distribution 1: $n = 200$; $p_i = e^{-100}(100^i/i!)$, $1 \leqq i \leqq 200$, Poisson distribution.

Distribution 2: $n = 200$; the second distribution was obtained by counting the number of words in a German dictionary starting with different two letter combinations.

Table 1 shows the statistics for the first 5000 searches in the case $\alpha = 0.25$. The

TABLE 1
$\alpha = 0.25$, *different distributions.*

	Distribution 1		Distribution 2	
# of searches	$(P_D - P_{opt})/P_{opt} \cdot 100$	#R+DR	$(P_D - P_{opt})/P_{opt} \cdot 100$	#R+DR
0	48.6	0	31.5	0
100	36.3	13	14.4	52
200	33.4	18	12.4	71
500	20.9	22	8.1	106
1000	15.3	25	5.9	145
2000	8.8	33	5.4	187
3000	6.3	34	5.2	207
4000	5.2	34	5.2	229
5000	4.9	35	4.9	244

table shows the deviation (in percent) from the weighted path length of the optimum tree $(P_D - P_{opt})/P_{opt} \cdot 100$ and the total number of rotations and double rotation (# R+DR).

Table 2 shows the statistics for the same initial tree, but different values of α. The second distribution was used. In Table 2 deviation is written to abbreviate $(P_D - P_{opt})/P_{opt} \cdot 100$.

TABLE 2
Same initial tree, 2-nd distribution, different values of α.

# of searches	$\alpha = 0.1$		$\alpha = 0.2$		$\alpha = 0.25$		$\alpha = 1-\sqrt{2}/2$	
	deviation	# R+DR	deviation	# R+DR	deviation	# R+DR	deviation	# R+DR
0	22.9	0	22.9	0	22.9	0	22.9	0
100	23.5	0	22.7	7	18.3	22	14.5	52
200			20.6	20	16.0	38	12.6	79
500	23.9	9	17.0	47	14.5	74	9.9	148
1000	23.3	18	15.1	80	11.7	115	7.6	207
2000	19.9	37	14.6	109	10.5	166	6.7	273
3000	19.6	43	14.4	135	10.5	208	6.2	315
4000	20.9	48	13.9	150	10.1	232	6.0	345
5000	19.4	54	13.8	165	10.1	248	5.8	370

The examples in the tables show that the weighted path length of the compact D-tree approaches the weighted path length of the optimum tree quite rapidly. They also show that the overhead for maintaining the D-tree is not very large. The maximal number of rotations and double rotations in the first 5000 searches were 370. It is interesting to observe here that Blum and Mehlhorn [24] have shown that at most $c \cdot n$ rotations and double rotations suffice to perform n insertions and deletions on an initially empty BB[α] tree. c is a constant independent of n. This is in sharp contrast to the self-organizing binary search trees proposed by Allan and Munro: A rotation about every node on the path of search is required there. In our examples the weighted path length of the optimum tree was about 6.8. Hence about $5000 \cdot 6.8 = 34000$ rotations would be required for the first 5000 searches. This shows that with respect to efficiency self-organizing binary search trees are not competitive with D-trees. However, they use less space.

8. An application to TRIES. An alternative to searching based on key comparison is digital searching. Here a key is identified by successive identification of its component characters. One such method is the TRIE (cf. [14]). A set of strings over some alphabet Σ is represented by its tree of prefixes. So every node of a TRIE corresponds to a word over Σ. Several implementations of TRIES were proposed.

1) Each node of the TRIE is represented by a vector of length $|\Sigma|$. Identification of a character is done by indexing this vector. This method is very fast (one access per character) but it uses a large amount of storage.

2) Each node of the TRIE is represented by a linear list (Sussenguth [22]). In a node w this list contains only those characters $a \in \Sigma$ such that wa is a prefix of some key. Identification of a character is done by a linear search through the list. This method is slow (up to $|\Sigma|$ comparisons per character) but it mostly saves storage space.

3) Each node of the TRIE is represented by a binary search tree (Clampett [4]). In a node w of the TRIE this tree contains those characters $a \in \Sigma$ such that wa is a prefix of some key. Identification of a character is by tree searching. This method is a compromise in speed and space requirement.

There is no a priori reason why the identification of characters has to proceed from left to right; any order will do. Comer and Sethi [5] show that it is *NP*-complete to find the ordering which minimizes average search time under implementation 1.

However, with respect to implementation 3 and nearly optimal average search time, all orderings will do. Let $S = \{B_1, \cdots, B_n\}$ be the set of keys and suppose all keys are of equal length m. For a string $w \in \Sigma^*$ let

$$p_w = |\{B_i;\ w \text{ is a prefix of } B_i\}|.$$

We represent a node w of a TRIE by a D-Tree (or any other kind of nearly optimal search tree) for the distribution $\{p_{wa};\ a \in \Sigma\}$. A key $B_i = a_{i1}a_{i2}, \cdots, a_{im}$ is identified by successively identifying the character a_{ik} in the tree corresponding to the node $a_{i1}, \cdots, a_{i(k-1)}$ of the tree. It takes time $O(c_1 \cdot \log p_{a_{i1},\cdots,a_{i(k-1)}}/p_{a_{i1},\cdots,a_{ik}} + c_2)$ to identify a_{ik} where c_1, c_2 only depend on the balance parameter (cf. Lemma 2). Hence B_i can be identified in time $O(c_1 \log p_\varepsilon/p_{a_{i1},\cdots,a_{im}} + c_2 \cdot m) = O(c_1 \log n + c_2 m)$. Since $\log n$ comparisons are required in any scheme based on comparisons with binary outcome and every character of the input has to be inspected we have nearly optimal TRIES under implementation 3. This problem is discussed in greater detail in Fredman [7] and Güttler, Mehlhorn, Schneider and Wernet [10].

We use D-trees to implement the nodes of a TRIE because we want to deal with updates, i.e. insertions and deletions of names. Suppose we want to insert a new name B into the set S. This amounts to increase p_w by 1 for all prefixes of w. Retaining near optimality is no problem since we used D-trees to implement the nodes of a TRIE. Conversely, suppose we want to delete a name B from the set S. This amounts to decrease p_w by 1 for all prefixes of B. Again it is no problem to retain near optimality since we use D-trees to implement the nodes of a TRIE. (Although we assumed in §§ 3–6 that a search increases the frequency of a node by one we used in the proofs only that it changes the frequency by at most one. A slight complication arises in the case of compact D-trees since the active node can switch sides with respect to the joint node; the reader should have no difficulties in remedying that problem). This leads to the following

THEOREM. *Let S be a set of keys of m characters each. If a TRIE is used to represent the set S and every node of the TRIE is implemented as a D-tree, then searching for a key in S, inserting a new key into S and deleting a key from S can be done in time* $O(\log |S| + m)$.

In database applications keys frequently are m-tuples and comparison between keys is no longer an elementary operation. Balanced tree schemes based on key comparisons (AVL-trees, B-trees, $\cdots$) lose some of their usefulness in this context. In this case TRIES combined with D-trees may prove a real alternative.

We restricted our discussion to keys of uniform length and equal probability. The results are readily extended to the general case [19].

9. Conclusion. We introduced D-trees as an extension of weight-balanced trees. D-trees permit near optimal access under time-varying access probabilities. More precisely, let p be the number of accesses to object B and let W be the total number of accesses up to time t_0. Then at time t_0 an access to object B can be performed with $c_1 \cdot \log(W/p) + c_2$ comparisons between keys for some small constants c_1, c_2. Furthermore, updating the tree structure is limited to the path of search and takes time $O(c_1 \log W/p + c_2)$. On the average only a constant amount of work is required [20], [24].

Two different versions of D-trees are introduced, one favoring access time, the other favoring update time. Compact D-trees were introduced to cut down on the space

requirement of the basic scheme. Finally, an application to TRIES is given. Searching in a set S of multi-attribute keys (length m), inserting into and deleting from it can be done in time $O(c_1 \log |S| + c_2 \cdot m)$.

Similar problems were considered by Allan and Munro [1], Baer [2] and Unterauer [23]. Unterauer and Baer also describe extensions of weight-balanced trees. Unterauer proves bounds on the search time (similar to ours); however update time may be $\Omega(n^2)$ in the worst case. It is O (search time) in the average case. Baer only gives empirical results. Allan and Munro describe an extension of self-optimizing linear list schemes to trees. They derive a bound on the asymptotic average search time; updating is limited to the path of search; however, a rotation about every node of the path of search is necessary.

Acknowledgment. Compact D-trees were implemented by H. Reinshagen and A. Del Fabro. They kindly provided some of the results of their experiments.

REFERENCES

[1] L. ALLAN AND J. MUNRO, *Self-organizing binary search*, Proc. 17th IEEE symposium on Foundation of Computer Science, 1976.

[2] J. L. BAER, *Weight-balanced trees*, Proc. AFIPS National Computer Conference, Vol. 44 (1975), pp. 467–472.

[3] P. BAYER, *Improved bounds on the costs of optimal and balanced binary search trees*, Dept. of Computer Science, Mass. Inst. of Technology, Cambridge, 1975.

[4] H. A. CLAMPETT, *Randomized binary searching with the tree structures*, Comm. ACM, 7 (1964), no. 3, pp. 163–165.

[5] M. COMER AND R. SETHI, *Complexity of Trie index construction*, 17th IEEE Symposium on Foundations of Computer Science, 1976, pp. 197–207.

[6] E. FREDKIN, *Trie memory*, Comm. ACM, 3 (1960), no. 9, pp. 490–499.

[7] M. L. FREDMAN, *Two applications of a probabilistic search technique: Sorting $X+Y$ and building balanced search trees*, Proc. 7th Annual ACM Symp. on Theory and Computing, 1975.

[8] E. N. GILBERT AND E. F. MOORE, *Variable length binary encodings*, Bell System Tech. J., 38 (1959), pp. 933–968.

[9] C. C. GOTLIEB AND W. A. WALKER, *A top-down algorithm for constructing nearly optimal lexicographical trees*, Graph Theory and Computing, Academic Press, 1972, pp. 303–323.

[10] R. GÜTTLER, K. MEHLHORN, W. SCHNEIDER AND N. WERNET, *Binary Search Trees: Average and Worst Case Behaviour*, GI-Jahrestagung 1976, Informatik Fachberichte Nr. 5, Springer-Verlag, Berlin, 1976.

[11] G. HOTZ, *Schranken für die mittlere Suchzeit bei ausgewogenen Verteilungen*, Theoretical Computer Science, 3 (1977), pp. 51–59.

[12] 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.

[13] D. E. KNUTH, *Optimum binary search trees*, Acta Informatica, 1 (1971), pp. 14–25, 270.

[14] ———, *The Art of Computer Programming, Vol. III*, Addison-Wesley, Reading, MA, 1973.

[15] J. VAN LEEUWEN, *On the construction of Huffmann trees*, Proc. 3rd Coll. on Automata Languages and Programming, 1976, S. Michaelson, ed., University Press, Edinburgh.

[16] K. MEHLHORN, *Nearly optimal binary search trees*, Acta Informatica, 5 (1975), pp. 287–295.

[17] ———, *Best possible bounds on the weighted path length of optimum binary search trees*, this Journal, 2 (1977), pp. 235–239.

[18] ———, *Effiziente Algorithmen*, Teubner-Verlag, Stuttgart, 1977.

[19] ———, *Some Remarks on Digital Searching*, Troisième Colloque de Lille, Feb. 1978, Lille, France.

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

[21] J. RISSANEN, *Bounds for weighted balanced trees*, IBM J. Res. Develop., 17 (1973), pp. 101–106.

[22] E. H. SUSSENGUTH, *Use of tree structures for processing files*, Comm. ACM, 6 (1963), no. 5, pp. 272–279.

[23] K. UNTERAUER, *Optimierung gewichteter Binärbäume zur Organisation geordneter dynamischer Dateien*, Doktor-arbeit, TU München, 1977.

[24] N. BLUM AND K. MEHLHORN, *On the average behavior of weight-balanced trees*, Technischer Bericht-, FB 10 der Universität des Saarlandes, Saarbrücken, West-Germany, 1978.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0007 $01.00/0

A COUNTEREXAMPLE TO REINGOLD'S PUSHDOWN PERMUTER CHARACTERIZATION THEOREM*

CARL R. CARLSON†

Abstract. There is a well-known class of algorithms for permuting symbols which has been formally characterized by a device called a pushdown permuter. A theorem attempting to characterize the type of permutations that can be achieved by a pushdown permuter has appeared in the literature. This paper presents a counterexample to this theorem.

Key words. algorithm, permutation, pushdown permuter, stack

1. Introduction. In the theory of computing, there is a well-known algorithm which reads infix arithmetic expressions one symbol at a time from left to right and produces the postfix form of these expressions. This algorithm is of interest here because it is representative of an important class of algorithms which are frequently used by software engineers. This class is characterized by its use of a single pushdown stack and a finite number of random access memory cells as temporary memory locations for storing symbols which are read, one at a time, from an input string and are later placed in the output string. Reingold [1] has developed a formal model for this class of algorithms, which he calls a pushdown permuter, and has attempted to characterize the type of permutations that can be achieved by a pushdown permuter. In this paper, a counterexample to Reingold's pushdown permuter characterization theorem is presented.

2. Pushdown permuter. Reingold defines a *pushdown permuter* to be a variant of a one-way deterministic finite state pushdown transducer with a finite number of random access memory cells. At each time step, a pushdown permuter (p.d.p.) can perform any one of the following actions: (a) It can read the input string one symbol at a time from left to right until it reaches an end-marker. Each symbol read from the input string can be either placed in the output string, which is also produced one symbol at a time from left to right, or placed on top of the pushdown stack or placed in a vacant memory cell. Once a symbol has been placed in the output string, it becomes inaccessible. (b) At any time, the only symbol accessible on the stack is the top symbol, which can be removed from the stack and either thrown away or placed in a vacant memory cell or placed in the output string. (c) At any time, a symbol stored in any one of the memory cells can be removed from that cell and either thrown away or placed on the pushdown stack or placed in the output string.

When the function of a p.d.p. is limited to just the permutation of the symbols from the input string, then the capability of a p.d.p. to throw away symbols (described in (b) and (c)) is not needed. Thus, for the purposes of both this and Reingold's papers, this capability could have been eliminated from the definition of a pushdown permuter.

3. Counterexample. Reingold's theorem states that a p.d.p. with M memory cells can permute the input string $1\,2\cdots n$ to $p_1p_2\cdots p_n$ if, and only if, there is no subsequence[1] $x, y_1, \cdots, y_{M+1}, z_1, \cdots, z_{M+1}$ of $p_1p_2\cdots p_n$ such that for all i and j, $x > z_i > y_j$.

* Received by the editors May 8, 1978.

† Electrical Engineering and Computer Science Department, Northwestern University, Evanston, Illinois 60201.

[1] Reingold states that $p_{i_1}, p_{i_2}, \cdots, p_{i_k}$ is a subsequence of $p_1p_2\cdots p_n$ provided that $1 \leqq i_1 < \cdots < i_k \leqq n$.

Input string:	1	2	3	4	5	6
Output string:	4	1	6	2	3	5

FIG. 1. *Counterexample to Reingold's theorem.*

The permutation shown in Fig. 1 contradicts the "if" part of this theorem. That is, for $M = 1$, there is no subsequence x, y_1, y_2, z_1, z_2 of the output string such that for all i and j, $x > z_i > y_j$. Thus, according to the theorem, there should exist a p.d.p. having just one memory cell which can perform the specified permutation. However, as the reader will soon see, no such p.d.p. exists.

TABLE 1
Steps taken by a p.d.p. with one memory cell.

Steps	Input String	Output String	Stack*	Memory ($M = 1$)
	123456	—	—	—
1	23456	—	—	1
2	3456	—	2	1
3	456	—	32	1
4	56	4	32	1
5	56	41	32	—
6	56	41	2	3
7	6	41	52	3
8	—	416	52	3

* The leftmost symbol occupies the top position on the stack.

Table 1 shows the sequence of steps taken by a particular p.d.p. with only one memory cell. This p.d.p. is unable to perform the permutation described in Fig. 1, because symbol 2 is below the top of the stack and, therefore, inaccessible when it is to be placed into the output string. From the following discussion about the steps taken by this particular p.d.p., it should be clear that no other p.d.p. with only one memory cell can perform the desired permutation either.

Steps.

(1–3) When symbol 1 is read, it can be placed either on the stack or in the single random access memory cell. If it is placed on the stack, then either symbol 2 or 3 will be on top of it in the stack when symbol 4 is read and placed in the output. Thus, symbol 1 would be inaccessible when it is to be placed in the output. If symbol 1 is placed in the single random access memory cell, then the only place to store symbols 2 and 3 is on the stack.

(4–5) When symbol 4 is read from the input string, it is placed in the output string directly. Symbol 1 can then be placed in the output string, since it is accessible.

(6–8) Symbol 5 must be read from the input string in order to gain access to symbol 6, which is the next symbol to be placed in the output string. Because of the decision to place symbol 3 in the vacated memory cell (step 6), the only place to store symbol 5 is on the stack. Had step 6 not been taken, then symbol 5 could have been stored in the single memory cell instead. However, in both cases the results are the same. Symbol 6 is then read from the input string and placed in the output string directly. At this point the p.d.p. is unable to write symbol 2, which is the next output symbol, because it is below the top of the stack. It should be clear from this that no p.d.p. with only one memory cell can perform the desired permutation of the input string. Thus, the "if" part of the

theorem is false. It should be noted that this is not an isolated counterexample. Rather, several counterexamples can be constructed for each value of M.

REFERENCE

[1] E. M. REINGOLD, *Infix to prefix translation: The insufficiency of a pushdown stack*, this Journal, 1 (1972), pp. 350–353.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0008 $01.00/0

COMBINATORIAL ANALYSIS OF AN EFFICIENT ALGORITHM FOR PROCESSOR AND STORAGE ALLOCATION*

E. G. COFFMAN, JR.† AND JOSEPH Y-T. LEUNG‡

Abstract. An *NP*-complete bin-packing problem is studied in which the objective is to maximize the number of pieces packed into a fixed set of equal capacity bins. Applications to processor and storage allocation in computer systems are discussed, and an efficient approximation algorithm is defined and studied. The main results are bounds on the complexity of the algorithm and on its performance.

Key words. bin-packing, storage allocation, scheduling theory, combinatorial algorithms, approximation algorithms, worst-case performance bounds

1. Introduction. Allocation problems arising in computer operation include maximizing the number of records stored in multiple, autonomous storage units and maximizing the number of tasks that can be executed on multiple processors over a fixed time interval. Solutions to the former problem lead to efficient accessing patterns (maximizing the number of records in fast-access storage), and solutions to the latter are clearly appropriate to deadline or real-time scheduling on several processors. Both of these allocation problems are mathematically equivalent to the abstract bin-packing problem in which the number of bins is fixed and the object is to maximize the number of pieces packed.

To fix on specific applications, consider the case where storage must be assigned to a large collection of variable-length records (data-sets, programs, etc.). Fast storage consists of multiple, autonomous units; these could be disk cylinders, or they could be pages in a paged memory system. Assuming that each record is to be accessed with equal likelihood (an assumption that is often necessary by default), the problem is to maximize the number of records stored in fast memory. For in this way we achieve a minimum average access time to the records.

In real-time multiprocessor applications it may be desirable to maximize the number of independent tasks that can be completed prior to a given deadline. In fact, the real problem may be to select a minimum deadline by which all of the tasks can be completed. As can be seen, the link between the allocation and sequencing applications of any bin-packing problem is obtained from the term associations: Storage unit—processor, storage capacity—deadline, records—tasks.

Algorithms for the classical bin-packing problem in which the object is to minimize the number of bins used are proposed and analyzed in [1]–[3]; an application to multiprocessor scheduling is studied in [4]. Recently, fast heuristics for our problem of maximizing the number of pieces packed have been proposed and analyzed in [5]. In the sequel we shall mention these results, and then propose a new algorithm which, at a modest increase in complexity, is still very fast and provides a significantly superior performance.

Each of the problems we have mentioned or referenced is easily shown to be *NP*-complete [6], [7]. For this reason we are moved to consider fast (polynomial-time) heuristic algorithms; our approach to the characterization of their performance will be the usual one of assessing their worst-case performance relative to the best achievable.

* Received by the editors September 29, 1977, and in revised form August 1, 1978.

† Department of Electrical Engineering and Computer Science, University of California, Santa Barbara, California 93106.

‡ Department of Computer Science, Northwestern University, Evanston, Illinois 60201.

As we shall see, fast heuristics can be devised for our particular problem which produce packings never having fewer than 6/7 the number of pieces in an optimum packing. This improves substantially over earlier heuristics which could produce packings having as few as 3/4 the number of pieces in an optimum packing.

In the next section we shall describe the basic model and present the main results. In § 3 we shall summarize results and draw some conclusions.

2. The model and main results. In this section our terminology will refer to abstract bin-packing models; the mapping of this terminology into the practical problems mentioned in other sections will be obvious.

We assume a fixed set of $m \geqq 1$ identical, equal capacity bins $B_1, \cdots, B_m$, and for convenience we assume that the common capacity is unity. We are given a list of $n \geqq 1$ pieces, $L = (p_1, p_2, \cdots, p_n)$; the piece sizes (as well as names) are denoted by p_i, $1 \leqq i \leqq n$, and are constrained to be in the interval $(0, 1]$. We wish to consider algorithms for packing into the m bins as much of L as possible.

In a search for approximation algorithms we can make use of the simple observation that we need only consider packing sets of smallest pieces; i.e. if $p_1 \leqq \cdots \leqq p_n$, then we can restrict ourselves to algorithms for packing prefixes of the list L. As a consequence we shall hereafter assume that L is in the above nondecreasing order of piece size.

In view of the results for classical bin-packing, an algorithm that quickly comes to mind is the one that scans L from left to right, placing successive pieces into the lowest indexed bin into which they will fit. Because of the assumed ordering of L, this is called the first-fit-increasing (FFI) algorithm. Figure 1 shows an example. Note that the bins are filled one by one; as soon as B_i begins to be filled, $B_1, \cdots, B_{i-1}$ remain fixed for the remainder of the packing sequence.

Let $n_{\mathrm{FI}}(L, m)$ and $n_0(L, m)$ denote the number of pieces packed from L into m bins by the FFI algorithm and an optimization algorithm, respectively. It is known [5] that for all L and m

$$n_0(L, m) \leqq 4/3 n_{\mathrm{FI}}(L, m) \tag{1}$$

In an effort to improve on this worst-case performance we consider the iterated first-fit-decreasing (FFD*) algorithm which works as follows. The algorithm first scans L to find the maximum length prefix $L^{(1)} = (p_1, \cdots, p_t) \subset L$ such that $\sum_{i=1}^{t} p_i \leqq m$. The algorithm then packs $L^{(1)}$ into as many, say m', bins as required, by scanning right to left and placing the next smaller piece into that bin with lowest index into which it will fit. The algorithm terminates successfully if $m' \leqq m$; otherwise, the algorithm constructs $L^{(2)} \subset L^{(1)}$ by discarding the largest piece in $L^{(1)}$ and then proceeds as above to pack $L^{(2)}$

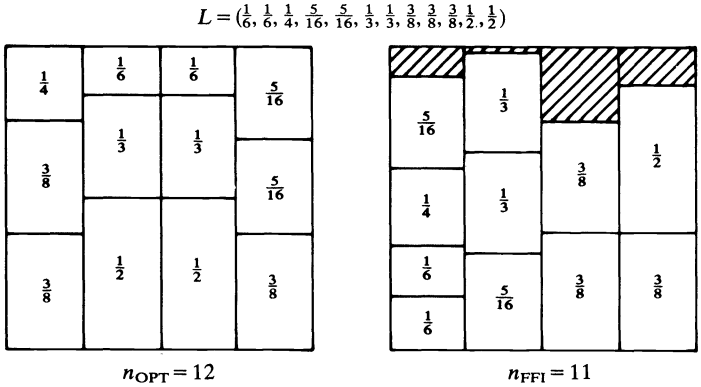


FIG. 1. *An FFI example.*

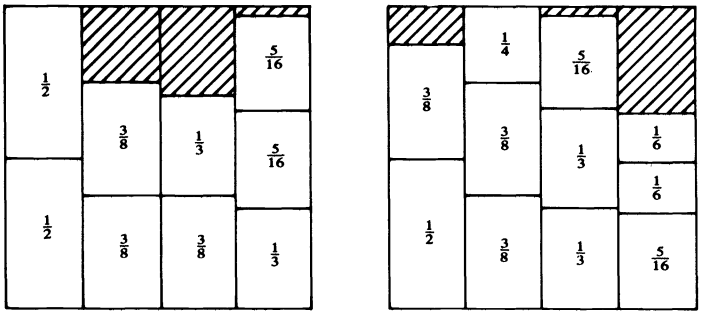


FIG. 2. *An FFD* example (L as in Fig. 1).*

by the FFD rule. This process is repeated until for some j, $L^{(j)}$ has been packed into $m' \leqq m$ bins. Figure 2 shows an example of the iterations involved in the FFD* algorithm.

In contrast to the FFI algorithm the FFD* algorithm does not possess a simple structure. First of all, it is iterative by nature, and secondly it is not on-line with respect to the bins. Moreover, it possesses certain anomalies; e.g. one can shorten certain lists by one piece only to find that more iterations are required by FFD*. Similarly, by adding only one piece to certain lists that FFD* packs into $m-1$ bins we can produce lists that FFD* can only pack into $m+1$ bins. The basis for such anomalies can be found in examples described in [3], [4], where it is noted that the presence of such anomalies effectively rules out the possibility of relatively simple induction arguments for analyzing the performance of algorithms based on the FFD rule.

In view of such anomalies it may be somewhat surprising to discover that the FFI algorithm never outperforms the FFD* algorithm.

THEOREM 1. *Let $n_{FD^*}(L, m)$ denote the number of pieces that the FFD* algorithm packs from L into m bins. Then for all L and $m \geqq 1$, $n_{FD^*}(L, m) \geqq n_{FI}(L, m)$.*

Proof. Consider the FFI packing $B_1, \cdots, B_m$ of a prefix L' of a given list L. Let $B'_i = B_{m-i+1} (1 \leqq i \leqq m)$; i.e. $B'_1, \cdots, B'_m$ is the reverse of the FFI sequence. Let $B''_1, \cdots B''_m$ be the FFD packing of L' into m bins. It is routine to show that the FFD packing of L' is such that $p \in B'_i$ implies $p \in B''_j$ for some $j \leqq i$. It follows easily that the FFD* algorithm must pack a (possibly proper) superset of L' in L. □

Next, it can be shown that, as a function only of n, the number of pieces, the FFD* algorithm has the same $O(n \cdot \log_2 n)$ worst-case time complexity as the FFI algorithm (resulting from the initial ordering of L). Moreover, as a function only of m the complexity of the FFD* algorithm is $O(m \cdot \log_2 m)$. Specifically, we have the following result.

THEOREM 2. *The FFD* algorithm requires no more than m iterations; the worst-case time complexity of the FFD* algorithm is therefore* $O(n \cdot \log_2 n + mn \cdot \log_2 m)$.

Proof. The time required by one iteration of the FFD* algorithm is easily shown to be $O(n \cdot \log_2 m)$ at worst. Thus, the result is obtained from the following proof that the algorithm requires at most m iterations.

Assume on the contrary that $L = (p_1, \cdots, p_n)$ violates this assertion in m bins. Let t be the index such that $L' = (p_1, \cdots, p_t)$ is the maximum-length prefix of L with the property that $\sum_{i=1}^{t} p_i \leqq m$. From the definition of FFD*, the algorithm must start its initial iteration with L', successively discarding the largest piece until all remaining pieces fit in an FFD packing of no more than m bins. Now consider the FFD packing produced at the mth iteration. The pieces $p_t, p_{t-1}, \cdots, p_{t-(m-1)+1}$ must have been discarded. Moreover, by our assumption that L violates the assertion in m bins, there must be a piece p_r that cannot be made to fit in the first m bins of the FFD packing. Each of the first m bins of the FFD packing must have a level exceeding $1 - p_r$. Therefore, we have

$$\sum_{i=1}^{t-m+1} p_i > m(1-p_r) + p_r = m - (m-1)p_r.$$

Since $p_i \geqq p_r$ for $t-(m-1)+1 \leqq i \leqq t$, we have

$$\sum_{i=1}^{t} p_i = \sum_{i=1}^{t-m+1} p_i + \sum_{i=t-(m-1)+1}^{t} p_i > m - (m-1)p_r + (m+1)p_r = m.$$

This contradicts the fact that $\sum_{i=1}^{t} p_i \leqq m$, and hence the result follows. □

It follows directly from the above result that

$$n_0(L, m) \leqq n_{\mathrm{FD}^*}(L, m) + m - 1. \tag{2}$$

So far we know only that the FFD* algorithm is never worse than the FFI algorithm. The next theorem, which comprises the main result, shows that the worst-case FFD* performance is substantially closer to that of an optimization algorithm than is the worst-case FFI performance. Unfortunately, a detailed proof of this result [8] requires well over 100 pages and hence can not be presented here in full. However, the structure of the proof can be presented and fully illustrated; that part of the proof omitted is largely mechanical.

THEOREM 3. *For all L and* $m \geqq 1$, *we have*

$$n_0(L, m) \leqq 7/6 n_{\mathrm{FD}^*}(L, m) + 3. \tag{3}$$

Moreover, there exist lists for every even m such that

$$n_0(L, m) = (8/7) n_{\mathrm{FD}^*}(L, m).$$

Proof. Let m be even and consider a list L of $n = 4m$ pieces such that $p_i = 1/4 - \varepsilon$ for $1 \leqq i \leqq 2m$ and $p_i = 1/4 + \varepsilon$ for $2m + 1 \leqq i \leqq 4m$, where $0 < \varepsilon < 1/20$. It is routine to show that $n_0(L, m) = (8/7) n_{\mathrm{FD}^*}(L, m)$. Figure 3 shows the general case.

The proof of the upper bound proceeds by contradiction, assuming that there is a list $L = (p_1, \cdots, p_n)$ violating (3). We begin by characterizing in a general way the nature of such a counterexample, concluding that only a finite number of cases need to be considered; the remainder of the proof verifies that (3) is in fact satisfied by each case.

We may suppose that all of the pieces in the list L violating (3) can be packed into m bins by an optimization algorithm. With this assumption it is now more convenient to suppose that $p_1 \geqq p_2 \geqq \cdots \geqq p_n$ and that each iteration of the FFD* rule packs pieces in a sequence $p_j, p_{j+1}, \cdots, p_n$ for some $1 \leqq j \leqq n$. At any given iteration we define the FFD

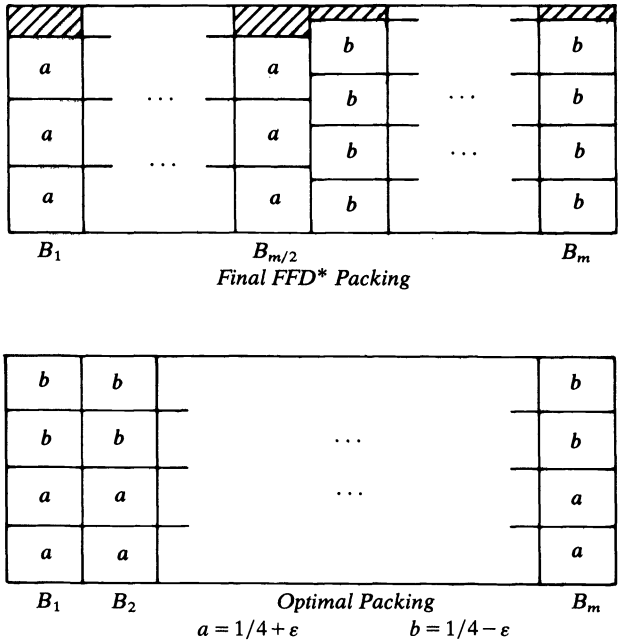


FIG. 3. *The 8/7 example.*

packing of $p_j, \cdots, p_n$ to consist of only those pieces packed in the first m bins when first a piece is assigned to the $(m+1)$st bin, or when p_n is packed, whichever occurs first. We use n_{FD} to denote the number of pieces in an FFD packing.

Now consider the *penultimate* pass of the FFD* algorithm and define the indices u and v such that $p_u, p_{u+1}, \cdots, p_{v-1}$ are just those pieces in the FFD packing of $p_u, \cdots, p_n$, $1 \leqq u < v \leqq n$. Thus, $p_{u+1}, \cdots, p_v, \cdots, p_n$ are the pieces packed in the final iteration and we have $n_{FD^*}(L, m) = n_{FD}((p_u, \cdots, p_v), m) + (n - v)$. Since $n_0(L, m) = n_0((p_1, \cdots, p_v), m) + (n - v)$ it is easily seen that violation of (3) implies $n_0((p_1, \cdots, p_v), m) > (7/6)n_{FD}((p_u, \cdots, p_v), m) + 3$. Thus, we may now concentrate on the comparison of the FFD packing of $(p_u, \cdots, p_v)$. In other words, our reduced problem consists of two sub-lists $L_1 = (p_1, \cdots, p_{u-1})$ and $L_2 = (p_u, \cdots, p_v)$, $p_i \geqq p_j$ $(i \leqq j)$, such that

(i) the FFD rule packs all but the smallest piece (p_v) of L_2 into m bins,

(ii) an optimization algorithm packs all pieces of both L_1 and L_2 into m bins, and

(iii) if $n_{FD} = v - u$ and $n_0 = v$ denote the respective numbers packed, then

$$n_0 > (7/6)n_{FD} + 3. \tag{4}$$

The proof now focuses on the reduced problem and verifies that (4) can not in fact hold. Let us assume, as we may, that given L_1, L_2 and m are respectively the smallest list and number of bins for which (i)–(iii) hold. This assumption implies a number of properties which greatly simplify our problem.

CLAIM 1. *The FFD packing, P_F, of L_2 in m bins (which fails to pack only the smallest piece p_v) has at least two pieces per bin. Hence, $n_{FD} \geqq 2m$.*

Proof. From the nature of the FFD rule pieces uniquely occupying bins in P_F must also uniquely occupy bins in an optimum packing. By eliminating such pieces it is readily seen that we can construct a smaller list L_2 satisfying (i)–(iii) in a smaller number of bins. □

CLAIM 2. *$P_U > 1/6$ and hence $p_i > 1/6$, $1 \leqq i \leqq u$.*

Proof. If $p_u \leqq 1/6$, then there are at least six pieces per bin in P_F, and hence $n_{FD} \geqq 6m$. From (2) and the definition of u we have $n_0 - n_{FD} = u = n_0(L, m) - n_{FD}(L, m) \leqq m - 1$. Thus, $n_0/n_{FD} = 1 + (n_0 - n_{FD})/n_{FD} \leqq 1 + (m-1)/(6m) < 7/6$ which contradicts (4). □

CLAIM 3. *If $p_u \leqq 1/k$, then $p_v \leqq 1/(k+1)$.*

Proof. Suppose on the contrary that $p \in (1/(1+k), 1/k]$ for all p in P_F. Then there are k pieces per bin in P_F. But there are no k or more pieces in L_1 or P_F with which p_v can be packed in a single bin. Hence, we have the contradiction that p_v cannot be placed in any optimum packing along with the km pieces of the FFD packing. □

CLAIM 4. *Let $p_u \in (1/(1+k), 1/k]$. Then $p_v > k/[6(k+1)]$ for $k \geqq 2$ and $p_v > 1/6$ for $k = 1$.*

Proof. The cumulative size of the pieces in an optimum packing must not exceed the total capacity, m. Therefore, $\sum_{i=1}^{u-1} p_i + \sum_{i=u}^{v-1} p_i + p_v \leqq m$. Since p_v does not fit into the FFD packing of $(p_u, \cdots, p_{v-1})$ we have

$$\sum_{i=u}^{v-1} p_i > (1 - p_v)m.$$

Thus, by making use of $p_i \geqq p_u$, $1 \leqq i \leqq u$, we can write

$$(n_0 - n_{FD} - 1)p_u + (1 - p_v)m + p_v < m$$

or

$$p_v > (n_0 - n_{FD} - 1)p_u/m. \tag{5}$$

By hypothesis $n_0 > (7/6)n_{FD} + 3$ and hence $n_0 - n_{FD} - 1 > (1/6)n_{FD} + 2$. But since $p_u \leqq 1/k$, we have $n_{FD} \geqq km$ and consequently $n_0 - n_{FD} - 1 > km/6 + 2$. Substituting into (5) and using $p_u > 1/(1+k)$ we get $p_v > k/[6(k+1)]$ as desired.

Since there must be at least two pieces per bin even when $k = 1$, we obtain from (5): $p_v > (2m/6 + 2)p_u/m > 1/6$, when $p_u > 1/2$. □

As a result of Claims 2–4 the remaining possibilities can be accounted for by disposing of each of the cases shown in Table 1. (Division of the ranges of p_u and p_v into intervals bounded by unit fractions is motivated by the implication that there are exactly k pieces in a bin when each piece-size is in the interval $(1/(k+1), 1/k]$.)

TABLE 1
List of cases.

	p_u	p_v
Cases A1–A4;	(1/2, 1];	(1/3, 1/2], (1/4, 1/3], (1/5, 1/4], (1/6, 1/5].
Cases B1–B6;	(1/3, 1/2];	(1/4, 1/3], (1/5, 1/4], (1/6, 1/5], (1/7, 1/6], (1/8, 1/7], (1/9, 1/8].
Cases C1–C4;	(1/4, 1/3];	(1/5, 1/4], (1/6, 1/5], (1/7, 1/6], (1/8, 1/7].
Cases D1–D3;	(1/5, 1/4];	(1/6, 1/5], (1/7, 1/6], (2/15, 1/7].
Cases E1–E2;	(1/6, 1/5];	(1/7, 1/6], (5/36, 1/7].

The basic strategy in disposing of the above cases is by means of what we shall term a *weighting argument.* Such an argument has the effect of reducing the combinatorics of the general problem to those concerned with the configurations of pieces within individual bins. In particular, a weighting function, f, is defined which is a nondecreasing step function of piece-size, mapping (0, 1] into a finite set of rationals in (0, 1].

With certain variations the weighting function is used as follows. First, for some given α it is shown that the total weight in each bin of an optimum packing (i.e., the sum of the weighting-function values for the pieces in each bin) can not exceed α. Next, it is verified for a given $\beta < \alpha$ that each bin in the FFD packing, except for a small number of so-called *deficit* bins, has a weight no less than β.

The total weight of pieces in an optimum packing can not be less than the sum of the weights of $p_1, \cdots, p_{u-1}$, the weights of the pieces in the FFD packing, and the weight of p_v. Thus, since $p_1 \geqq p_2 \geqq \cdots \geqq p_u$, we have

$$w_F + (u-1)f(p_u) + f(p_v) \leqq w_0$$

where w_F and w_0 are the respective total weights of the FFD and optimum packings. Hence, $m\beta - w_d + (u-1)f(p_u) + f(p_v) \leqq m\alpha$, or

$$u \leqq \frac{1}{f(p_u)}[m(\alpha-\beta) + w_d - f(p_v)] + 1$$

where w_d is the total (deficit) by which the weight of the, say d, deficit bins is exceeded by $d\beta$. Finally, a lower bound $g(m)$ is introduced for n_{FD}. By Claim 1 $g(m) \geqq 2m$, but in certain cases a tighter bound is necessary. Using $n_{\mathrm{FD}} \geqq g(m)$ we derive

$$u \leqq \frac{m(\alpha-\beta)}{f(p_u)g(m)} n_{\mathrm{FD}} + \frac{w_d - f(p_v)}{f(p_u)} + 1.$$

The argument concludes with the contradiction that for the given parameter values

$$\frac{m(\alpha-\beta)}{g(m)f(p_u)} \leqq \frac{1}{6}, \qquad \frac{w_d - f(p_v)}{f(p_u)} + 1 \leqq 3. \tag{6}$$

As will be observed, the number, d, of deficit bins will be independent of m (in fact, $d \leqq 17$ for all cases); hence, their effect is restricted to an additive constant.

Variations in the above argument concern computations of the bounds w_0 and w_F which are slightly more complicated than those outlined above. The proofs of the bin-weight bounds α and β are rather arduous enumerations in a number of cases. For this reason we shall tabulate much of the proof, listing only relevant bin configurations along with their bin weights. The pieces will be classified, with a distinct name for each class, according to the interval in which their size falls. Thus, we shall define A-pieces, B-pieces, etc. Piece class-names increase in lexicographic order as the size decreases. Also, a bin whose largest piece is a λ-piece for some λ will be called a λ-bin. Bin configurations will be identified by piece-name sequences in increasing lexicographic order (decreasing size). Thus, the configuration A C C D refers to an A-bin with an A-piece, two C-pieces, and a D-piece. These conventions are made clear in cases A2 and following.

We now proceed to the case analysis. Because of the space required, we have omitted the proofs of many of the cases listed in Table 1. However, a full version of this paper containing them is available from the authors; it can also be found in [8]. We shall limit ourselves here to proving cases A1, A2, A3, B1, B2, and B6. These cases are fully representative of the arguments used in the cases not presented.

CASE A1. $p_u \in (1/2, 1]$ *and* $p_v \in (1/3, 1/2]$.

Proof. We do not need a weighting argument here, for in this case no bin can contain more than two pieces in *any* packing. Since each bin in the FFD packing has at least two pieces by Claim 1, we have the contradiction that, in order to pack p_v, an optimum packing *must* have a bin with three pieces. □

CASE A2. $p_u \in (1/2, 1]$ *and* $p_v \in (1/4, 1/3]$.

Proof. Let $p_u = 1/4 + \delta$, $0 < \delta \leqq 1/12$. The weighting function, along with the classification of the pieces, is given as follows.

$$f(p) = \begin{cases} 1/2 & 1/2 < p \leqq 1 & \text{A-piece} \\ 3/8 & 3/8 - \delta/2 < p \leqq 1/2 & \text{B-piece} \\ 1/4 & 1/3 < p \leqq 3/8 - \delta/2 & \text{B1-piece} \\ 1/4 & 1/4 + \delta \leqq p \leqq 1/3 & \text{C-piece} \end{cases}$$

Clearly, the FFD packing consists of a sequence of A-bins, followed by a (possibly empty) sequence of B-bins, followed by a (possibly empty) sequence of B1-bins, followed finally by a (possibly empty) sequence of C-bins. Figure 4 illustrates such a

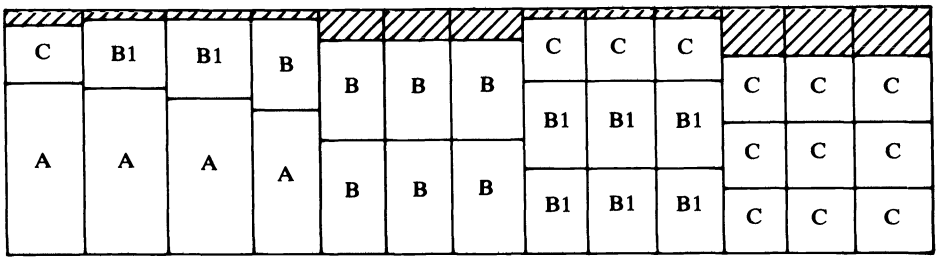


FIG. 4. *FFD packing* (*Case* A2).

packing. In any FFD packing the last λ-bin, for any $\lambda = A, B, \cdots$, will be called the *deficit* λ-bin. All other bins will be called *regular* bins. In drawings such as Fig. 4 we shall endeavor to illustrate all sequences of regular bins. However, it is not possible to show all possible deficit-bin configurations; this will become obvious below.

The next step of the proof determines an upper bound for w_0. This bound can be taken from Table 2A which enumerates the possible bin configurations in an optimum packing and lists the corresponding bin weights. Note that we have not considered bins with more than three pieces in an optimum packing, since the smallest piece size

TABLE 2A

Bin weights in an optimum packing ($\alpha = 7/8$).

2-piece Bins			3-piece Bins						
A	A	impossible	A	A	A	A	A	B	
A	B	7/8	A	A	B1	A	A	C	
A	B1	3/4	A	B	B	A	B	B1	
A	C	3/4	A	B	C	A	B1	B1	impossible
B	B	3/4	A	B1	C	A	C	C	
B	B1	5/8	B	B	B	B	B	B1	
B	C	5/8	B	B	C	B	B1	B1	
B1	B1	1/2	B	B1	C	7/8			
B1	C	1/2	B	C	C	7/8			
C	C	1/2	B1	B1	B1	impossible			
			B1	B1	C	3/4			
			B1	C	C	3/4			
			C	C	C	3/4			

exceeds 1/4. Also, we do not explicitly account for 1-piece bins; it is easy to check that $f(p) \leqq \alpha$ holds for the definitions of f and α here, and in all of the remaining cases. The configurations marked impossible simply denote sets of pieces whose cumulative size exceeds unity. From the table we find that no bin weight exceeds $\alpha = 7/8$ and hence $w_0 \leqq 7m/8$.

In many subsequent descriptions of optimum packings a full listing of all configurations, such as in Table 2A, would be prohibitively long. For this reason, we shall hereafter not list any impossible or dominated configurations. (A configuration $\lambda_1 \lambda_2 \cdots \lambda_r$ is dominated if there is already listed another configuration $\lambda_1' \lambda_2' \cdots \lambda_r'$ such that for each i $(1 \leqq i \leqq r) \lambda_i$ is lexicographically no smaller than λ_i'.) Thus, configurations such as B1 B1 C (dominated by B B1 C) in Table 2A will not be listed. Indeed with these conventions Table 2A reduces to the single entry, A B, for 2-piece bins and the single entry, B B1 C, for 3-piece bins.

We now turn to the FFD packing and verify that every bin, except possibly certain deficit bins, has a total weight at least $\beta = 3/4$. This bound follows from Table 2B where we have enumerated all valid bin configurations. In constructing this table the bin capacity constraint and the fact that p_v exceeds all unused bin capacities are used to determine valid bin configurations. For example, B B C can not appear as a B-bin configuration, since two B-pieces and a C-piece would exceed the bin capacity. Also, B1 B1 can not appear as a B1-bin configuration, since p_v can always fit into a bin along with two B1-pieces.

In the deficit-bin column only those valid configurations which are not valid regular-bin configurations are listed. We shall further reduce the number of entries in the tables describing FFD packings, but the conventions are best deferred to the more appropriate case (Case A3) that follows.

As can be seen from Table 2B every regular-bin weight is at least 3/4. This also applies to deficit-bin weights, except for the B C and B B1 configurations in a deficit B-bin, which have a weight $3/8+1/4=3/4-1/8$, and hence a deficit of 1/8. To complete the proof it remains only to verify that (6) holds for the parameter values $\alpha = 7/8$, $\beta = 3/4$, $w_d = 1/8$, $f(p_u) = 1/2$, $f(p_v) = 1/4$, and $g(m) = 2m$ (from Claim 1). □

TABLE 2B
Bin weights in an FFD packing ($\beta = 3/4$).

Bin	Regular			Weight	Deficit			Weight
A	A	B		7/8				
	A	B1		3/4		—		—
	A	C		3/4				
B	B	B		3/4	B	B1		3/4–1/8
					B	C		3/4–1/8
					B	C	C	7/8
B1	B1	B1	C	3/4	B1	C	C	3/4
C	C	C	C	3/4		—		—

CASE A3. $p_u \in (1/2, 1]$ *and* $p_v \in (1/5, 1/4]$.

Proof. Let $p_v = 1/5 + \delta$, $0 < \delta \leqq 1/20$. We divide this case into two sub-cases which are determined by certain bin configurations in the FFD packing.

CASE A3.1. In the FFD packing, whenever a bin contains a piece with size exceeding 1/2, it also contains a piece with size exceeding $2/5 - \delta/2$.

Proof. The weighting function and piece classification is as follows.

$$f(p)=\begin{cases} 1/2 & 1/2<p\leqq 1 & \text{A-piece} \\ 2/5 & 2/5-\delta/2<p\leqq 1/2 & \text{B-piece} \\ 3/10 & 1/3<p\leqq 2/5-\delta/2 & \text{B1-piece} \\ 4/15 & 4/15-\delta/3<p\leqq 1/3 & \text{C-piece} \\ 1/5 & 1/4<p\leqq 4/15-\delta/3 & \text{C1-piece} \\ 1/5 & 1/5+\delta<p\leqq 1/4 & \text{D-piece} \end{cases}$$

The assumption of the present sub-case will appear in the restricted A-bin configurations. Note that if $\delta = 1/20$ ($p_v = 1/4$), then the class of C1-pieces ceases to exist. Similar to the previous case the FFD packing consists of a sequence of A-bins, followed by a (possibly empty) sequence of B-bins, etc. Figure 5 illustrates such a packing; note that by our assumption each A-bin contains a B-piece.

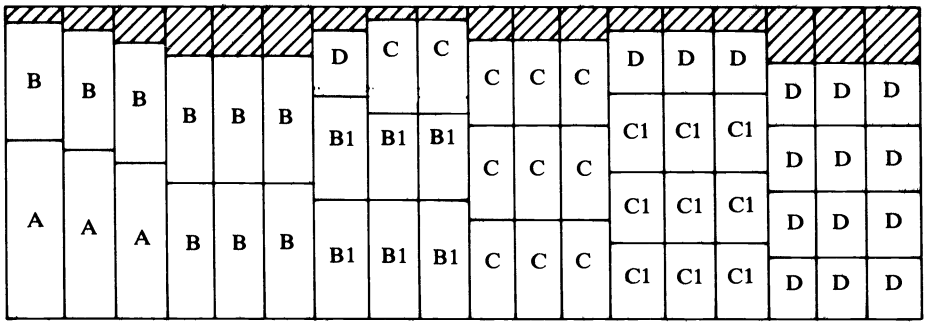


FIG. 5. *FFD packing* (*Case* A3.1).

Table 3A verifies that no bin weight in an optimum packing exceeds 29/30 and hence that we have the bound $w_0 \leqq 29m/30$. (Recall that all impossible and dominated configurations are now suppressed.) For the special case $\delta = 1/20$ the C1's in this and the following table should be replaced by D's.

Table 3B demonstrates that the weight of each regular bin in the FFD packing is at least 4/5. Recall that size constraints and the fact that p_v fits into no FFD bin are used to identify valid configurations. We have simplified the regular-bin column by listing only that valid configuration which has least weight. Thus, in Table 3B a configuration such as B1 B1 C is not listed.

TABLE 3A
Bin weights in an optimum packing ($\alpha = 29/30$).

2-piece bins				4-piece bins				
A	B		9/10	B1	C1	D	D*	9/10
				C	C	C1	D*	14/15
3-piece bins								
A	B	C	29/30					
B	B1	C	29/30					

* For the special case $\delta = 1/20$ this configuration is impossible even after replacing the C1's by D's.

TABLE 3B
Bin weights in the FFD *Packing* ($\beta = 4/5$).

Bin	Regular					Deficit			
A	A	B			9/10				—
B	B	B			4/5	B	B1		1/10
B1	B1	B1	D		4/5	B1	C	D	1/30
C	C	C	C		4/5	C	C1	D	2/15
C1	C1	C1	C1	D	4/5				—
D	D	D	D	D	4/5				—
						Global deficit $w_d = 4/15$			

In the deficit-bin column the numbers shown are the deficits, i.e., the amounts by which the bin weights fall short of β. Dashes appear where there are no deficit bins which produce a nonzero deficit. The listed deficit-bin configurations are restricted to those which produce a maximum global deficit over the entire packing. For example, the configuration B C would give a greater B-bin deficit, but such a configuration would imply the absence of B1-bins and hence the absence of a B1-bin deficit. Because of space required the routine details of determining minimum-weight, regular-bin configurations and maximum global-deficit configurations must be left to the interested reader.

From the tables we use $\alpha = 29/30$, $\beta = 4/5$, and $w_d = 4/15$; using $f(p_u) = 1/2$, $f(p_v) = 1/5$, and $g(m) = 2m$ from Claim 1 we once again arrive at the contradiction in (6).

CASE A3.2. *In the FFD packing there exists a* 2*-piece bin containing a piece of size greater than* $1/2$ *and a piece of size no greater than* $2/5 - \delta/2$.

Proof. We have the following weighting function and piece classification.

$$f(p) = \begin{cases} 3/5 & 8/15-2\delta/3 < p \leqq 1 & \text{A-piece} \\ 8/15 & 1/2 < p \leqq 8/15-2\delta/3 & \text{A1-piece} \\ 2/5 & 2/5-\delta/2 < p \leqq 1/2 & \text{B-piece} \\ 3/10 & 1/3 < p \leqq 2/5-\delta/2 & \text{B1-piece} \\ 4/15 & 4/15-\delta/3 < p \leqq 1/3 & \text{C-piece} \\ 1/5 & 1/4 < p \leqq 4/15-\delta/3 & \text{C1-piece} \\ 1/5 & 1/5+\delta \leqq p \leqq 1/4 & \text{D-piece} \end{cases}$$

By hypothesis the FFD packing has at least one A or A1-bin with a B1, C, C1 or D-piece. Figure 6 illustrates the FFD packing.

We now derive the tighter bound $g(m) = 7m/3$ which is to be used for the present case. Let y denote the sum of the numbers of A, A1, and B-bins. Next, observe that the p_i, $1 \leqq i \leqq u-1$, and the A, A1, and B-pieces must occupy $u-1+y$ bins. The only way fewer bins could be used is for more B-pieces to be paired with the A-pieces, A1-pieces, or the p_i, $1 \leqq i \leqq u-1$. That this can not be possible follows from our assumption that there is an A or A1-bin with a piece smaller than any B-piece; i.e. either there were no more B-pieces to pack in such a bin, or they were all too large to fit. Thus, we must have $u+y-1 \leqq m$. But since we are proceeding by contradiction we have $u-1 > n_{FD}/6+2 \geqq 2m/6+2 > m/3$ by Claim 1. Therefore, $y \leqq 2m/3$. Finally, it is easily verified that there must be at least three pieces in every B1, C, C1, and D-bin. Hence, $n_{FD} \geqq 2y+3(m-y) = 3m-y$ and

$$n_{FD} \geqq g(m) = 7m/3.$$

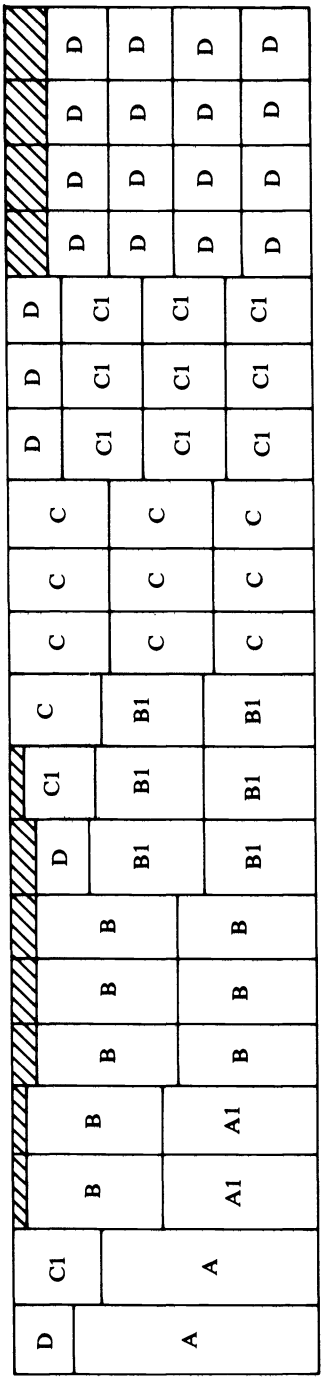


FIG. 6. *FFD packing (case* A3.2).

Table 4A is now used to establish $\alpha = 1$. (For the special case $\delta = 1/20$ the C1 and A1-pieces no longer exist; wherever they appear in this and the following table they are to be replaced by D and A-pieces, respectively). Table 4B establishes $\beta = 4/5$ and a maximum global deficit of $w_d = 4/15$; along with $f(p_u) \geqq 8/15$, $f(p_v) = 1/5$, and $g(m) = 7m/3$ we see that (6) holds. □

TABLE 4A
Bin weights in an optimum packing ($\alpha = 1$).

2-piece bins				4-piece bins				
A	B		1	B1	C1	D	D*	9/10
				C	C	C1	D*	14/15
3-piece bins								
A	C1	D*	1					
A1	C	D*	1					
B	B1	C	29/30					

* For the special case $\delta = 1/20$ this configuration is impossible even after replacing the C1's by D's, and/or the A1's by A's.

We note that the proof of case A4 is similar to the proof of case A3, although a different weighting function is employed. Because of space limitations we shall shorten the proofs of the remaining cases by omitting the tables of bin weights in the optimum and FFD packings. Furthermore, we shall not draw the diagrams illustrating the FFD packings, trusting that the reader can construct them without difficulty. This allows us to fully illustrate the computations of tighter bounds for w_0 and w_F that are necessary in certain cases.

TABLE 4B
Bin weights in the FFD packing ($\beta = 4/5$).

Bin	Regular					Deficit			
A	A	D			4/5				—
A1	A1	C			4/5				—
B	B	B			4/5	B	B1		1/10
B1	B1	B1	D		4/5	B1	C	D	1/30
C	C	C	C		4/5	C	C1	D	2/15
C1	C1	C1	C1	D	4/5				—
D	D	D	D	D	4/5				—
						Global deficit $w_d = 4/15$			

CASE B1. $p_u \in (1/3, 1/2]$ *and* $p_v \in (1/4, 1/3]$.
Proof. Let $p_v = 1/4 + \delta$, $0 < \delta \leqq 1/12$. The weighting function is as follows.

$$f(p) = \begin{cases} 3/8 & 3/8 - \delta/2 < p \leqq 1 \quad \text{B-piece} \\ 1/4 & 1/3 < p \leqq 3/8 - \delta/2 \quad \text{B1-piece} \\ 1/4 & 1/4 + \delta \leqq p \leqq 1/3 \quad \text{C-piece} \end{cases}$$

Note that the FFD packing must have at least one B-piece in it. For if not, then each bin in the FFD packing would have three pieces in it. Since no bin can contain more than

three pieces in *any* packing, we have the contradiction that, in order to pack p_v, an optimum packing *must* have a bin with four pieces.

The next step of the proof determines the parameter values α, β, and w_d. It can be shown that no bin weight in an optimum packing exceeds $7/8$, and that the weight of each regular bin in the FFD packing is at least $3/4$. The maximum deficit, w_d, is obtained from the deficit B-bin with configuration B B1; hence $w_d = 1/8$.

Using the parameter values $\alpha = 7/8$, $\beta = 3/4$, and $w_d = 1/8$, along with $f(p_u) = 3/8$, $f(p_v) = 1/4$, and $g(m) = 2m$ (from Claim 1), we obtain (6) immediately. □

We note that the proofs of cases C1, D1, E1, and E2 are similar to the proof of case B1.

CASE B2. $p_u \in (1/3, 1/2]$ *and* $p_v \in (1/5, 1/4]$.

Proof. Let $p_v = 1/5 + \delta$, $0 < \delta \leqq 1/20$. The weighting function is given as follows.

$$f(p) = \begin{cases} 2/5 & 2/5 - \delta/2 < p \leqq 1 \quad \text{B-piece} \\ 3/10 & 1/3 < p \leqq 2/5 - \delta/2 \quad \text{B1-piece} \\ 4/15 & 4/15 - \delta/3 < p \leqq 1/3 \quad \text{C-piece} \\ 1/5 & 1/4 < p \leqq 4/15 - \delta/3 \quad \text{C1-piece} \\ 1/5 & 1/5 + \delta \leqq p \leqq 1/4 \quad \text{D-piece} \end{cases}$$

The next step of the proof determines an upper bound for w_0 and a lower bound for w_F. First, it can be shown that the weight of each regular bin in the FFD packing is at least $4/5$, and that the maximum global deficit is $4/15$ (which is obtained from the deficit B, B1, and C-bins with respective configurations B B1, B1 C D, and C C1 C1). Next, it can be verified that in an optimum packing any bin containing zero, one, and two B-piece(s) must have a weight not exceeding $14/15$, $29/30$, and $4/5$, respectively. Note that we have introduced three different bin-weight bounds which depend on the number of B-pieces contained in the bin. These bounds are necessary here since we need to refine w_0 using different techniques in each of the following subcases. The number of B-pieces in the FFD packing will determine our calculation of w_0 in the subcases identified below.

CASE B2.1. *There is at least one B-piece in the FFD packing.*

Proof. Let y denote the number of B-bins in the FFD packing. We have $n_{FD} \geqq 2y + 3(m - y) = 3m - y$. Thus, if $y \leqq 13m/20 + 1/2$ then $n_{FD} \geqq 77m/30 - 1/2$. Using $\alpha = 29/30$, $\beta = 4/5$, $w_d = 4/15$, $f(p_u) = 2/5$, $f(p_v) = 1/5$, and $g(m) = 77m/30 - 1/2$, we find that (6) holds.

But if $y > 13m/30 + 1/2$ we use $g(m) = 2m$ and refine the calculation of w_0 as follows. Let y_0 be the number of bins with two B-pieces in an optimum packing, and let z (respectively z_0) denote the number of B-pieces in the FFD (respectively an optimum) packing. Routinely, we have $z \geqq 2y - 1 > 2(13m/30 + 1/2) - 1 = 13m/15$. Since the theorem is assumed false we have as before $u - 1 > n_{FD}/6 + 2 \geqq 2m/6 + 2 > m/3$. Therefore, $z_0 > 13m/15 + m/3 = 6m/5$. Thus, a lower bound on the number of bins with two B-pieces in an optimum packing is obviously $y_0 > 6m/5 - m = m/5$. It follows that $w_0 = 4y_0/5 + 29(m - y_0)/30 = 29m/30 - y_0/6$ and hence $w_0 \leqq 14m/15$.

Thus, we may use our standard argument with $\alpha = 14/15$. Along with $\beta = 4/5$, $w_d = 4/15$, $g(m) = 2m$, $f(p_u) = 2/5$, and $f(p_v) = 1/5$ we see that (6) is satisfied.

CASE B2.2. *There are no B-pieces in the FFD packing.*

Proof. Immediately, we see that $n_{FD} \geqq 3m$, for there are at least three pieces per bin. Now let y' and y'' denote the numbers of B-pieces and B1-pieces, respectively, in an optimum packing but not the FFD packing. Clearly, $u - 1 = y' + y''$. Using the bin-weight bounds we find

$$w_0 \leqq 29y'/30 + 14(m - y')/15 = 14m/15 + y'/30. \tag{7}$$

Using $w_F + w_u + f(p_v) = w_0$, where w_u is the cumulative weight of the pieces $p_1, \cdots, p_{u-1}$, we obtain

$$w_0 \geqq \beta m - w_d + 2y'/5 + 3y''/10 + 1/5. \tag{8}$$

Since $\beta = 4/5$, and since $w_d = 1/6$ when there are no B-bins, we get from (7) and (8)

$$14m/15 + y'/30 \geqq 4m/5 - 1/6 + 2y'/5 + 3(u - y' - 1)/10 + 1/5,$$

from which we derive

$$u \leqq 4m/9 - 2y'/9 + 8/9.$$

Substituting $n_{\mathrm{FD}} \geqq 3m$ we obtain the desired contradiction

$$u \leqq 4n_{\mathrm{FD}}/27 - 2y'/9 + 8/9 < n_{\mathrm{FD}}/6 + 3. \qquad \square$$

The analyses of cases, B3, B4, B5, C2, and C3 are similar to case B2.

CASE B6. $p_u \in (1/3, 1/2]$ *and* $p_u \in (1/9, 1/8]$.

Proof. Let $p_v = 1/9 + \delta$, $0 < \delta \leqq 1/72$. The weighting function is given as follows.

$$f(p) = \begin{cases} 4/9 & 4/9 - \delta/2 < p \leqq 1 & \text{B-piece} \\ 7/18 & 8/21 - 3\delta/7 < p \leqq 4/9 - \delta/2 & \text{B1-piece} \\ 8/21 & 47/126 - \delta/2 < p \leqq 8/21 - 3\delta/7 & \text{B2-piece} \\ 41/108 & 10/27 - 5\delta/12 < p \leqq 47/126 - \delta/2 & \text{B3-piece} \\ 10/27 & 13/36 - \delta/2 < p \leqq 10/27 - 5\delta/12 & \text{B4-piece} \\ 11/30 & 16/45 - 2\delta/5 < p \leqq 13/36 - \delta/2 & \text{B5-piece} \\ 16/45 & 31/90 - \delta/2 < p \leqq 16/45 - 2\delta/5 & \text{B6-piece} \\ 25/72 & 1/3 < p \leqq 31/90 - \delta/2 & \text{B7-piece} \\ 8/27 & 8/27 - \delta/3 < p \leqq 1/3 & \text{C-piece} \\ 7/27 & 1/4 < p \leqq 8/27 - \delta/3 & \text{C1-piece} \\ 2/9 & 2/9 - \delta/4 < p \leqq 1/4 & \text{D-piece} \\ 7/36 & 1/5 < p \leqq 2/9 - \delta/4 & \text{D1-piece} \\ 8/45 & 8/45 - \delta/5 < p \leqq 1/5 & \text{E-piece} \\ 7/45 & 1/6 < p \leqq 8/45 - \delta/5 & \text{E1-piece} \\ 4/27 & 4/27 - \delta/6 < p \leqq 1/6 & \text{F-piece} \\ 7/54 & 1/7 < p \leqq 4/27 - \delta/6 & \text{F1-piece} \\ 8/63 & 8/63 - \delta/7 < p \leqq 1/7 & \text{G-piece} \\ 1/9 & 1/8 < p \leqq 8/63 - \delta/7 & \text{G1-piece} \\ 1/9 & 1/9 + \delta \leqq p \leqq 1/8 & \text{H-piece} \end{cases}$$

It can be shown as before that no bin weight in an optimum packing exceeds 28/27, and that the weight of each regular bin in the FFD packing is at least 8/9. The maximum global deficit is 236/315; it is obtained from the deficit B, B1, B2, B3, B4, B5, B6, B7, C, C1, D, D1, E, E1, F, F1, and G-bins with respective configurations B B1, B1 B2 H, B2 B3 G, B3 B4 F1, B4 B5 F, B5 B6 E1, B6 B7 E, B7 C D1, C C1 C1, C1 D D H, D D1 D1 D1, D1 E E E H, E E1 E1 E1 E1, E1 F F F F H, F F1 F1 F1 F1 F1, F1 G G G G G H, and G G1 G1 G1 G1 G1 G1. We consider the following two subcases.

CASE B6.1. *There is at least one B-piece in the FFD packing.*

Proof. In this case we have $g(m) = 2m$. Using $\alpha = 28/27$, $\beta = 8/9$, $w_d = 236/315$, $f(p_u) = 4/9$, and $f(p_v) = 1/9$ we find that (6) holds.

CASE B6.2. *There are no B-pieces in the FFD packing.*

Proof. With no B-Pieces we must have $n_{FD} \geqq 3m$ and a maximal global deficit $w_d = 437/630$. Using these together with $\alpha = 28/27$, $\beta = 8/9$, $f(p_u) \geqq 25/72$, and $f(p_v) = 1/9$ in (6) gives us the desired contradiction. □

We conclude our proof of Theorem 3 by noting that the analyses of cases C4, D2 and D3 are similar to case B6. □

Concluding remarks. We should emphasize that the bound of 8/7 is only conjectured to be best asymptotically; i.e., as $n_0(L)$ tends to infinity. We make this point in view of "edge" effects resulting from small values of m. For example, with $m = 2$, a suitable $\varepsilon > 0$, and $L = (1/3+2\varepsilon, 1/3, 1/3, 1/3-\varepsilon, 1/3-\varepsilon)$, we obtain a ratio $n_0/n_{FD^*} = 6/5$.

We have shown that at most m largest pieces need be discarded before an FFD fit is assured. Thus, one might reasonably question whether it is not better to organize a binary search over this range, rather than a linear one. This would reduce the complexity of FFD* to $O(n \log n + n(\log m)^2)$. However, a difficulty arises in proving bounds for this variation (which of course can never produce a packing with more pieces than the one produced by a linear search). Specifically, we have as yet been unable to prove that if FFD fails when the x largest pieces are discarded, then it must also fail when the $x-1$ largest pieces are discarded. Such FFD anomalies occurring when one piece is removed are known to exist [3], but anomalies which are restricted to the removal of a *largest* piece have not been found.

An alternative algorithm for our problem is the iterated largest-first algorithm which is organized like FFD*, but which applies to the largest-first (level) [3] algorithm in each iteration instead of the FFD algorithm. However, it is not difficult to find examples that verify an inferior worst-case performance.

The problem we have studied is just one of a number of bin-packing type problems that occur in operating system design. These problems differ in the measure of performance used, but they all reflect the desire for maximum storage of information with minimum access times, and a maximum throughput of jobs with a minimum average turnaround time. Our analysis has in itself been of interest in that yet another application of a "weighting function" approach has been found successful.

REFERENCES

[1] 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*, this Journal, 3 (1974), pp. 299–326.

[2] D. S. JOHNSON, *Fast algorithms for bin-packing*, J. Comput. System Sci., 8 (1974), pp. 272–314.

[3] R. L. GRAHAM, *Bounds on the performance of scheduling algorithms*, Computer and Job-Shop Scheduling Theory, E. G. Coffman, Jr., ed., John Wiley, New York, 1976.

[4] E. G. COFFMAN, JR., M. R. GAREY AND D. S. JOHNSON, *An application of bin-packing to multiprocessor scheduling*, this Journal, 7 (1978), pp. 1–17.

[5] E. G. COFFMAN, JR., J. Y-T. LEUNG AND D. TING, *Bin-packing: Maximizing the number of pieces packed*, Acta Informat., 9 (1978), pp. 263–271.

[6] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computation, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972.

[7] J. D. ULLMAN, *Complexity of sequencing problems*, Computer and Job-Shop Scheduling Theory, E. G. Coffman, Jr., ed., John Wiley, New York, 1976.

[8] J. Y-T. LEUNG, *Efficient algorithms for storage allocation problems*, Ph.D. Thesis, Computer Science Dept., Pennsylvania State Univ., Middleton, PA, 1977.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0009 $01.00/0

EQUIVALENCES AMONG RELATIONAL EXPRESSIONS*

A. V. AHO†, Y. SAGIV‡ AND J. D. ULLMAN¶

Abstract. Many database queries can be formulated in terms of expressions whose operands represent tables of information (relations) and whose operators are the relational operations select, project, and join. This paper studies the equivalence problem for these relational expressions, with expression optimization in mind. A matrix, called a tableau, is proposed as a natural representative for the value of an expression. It is shown how tableaux can be made to reflect functional dependencies among attributes. A polynomial time algorithm is presented for the equivalence of tableaux that correspond to an important subset of expressions, although the equivalence problem is shown to be NP-complete under slightly more general circumstances.

1. Introduction. Codd's relational algebra is a high-level query language in which questions can be posed simply and succinctly [9], [11]. Concepts from relational algebra have been incorporated into the design of several new database query languages [13].

Expressions in relational algebra manipulate tables of information (called relations) by means of high-level operations such as select, project, and join. A disadvantage of relational algebra as a query language is that the efficiency with which a query can be answered varies considerably with the manner in which the query is formulated. The very flexibility of the language makes it easy to express queries that are hard to implement or for which efficient implementations are hard to find. Consequently, a number of papers [17], [19], [20], [21], [23], [25] have considered transformations that "optimize" relational queries. Like most work in code "optimization," however, these transformations improve expressions under some cost criterion, but do not claim to produce an equivalent expression of least cost. Chandra and Merlin [8] show how to perform true optimization on a large class of queries, but their algorithm is exponential in the size of the query.

In this paper we consider the inherent computational complexity of determining whether two queries are equivalent, with an eye toward globally optimizing queries under a variety of cost measures. We restrict the relational algebra to include only the three operators: select, project, and join. We show that the optimization problem for even this restricted subset of relational algebra is computationally difficult (NP-complete).

We introduce tableaux, two-dimensional representations of queries. Tableaux may be viewed as a form of Zloof's "Query-by-Example" language [27] and also as a stylized notation for a subset of Chandra and Merlin's "conjunctive queries" [8]. The tableau immediately removes one objection (see [24], e.g.) to relational algebra as a query language, since tableaux are nonprocedural representations of queries in exactly the sense that relational calculus [9], [11] is nonprocedural.

We reduce the equivalence problem for queries to the analogous problem for tableaux. One advantage of the tableau approach is that it allows us to deal with functional dependencies mechanically, a feature not possessed by more direct techniques. We then show how to minimize the number of rows in a tableau, an operation that corresponds to minimizing the number of joins needed to evaluate a query. Since join is typically a very expensive operator to implement, this approach is a good "first crack" at reducing the cost of evaluating a query. Row minimization also serves to eliminate common subexpressions from a query.

* Received by the editors March 23, 1978. This work was supported in part by the National Science Foundation under Grant MCS-76-15255.

† Bell Laboratories, Murray Hill, New Jersey, 07974.

‡ Princeton University, Princeton, New Jersey. Now at Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 6180.

¶ Princeton University, Princeton, New Jersey 08540.

Next we introduce "simple tableaux," a subclass of tableau for which we can show the equivalence and optimization problems that were computationally difficult for general tableaux are now tractable. Although the set of queries having simple tableaux is a proper subset of the set of relational expressions, we nevertheless feel that most practical queries that contain only selects, projects, and joins can be represented by simple tableaux. We conclude the paper with a discussion of some remaining problems.

2. Basic definitions. In this section we define our restricted subclass of relational expressions. We also show that there are several possible definitions of expression equivalence.

2.1. Relation schemes and relations. We assume the data are stored in a set of two-dimensional tables called *relations*. The columns of a table are labeled by distinct *attributes* and the entries in each column are drawn from a fixed *domain* for that column's attribute. For the purposes of this paper we assume the ordering of the attributes of a table is unimportant. Each row of a table is a mapping from the table's attributes to their respective domains. A row is often called a *tuple* or *record*. If r is a relation that is defined on a set of attributes that includes A, and if μ is a tuple of r, then $\mu(A)$ is the value of the A-component of μ.

A *relation scheme* is the set of attributes labeling the columns of a table. When there is no ambiguity, we shall use the relation scheme itself as the name of the table. A relation is just the "current value" of a relation scheme. The relation is said to be *defined on* the set of attributes of the relation scheme.

Example 1. Suppose we have the two relation schemes PAT and PR, representing two tables, one with columns P, A, and T, the other with columns P and R. (P stands for Paper-number, A for Author, T for Title, R for Referee.) Figure 1 shows two relations that might be current values of these relation schemes.

P	A	T
1	Black	*All About Horses*
2	Brown	*All About Dogs*
3	Blue	*All About Cats*

P	R
1	Turtle
1	Snake
2	Turtle
3	Ox

FIG. 1. *Two tables.*

2.2. Dependencies. Often the values of entries in relations satisfy certain constraints. Functional [4], [9] and multivalued [7], [14], [15], [26] dependencies are examples of such constraints. In this paper we assume all dependencies are functional. Our theory carries over to multivalued dependencies as well, although an efficient equivalence test in that case is elusive.

A *functional dependency* is a statement $X \to Y$, where X and Y are sets of attributes. A relation r satisfies this functional dependency if and only if for all μ and ν in r the following condition holds: If $\mu(A) = \nu(A)$ for all A in X, then $\mu(B) = \nu(B)$ for all B in Y. That is, if two rows of r agree in the columns for X, then they must agree in the columns for Y. Note that if r satisfies a given set of dependencies, then it may also satisfy additional dependencies, e.g., if r satisfies $A \to B$ and $B \to C$, it also satisfies $A \to C$.

For a set of attributes X, we define X^*, the *closure* of X, as follows:

(1) $X \subseteq X^*$.

(2) If $Y \subseteq X^*$, and $Y \to Z$ is a given functional dependency, then $Z \subseteq X^*$.

(3) No attribute is in X^* unless it so follows from (1) and (2).

We write $X \overset{*}{\to} Y$ if $Y \subseteq X^*$. Essentially, $X \overset{*}{\to} Y$ means that the functional dependency $X \to Y$ is in, or can be derived from, the given set of dependencies. Two sets of dependencies are *equivalent* if, for all X, the set X^* is the same under either set of dependencies. It is well known that any set of dependencies is equivalent to a set in which each right side consists of a single attribute, and we henceforth assume all sets of functional dependencies are of this form.

2.3. Restricted relational expressions. In this paper we shall consider relational expressions in which the only operators are select, project, and (natural) join. The operands are relation schemes. The operators are defined as follows.

(1) *Select.* Let r be a relation on a set of attributes X, A an attribute in X, and c a value from the domain of A. Then the *selection* $A = c$, written $\sigma_{A=c}(r)$, is the set

$$\{\mu \mid \mu \text{ is in } r \text{ and } \mu(A) = c\}$$

that is, the subset of r having value c for attribute A.

(2) *Project.* Let r be a relation on a set of attributes X. Let Y be a subset of X. We define $\pi_Y(r)$, the *projection* of r onto Y, to be the relation obtained by removing all the components of the tuples of r that do not belong to Y and identifying common tuples. That is, $\pi_Y(r) = \{\nu \mid \nu$ has components for all and only the attributes of Y, and for some μ in r, $\nu(A) = \mu(A)$ for all A in $Y\}$.

For example, if r is the second relation of Fig. 1, then $\pi_P(r) = \{1, 2, 3\}$.

(3) *Join.* The join operator, denoted by $\bowtie$, permits two relations to be combined into a single relation whose attributes are the union of the attributes of the two argument relations. Let R_1 and R_2 be two relation schemes with current values r_1 and r_2. Then

$$r_1 \bowtie r_2 = \{\mu \mid \mu \text{ is a tuple with components for all and only the attributes in } R_1 \cup R_2, \text{ and there exist tuples } \nu_1 \text{ in } r_1 \text{ and } \nu_2 \text{ in } r_2, \text{ such that } \nu_1(A) = \mu(A) \text{ for all } A \text{ in } R_1 \text{ and } \nu_2(A) = \mu(A) \text{ for all } A \text{ in } R_2\}.$$

Example 2. If r_1 and r_2 are the two relations of Fig. 1, then $r_1 \bowtie r_2$ is the relation

P	A	T	R
1	Black	*All About Horses*	Turtle
1	Black	*All About Horses*	Snake
2	Brown	*All About Dogs*	Turtle
3	Blue	*All About Cats*	Ox

□

Even with these three simple operators we can pose a variety of interesting queries. Here are two examples that refer to the database in Fig. 1.

(1) The query "List the author of the paper *All About Dogs*" can be represented by the expression $\pi_A(\sigma_{T=\text{"All About Dogs"}}(PAT))$.

(2) "List the authors and titles of all papers refereed by Turtle" becomes $\pi_{AT}(\sigma_{R=\text{"Turtle"}}(PAT \bowtie PR))$.

With these operators we can also define Cartesian product (if in a join the sets of attributes for the two relations are made disjoint) and intersection (which is a special case of the natural join where the two relations are over the same set of attributes). The relational algebra of Codd [9], [11] includes other operators, and to make a "complete" set we would need to add union, set difference and selections involving arithmetic comparisons between two components of a tuple.

2.4. Expression values. The notion that a relation is the "value" of a relation scheme can be generalized to expressions. Let E be an expression with operand relation schemes $R_1, R_2, \cdots, R_k$. An *assignment* associates a relation r_i with each relation scheme R_i, $1 \leqq i \leqq k$. Given an assignment α of relations to relation schemes, the value of E, denoted $\nu_\alpha(E)$ or $\nu(E)$ if α is understood, is computed by applying operators to operands in the following natural way.

(1) If E is a single relation scheme R_i, then $\nu(E) = r_i$.

(2) (a) If $E = \sigma_{A=c}(E_1)$, then $\nu(E) = \sigma_{A=c}(\nu(E_1))$.

(b) If $E = \pi_X(E_1)$, then $\nu(E) = \pi_X(\nu(E_1))$.

(c) If $E = E_1 \bowtie E_2$, then $\nu(E) = \nu(E_1) \bowtie \nu(E_2)$.

We may also regard expression E as a function, mapping assignments of values for its operands to values for the expression. That is, if E is an expression with operands $R_1, R_2, \cdots, R_k$, we define $V(E)$ to be the mapping that sends each assignment α of relations $r_1, r_2, \cdots, r_k$ for $R_1, R_2, \cdots, R_k$ to $\nu_\alpha(E)$. Intuitively, two expressions E_1 and E_2 are equivalent if $V(E_1)$ and $V(E_2)$ are the same mapping. However, we may not wish to allow completely arbitrary sets of R_i's and r_i's. We have therefore isolated three distinct notions of equivalence, which we shall discuss in turn.

2.5. Algebraic equivalence. If we do not fix the R_i's, that is, allow each relation scheme to be a variable set of attributes, we obtain a notion of algebraic equivalence. For example, the commutative law of joins $R \bowtie S = S \bowtie R$ is true independent of R and S. It is not clear how the select operator can be brought into this framework, although the project operator π_X can be covered if we regard X as a variable set of attributes. We shall not discuss algebraic equivalence further in this paper.

2.6. Strong equivalence. We may, instead, regard each $R_1, R_2, \cdots, R_k$ as a relation scheme with a fixed set of attributes, and call two expressions E_1 and E_2 *strongly equivalent* if $V(E_1) = V(E_2)$ under this assumption. That is, we regard E_1 and E_2 as equivalent if they define the same mapping. Strong equivalence appears to be the notion underlying previous attempts at expression optimization, and is probably the notion with which most people would feel secure.

2.7. Weak equivalence. A variety of papers such as [1], [4], [7] have viewed a database as though a single universal relation exists at each instant of time. In this framework we restrict assignments of values to relation schemes $R_1, R_2, \cdots, R_k$ by insisting that there be some relation I on the set of attributes $\bigcup_{i=1}^{k} R_i$ such that the value r_i assigned to R_i is $\pi_{R_i}(I)$. We call such a relation I an *instance of the universe*, or just an *instance*. If $\nu_\alpha(E_1) = \nu_\alpha(E_2)$ for all assignments α obtained in this way from an instance, then we say E_1 and E_2 are *weakly equivalent*, and write $E_1 \equiv E_2$.

The notion of weak equivalence is also well motivated. It is essential when we deal with equivalences between expressions whose operands are different relation schemes. For example, it allows the treatment of lossless joins, as in [1], [22], [26], and it is the notion of equivalence underlying the normal form decompositions of [9], [10].

We shall deal with weak equivalence, which we hereafter call simply *equivalence*, almost exclusively in this paper, ending with a demonstration of how our ideas carry over to strong equivalence as well. The motivation for so doing is not our belief that strong equivalence is an inferior notion; rather our ideas are more simply expressed when (weak) equivalence is considered. In particular, we may take advantage of the presence of universal instances to regard the value of an expression as a mapping from instances to relations. That is, if I is an instance, $\nu_I(E)$ is the value of expression E when each argument R_i of E is replaced by $\pi_{R_i}(I)$.

Example 3. Consider the expression $E = \pi_{AB}(AB \bowtie BC)$. Here, A, B and C are attributes, and relation schemes are denoted by strings of attributes, i.e., AB stands for $\{A, B\}$. If there is a univeral instance I over attributes A, B and C, in that order, then the value r_{AB} for relation scheme AB is

$$\{ab | \text{for some } c, abc \text{ is in } I\}$$

and the value of r_{BC} for BC is

$$\{bc | \text{for some } a, abc \text{ is in } I\}.$$

The value of $AB \bowtie BC$ is

$$r_{AB} \bowtie r_{BC} = \{abc | \text{for some } a' \text{ and } c', abc' \text{ and } a'bc \text{ are in } I\}.$$

Finally, the value of E is

(*) $$\{ab | \text{for some } a', c' \text{ and } c, abc' \text{ and } a'bc \text{ are in } I\}$$

which is just

$$\{ab | \text{for some } c, abc \text{ is in } I\}$$

as we may take $a = a'$ and $c = c'$ in (1). Thus, E is equivalent to the expression consisting of the single relation scheme AB.

On the other hand, consider strong rather than weak equivalence. Then r_{AB} and r_{BC} can be independently chosen relations. The value of E is

$$\{ab | \text{for some } c, ab \text{ is in } r_{AB} \text{ and } bc \text{ is in } r_{BC}\}$$

which is not necessarily equal to r_{AB}. For example, if $r_{AB} = \{ab\}$ and $r_{BC} = \varnothing$, then the value of E is $\varnothing$, not $\{ab\}$. Note that these values for r_{AB} and r_{BC} cannot come from one instance.

2.8. The effect of data dependencies. Constraints, such as functional dependencies, also affect the requirements for equivalence of expressions. For example, functional dependencies may be applied to instances, and in the presence of a set of functional dependencies we say that $E_1 \equiv E_2$ if $\nu_I(E_1) = \nu_I(E_2)$ for all instances I that satisfy the functional dependencies. Similarly, functional dependencies may apply to relations, and we define E_1 to be strongly equivalent to E_2 in the presence of functional dependencies, if $\nu_\alpha(E_1) = \nu_\alpha(E_2)$ for all assignments α of relations r_i to arguments R_i such that the r_i's satisfy the dependencies.

3. Tableaux. In this section we show how to represent the mappings defined by relational expressions by specialized matrices called "tableaux". Tableaux are similar to the tabular queries of Query-by-Example [27] and the conjunctive queries of Chandra and Merlin [8]. We shall see that for every query in our query language there is a tableau with the same value, but unfortunately, the correspondence is not exact. There are tableaux that do not correspond to any expression over the operators we discuss (or, to our knowledge, over any other set of operators that have appeared in the literature).

3.1. Definition of a tableau. A tableau is a matrix in which the columns correspond to the attributes of the universe in a fixed order. The first row of the matrix is called the *summary* of the tableau. The remaining rows are to be exclusively called *rows.*

The general idea is that a tableau is a shorthand for an explicit set description, such as (*) above, used to define the value of an expression. The summary represents what

appears to the left of the vertical bar, e.g., ab in (*) The rows represent the tuples required to be in I, such as abc' and $a'bc$ in (*)

To simplify later discussion we shall adopt the following conventions regarding tableaux. The symbols appearing in a tableau are chosen from:

(1) Distinguished variables, for which we use a's, possibly with subscripts. These correspond to the symbols to the left of the bar, as a and b in (*).
(2) Nondistinguished variables, for which we generally use b's. These are the other symbols appearing in set formers, such as a' and c' in (*).
(3) Constants, for which we use c's or nonnegative integers.
(4) Blank.

The summary of a tableau may contain only distinguished variables, constants, and blanks. The rows of a tableau may contain variables (distinguished and nondistinguished) and constants. We also require that the same variable not appear in two different columns of a tableau, and that a distinguished variable not appear in a column unless it also appears in the summary of that column.

Let T be a tableau and let S be the set of all symbols appearing in T (i.e., variables and constants). A *valuation* ρ for T associates with each symbol of S a constant, such that if c is a constant in S, then $\rho(c) = c$. We extend ρ to the summary and rows of T as follows. Let w_0 be the summary of T, and $w_1, w_2, \cdots, w_n$ the rows. Then $\rho(w_i)$ is the tuple obtained by substituting $\rho(v)$ for every variable v that appears in w_i.

A tableau defines a mapping from instances to relations on a certain subset of attributes, called the *target relation scheme*, in the following way. If T is a tableau and I an instance, then $T(I)$ is the relation on the attributes whose columns are nonblank in the summary, such that

$$T(I) = \{\rho(w_0) | \text{for some valuation } \rho \text{ we have } \rho(w_i) \text{ in } I \text{ for } 1 \leqq i \leqq n\}.$$

Example 4. Let T be the tableau

A	B	C
a_1	a_2	
a_1	b_1	b_3
b_2	a_2	1
b_2	b_1	b_4

We conventionally show the summary first, with a line below it. We can interpret this tableau as defining the following relation on AB

$$T(I) = \{a_1a_2 | (\exists b_1)(\exists b_2)(\exists b_3)(\exists b_4) \text{ such that } a_1b_1b_3 \text{ is in } I \text{ and } b_2a_21 \text{ is in } I \text{ and } b_2b_1b_4 \text{ is in } I\}$$

where I is any instance. For example, suppose I is the instance $\{111, 222, 121\}$.

Consider the valuation ρ which assigns 1 to all the variables. Under this valuation, the three rows of T each become 111, which is a member of I. Therefore, $\rho(a_1a_2) = 11$ is in $T(I)$.

If ρ assigns 2 to b_1 and a_2, and 1 to the other variables, all rows become 121, so $\rho(a_1a_2) = 12$ is in $T(I)$.

If ρ assigns 2 to a_1, b_1 and b_3, and 1 to the other variables, then $\rho(a_1b_1b_3) = 222$ is in I, $\rho(b_2a_21) = 111$ is in I, and $\rho(b_2b_1b_4) = 121$ is in I, so $\rho(a_1a_2) = 21$ is in $T(I)$.

Finally, if ρ assigns 1 to b_2 and b_4, and 2 to the other variables, then we see that 22 is in $T(I)$. Thus, $T(I) = \{11, 12, 21, 22\}$. □

Conventionally, we also regard $\varnothing$ as a tableau. This tableau represents the function that maps every instance to the empty relation.

Tableaux are closely related to the conjunctive queries of [8]. The significant differences between tableaux and conjunctive queries are that

(1) tableaux permit constants in the summary,
(2) columns of a tableau are associated with attributes, and
(3) tableaux do not permit symbols appearing in two different columns.

Condition (1) is needed to handle the select operator; condition (2) is required that we may talk about dependencies and their effect on equivalence of expressions. Condition (3) is assumed because it enables us to show that even restricted subsets of conjunctive queries have hard optimization problems, and, more importantly, it enables us to isolate a large subset of tableaux for which optimization is relatively easy.

3.2. Equivalence of tableaux. Two tableaux T_1 and T_2 are *equivalent*, written $T_1 \equiv T_2$, if for all I, $T_1(I) = T_2(I)$. We say that T_1 is *contained in* T_2, written $T_1 \subseteq T_2$, if for all I, $T_1(I) \subseteq T_2(I)$. Note that a necessary, but not sufficient, condition for both $T_1 \equiv T_2$ and $T_1 \subseteq T_2$ is that the relations defined by T_1 and T_2 have the same target relation scheme.

As we shall see, the questions of equivalence and containment of tableaux are in the general case hard combinatorial problems. We can, however, state a basic and not unexpected result, namely that consistent renaming of variables does not change the value of a tableau, thus providing many obvious equivalences.

LEMMA 1. *Let T be a tableau and ψ a one-to-one correspondence that maps distinguished variables to distinguished variables, nondistinguished variables to nondistinguished variables, and constants to constants. If we construct a tableau T' from T by simultaneously substituting $\psi(v)$ for every occurrence of symbol v in T, then $T \equiv T'$.*

Proof. This result follows immediately from the definitions. □

3.3. Representation of expressions by tableaux. In this section we show how to construct a tableaux to represent any expression over the operators select, project, and join. The construction proceeds inductively by first building tableaux for the individual operands of an expression, and then combining these tableaux to form tableaux for larger and larger subexpressions, until a tableau for the entire expression is found. The rules for building a tableaux T for an expression E are:

(1) If E is a single relation scheme R, then the tableau T for E has one row and a summary such that:
  (i) If A is an attribute in R, then in the column for A, tableau T has the same distinguished variable in the summary and row.
  (ii) If A is not in R, then its column has a blank in the summary and a nondistinguished variable in the row.

(2a) Suppose E of the form $\sigma_{A=c}(E_1)$, and we have constructed T_1, the tableau for E_1.
  (i) If the summary for T_1 has blank in the column for A, then $T = \varnothing$.
  (ii) If there is a constant $c' \neq c$ in the summary column for A, then $T = \varnothing$. If $c = c'$, then $T = T_1$.
  (iii) If T_1 has a distinguished variable a in the summary column for A, the tableau T for E is constructed by replacing a by c whenever it appears in T_1.

(2b) Suppose E is of the form $\pi_X(E_1)$, and T_1 is the tableau for E_1. The tableau T for E is constructed by replacing nonblank symbols by blanks in the summary of T_1 for those columns whose attributes are not in X. Distinguished variables in those columns become nondistinguished.

(2c) Suppose E is of the form $E_1 \bowtie E_2$ and T_1 and T_2 are the tableaux for E_1 and E_2, respectively. Let S_1 and S_2 be the symbols of T_1 and T_2, respectively. By Lemma 1, we may take S_1 and S_2 to have disjoint sets of nondistinguished variables, but identical distinguished variables in corresponding columns.

(i) If T_1 and T_2 have some column in which their summaries have distinct constants, then $T = \varnothing$.

(ii) If no corresponding positions in the summaries have distinct constants, the rows of the tableau T for E consist of the union of all the rows of T_1 and T_2. The summary of T has in a given column

(a) The constant c if one or both of T_1 and T_2 have c in that column's summary. In this case we also replace any distinguished variable in that column by c.

(b) The distinguished variable a if (a) does not apply, but one or both of T_1 and T_2 have a in that column's summary.

(c) Blank, otherwise.

THEOREM 1. *The rules above construct for any restricted relational expression E a tableau T such that for all instances I, $\nu_I(E) = T(I)$.*

Proof. The proof is an induction on the number of operators in E.

Basis. Rule (1). If there are no operators in E, then E is a single relation scheme R, and rule (1) clearly constructs the appropriate tableau T.

Induction. Rule (2a). $E = \sigma_{A=c}(E_1)$. Let T_1 be the tableau for E_1.

(i) If the summary for T_1 has blank in the column for A, then the expression E has no meaning and $\varnothing$ is the correct tableau for E.

(ii) If there is a constant $c' \neq c$ in the summary column for A, then for any I, $\nu_1(E_1)$ has only tuples with c' in the component for A, and $\nu_I(E)$ is empty. Again, $\varnothing$ is the correct tableau for E. If $c = c'$, then T_1 is the correct tableau for E.

(iii) If T_1 has a distinguished variable a in the summary column for A, and we construct T for E by replacing a by c whenever it appears in T_1, then we claim that for all I, $T(I) = \sigma_{A=c}(T_1(I))$. In proof, suppose ρ is a map from the symbols of T_1 to a set of constants C. Let $w_0, w_1, \cdots, w_n$ be the summary and rows of T_1, and let $w_0', w_1', \cdots, w_n'$ be the same for T. That is, w_i' is w_i with a replaced by c if a appears in w_i. Then,

$$\begin{aligned} T(I) &= \{\rho(w_0') | \rho(w_i') \text{ is in } I \text{ for } 1 \leqq i \leqq n\} \\ &= \{\rho(w_0) | \rho(a) = c \text{ and } \rho(w_i) \text{ is in } I \text{ for } 1 \leqq i \leqq n\} \\ &= \sigma_{A=c}(\{\rho(w_0) | \rho(w_i) \text{ is in } I \text{ for } I \leqq i \leqq n\}) \\ &= \sigma_{A=c}(T_1(I)). \end{aligned}$$

The third line above follows from the fact that w_0 is known to have a in its column for A.

Rule (2b). $E = \pi_X(E_1)$. A proof of the correctness of this case is straightforward and is omitted.

Rule (2c). $E = E_1 \quad E_2$.

(i) If T_1 and T_2 have some column in which their summaries have distinct constants, then $V(E)$ maps all instances to $\varnothing$, so $\varnothing$ is the correct tableau for E.

(ii) If no corresponding positions in the summaries have distinct constants, we claim that $T(I) = T_1(I) \bowtie T_2(I)$ for all I. Let w_0 be the summary of T. Let x_j, $0 \leqq i \leqq n_1$, and y_i, $0 \leqq i \leqq n_2$, be the summaries and rows of T_1 and T_2, respectively. Then

$$T_1(I) = \{\rho_1(x_0) | \rho_1(x_i) \text{ is in } I \text{ for } 1 \leqq i \leqq n_1\},$$

$$T_2(I) = \{\rho_2(y_0) | \rho_2(y_i) \text{ is in } I \text{ for } I \leqq i \leqq n_2\},$$

$T_1(I) \bowtie T_2(I) = \{\rho(w_0) |$ for some ρ_1 and ρ_2, ρ agrees with ρ_1 and/or ρ_2, respectively, on the attributes with nonblank symbols in x_0 and y_0, respectively, $\rho_1(x_i)$ is in I for $1 \leqq i \leqq n_1$, and $\rho_2(y_i)$ is in I for $1 \leqq i \leqq n_2\}$.

As S_1 and S_2 have disjoint sets of nondistinguished variables, we may extend ρ to agree with ρ_1 and ρ_2 on all symbols present in T. Therefore

$$T_1(I) \bowtie T_2(I) = \{\rho(w_0) | \rho(x_i) \text{ is in } I \text{ for } 1 \leqq i \leqq n_1 \text{ and } \rho(y_i) \text{ is in } I \text{ for } I \leqq i \leqq n_2\}.$$ □

Example 5. Let A, B and C be the attributes, in that order, and suppose we have the expression $\pi_{AC}(\sigma_{B=0}(AB \bowtie BC))$. By Rule (1), the tableaux for AB and BC are

A	B	C
a_1	a_2	
a_1	a_2	b_1

and

A	B	C
	a_2	a_3
b_2	a_2	a_3

By Rule (2c), the tableau for $AB \bowtie BC$ is

A	B	C
a_1	a_2	a_3
a_1	a_2	b_1
b_2	a_2	a_3

By Rule (2a), the tableau for $\sigma_{B=0}(AB \bowtie BC)$ is

A	B	C
a_1	0	a_3
a_1	0	b_1
b_2	0	a_3

Finally, by Rule (2b), the tableau for $\pi_{AC}(\sigma_{B=0}(AB \bowtie BC))$ is

A	B	C
a_1		a_3
a_1	0	b_1
b_2	0	a_3

□

It is interesting to note that Chandra and Merlin [8] prove an analogue of Theorem 1 and also its converse, using select, project and join operations that are suitably generalized to take advantage of the fact that columns are not pinned down to particular attributes, and also an operator called restriction, that in effect identifies two distinguished variables of the same relation. However, in our model the converse to Theorem 1 is false. That is, there are tableaux that come from no expression, as the following example shows.

Example 6. The tableau

a_1	a_2
a_1	b_2
b_1	a_2
b_1	b_2

cannot be derived from any restricted relational expression. If there is such an expression, suppose that the last two rows come from the first two relations joined. The expression resulting from this join must later be joined with a relation from which the first row, a_1b_2, is derived. Since b_2 appears in rows 1 and 3, b_2 must have been distinguished at this later time, else the symbols in these positions could not be identified with one another. Since a_2 is currently distinguished, however, it must have been so when the last join was performed, and symbols b_2 and a_2 would not be distinct. A similar contradiction is obtained no matter which two rows we assume are grouped first.

In fact, even had we introduced a restriction operator, we could not produce the above tableau. In proof, note that if a tableau has a symbol appearing in two columns, the operations on tableaux corresponding to select, project and join preserve that property. Since the above tableau has no symbol in both columns, we know that restriction could be of no help in forming it. □

We know of no natural set of operators that characterizes tableaux exactly.

The construction rules above can also be used to define the operations select, project and join on tableaux. The result of applying any one of these operations to tableaux (not necessarily tableaux derived from expressions) is defined to be the tableau described in the rule for that operation.

4. Testing equivalence of tableaux. In this section we shall give a method for testing the equivalence of tableaux, thus providing an algorithm for testing the equivalence of expressions.

4.1. Homomorphisms. Chandra and Merlin [8] give a necessary and sufficient condition for the equivalence of conjunctive queries in terms of "homomorphisms," which are symbol-symbol mappings with certain properties. We shall prove the analogous result here for tableaux. We shall then prove a dual formulation of the equivalence test of [8] in terms of row-row mappings called "containment mappings."

Let T_1 and T_2 be two tableaux with sets of symbols S_1 and S_2. A *homomorphism* is a mapping $\psi : S_1 \to S_2$ such that:

(i) If c is a constant, then $\psi(c) = c$.

(ii) If a is distinguished, then $\psi(a)$ either is distinguished or is the constant appearing in the corresponding column of the summary of T_2.

(iii) If w is any row of T_1, then $\psi(w)$ is a row of T_2.

Then, intuitively, any time that we can map the rows of T_2 into elements of an instance I, the homomorphism ψ gives us a map from rows of T_1 into I as well. Thus, $T_2(I) \subseteq T_1(I)$ for all I, so $T_2 \subseteq T_1$.

The converse holds as well. If $T_2 \subseteq T_1$, then we can make the rows of T_2 be an instance I of the universe, by treating all symbols of S_2 as distinct constants. The fact that $T_2 \subseteq T_1$ implies that $T_2(I) \subseteq T_1(I)$. The fact that the summary of T_2, with blanks deleted, is in $T_2(I)$, and hence in T_1(I), implies that the homomorphism $\psi : S_1 \to S_2$ exists. We may formalize the above as follows.

THEOREM 2. *Let T_1 and T_2 be two tableaux with sets of symbols S_1 and S_2. $T_2 \subseteq T_1$ if and only if they have the same target relation scheme, and there is a homomorphism $\psi : S_1 \to S_2$.*

Proof [8] (*If*). Let I be an instance, and let $\rho : S_2 \to C$ be a valuation, where C is a set of constants, such that for each row w of T_2, $\rho(w)$ is an element of I. Then $\rho \cdot \psi : S_1 \to C$ is a valuation that sends each row of T_1 to an element of I, by condition (iii). By conditions (i) and (ii), if s_1 and s_2 are the summaries of T_1 and T_2, respectively, with blanks deleted, then $\rho(s_2) = \rho(\psi(s_1))$. Thus, any tuple in $T_2(I)$ is in $T_1(I)$, so $T_2 \subseteq T_1$.

(*Only if*). Let ρ be a one-to-one correspondence between the symbols of T_2 and some set of constants, and let I be the instance consisting of all the elements $\rho(w)$, for w a row of T_2. Then $\rho(s_2)$ is in $T_2(I)$, and since $T_2 \subseteq T_1$, it is also in $T_1(I)$. Thus, there is a homomorphism $\psi : S_1 \to S_2$ satisfying (i)–(iii) by the definition of the application of a tableau to an instance and the fact that ρ is one-to-one. □

4.2. Containment mappings. A containment mapping is a mapping from the rows of one tableau to another that preserves distinguished variables and constants and does not map any symbol to two different symbols. Formally, let T_1 and T_2 be tableaux, and let θ be a mapping from the rows of T_1 to the rows of T_2. We say θ is a *containment mapping* if:

(a) For each row i of T_1, if row i has a distinguished variable in some column A, then row $\theta(i)$ of T_2 has a distinguished variable or constant in column A.

(b) If row i of T_1 has a constant c in column A, then row $\theta(i)$ has c in column A.

(c) If rows i and j of T_1 have the same nondistinguished variable in column A, then rows $\theta(i)$ and $\theta(j)$ have the same symbol in that column. That symbol could be constant, distinguished, or nondistinguished. Also note that $\theta(i) = \theta(j)$ is possible.

We may prove the following analogue to Theorem 2.

THEOREM 3. *$T_2 \subseteq T_1$ if and only if they define the same target relation and there is a containment mapping θ from T_1 to T_2.*

Proof (*If*). Let $\psi : S_1 \to S_2$ be a symbol-symbol mapping such that if symbol d appears in column A of row r of T_1, and symbol d' appears in column A of row $\theta(r)$ of T_2, then $\psi(d) = d'$. The map ψ is consistent by condition (c). Conditions (i)–(iii) for ψ are immediate. That is, (a) implies (ii), (b) implies (i), and (iii) is implied by the definition of ψ from θ. Thus, ψ is a homormorphism, and by Theorem 2, $T_2 \subseteq T_1$.

(*Only if*). By Theorem 2, there is a homomorphism $\psi : S_1 \to S_2$ satisfying (i)–(iii). The existence of a map from the rows of T_1 to the rows of T_2 satisfying (c) follows from (iii); (i) and (ii) imply (a) and (b). □

As a containment mapping on rows induces a homomorphism satisfying (i)–(iii), we shall sometimes fail to distinguish a containment mapping from its corresponding homomorphism.

COROLLARY 1. *$T_1 \equiv T_2$ if and only if T_1 and T_2 have identical summaries up to renaming of distinguished variables, and containment mappings exist in both directions. In this case the possibility that row $\theta(i)$ in condition* (a) *has a constant can be ignored, since a constant cannot map back to a distinguished variable.*

Example 7. The expression $\pi_{AB}(AB \bowtie BC)$ of Example 3 has tableau

	A	B	C
	a_1	a_2	
$T_1 = w_1$	a_1	a_2	b_1
w_2	b_2	a_2	b_3

while the expression AB over set of attributes A, B and C has tableau

$$T_2 = \begin{array}{c|ccc} & A & B & C \\ \hline & a_1 & a_2 & \\ \hline w_3 & a_1 & a_2 & b_1 \end{array}.$$

In one direction, the map that sends both w_1 and w_2 to w_3 is a containment mapping. The induced homomorphism is:

in T_1	in T_2
a_1	a_1
a_2	a_2
b_1	b_1
b_2	a_1
b_3	b_1

In the opposite direction, we may map w_3 to w_1, showing the containment in the opposite direction as well. Thus AB and $\pi_{AB}(AB \bowtie BC)$ are equivalent.

For another example, let $E_1 = AB \bowtie AC \bowtie BC$ and $E_2 = ABC \bowtie \sigma_{C=0}(BC)$. The tableaux for E_1 and E_2 are, respectively,

$$T_1 = \begin{array}{c|ccc} & A & B & C \\ \hline & a_1 & a_2 & a_3 \\ \hline w_1 & a_1 & a_2 & b_1 \\ w_2 & a_1 & b_2 & a_3 \\ w_3 & b_3 & a_2 & a_3 \end{array} \qquad T_2 = \begin{array}{c|ccc} & A & B & C \\ \hline & a_1 & a_2 & 0 \\ \hline w_4 & a_1 & a_2 & 0 \\ w_5 & b_1 & a_2 & 0 \end{array}.$$

Then $T_2 \subseteq T_1$, since we may produce a containment mapping by sending w_1, w_2 and w_3 to w_4. We may alternatively map w_3 to w_5 if we like. However, in the opposite direction there is no containment mapping, since the constant 0 cannot map to a variable. Thus $T_1 \not\subseteq T_2$. To prove this we may make an instance I from the rows of T_1 by assigning, say, $1, 2, \cdots, 6$ to a_1, a_2, a_3, b_1, b_2 and b_3. Then $T_1(I)$ contains 123, but $T_2(I)$ does not. □

An additional corollary to Theorem 3 gives a simple row elimination rule for tableaux.

COROLLARY 2. *Let T be a tableau, w some row of T, and suppose there is some other row x of T such that in whatever column w and x disagree, w has a nondistinguished variable that appears nowhere else in T. Then the tableau T', obtained by deleting row w from T, is equivalent to T.*

Proof. We may map each row of T' to itself in T, and we may map each row of T other than w to itself, while mapping w to x. □

Example 8. In the first part of Example 7, row w_2 may be eliminated by w_1, which immediately transforms T_1 into T_2 and proves their equivalence. □

We state without proof two additional results for tableaux. There are analogous results for conjunctive queries [8].

THEOREM 4. *If T_1 and T_2 are equivalent tableaux, and neither is equivalent to a tableau with fewer rows, then there is a one-to-one correspondence of rows of T_1 to rows of T_2 that is a containment map in both directions.*

THEOREM 5. *Given any tableau T we can create a minimum row tableau equivalent to T by deleting some rows of T.*

Theorems 4 and 5 imply that for every tableau T there is a minimum row tableau equivalent to T that is unique up to renaming of symbols and reordering of rows; moreover, this minimum row tableau can be found by removing some of the rows of T. In § 5 we shall see that it is, nevertheless, a computationally difficult task to determine which rows of a tableau are redundant.

4.3. The effect of functional dependencies. When functional dependencies are present, we can use them to transform tableaux to equivalent forms. This can be done in the following way. Suppose X is a set of attributes, A is an attribute, and $X \to A$. Suppose also that two rows i and j of T have identical symbols in all columns corresponding to attributes of X. Let T' be constructed from T as follows.

(a) If rows i and j have two distinct constants in the column corresponding to A, then T' is $\varnothing$.

(b) Otherwise, make the symbols found in row i and row j of column A identical. If one of them is a constant then the resulting symbol is the same constant; if both of them are variables and one is distinguished, so is the resulting symbol.

LEMMA 2. *If T' is obtained from T as described above, then $T(I) = T'(I)$ for every instance I that satisfies the functional dependency $X \to A$.*

Proof. Let d_1 and d_2 be the symbols identified, and let d_3 be the symbol that replaces them in T'. Let S and S' be the sets of symbols of T and T' respectively. Suppose that I is any instance that satisfies $X \to A$, and $\rho : S \to C$ is a valuation under which each row of T becomes a member of I. Since I satisfies $X \to A$, we must have $\rho(d_1) = \rho(d_2)$. Define an assignment $\rho' : S' \to C$ as follows

$$\rho'(d) = \rho(d) \quad \text{if} \quad d \neq d_3, \quad \text{and} \quad \rho'(d_3) = \rho(d_1).$$

The application of ρ and ρ' to T and T' respectively produces identical results, and therefore $T(I) \subseteq T'(I)$.

The converse, that $T'(I) \subseteq T(I)$, is proved in a similar way. □

Example 9. Consider the expression $\pi_{AC}(AB \bowtie BC) \bowtie (AB \bowtie AD)$ whose syntax tree is shown in Fig. 2.

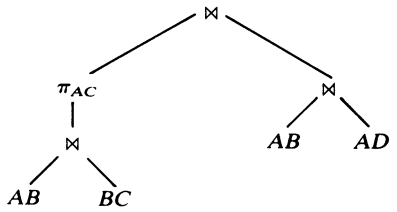


FIG. 2. *Syntax tree for expression.*

The tableau for this expression is

A	B	C	D
a_1	a_2	a_3	a_4
a_1	b_1	b_2	b_3
b_4	b_1	a_3	b_5
a_1	a_2	b_6	b_7
a_1	b_8	b_9	a_4

Suppose the functional dependencies $B \to A$ and $A \to C$ hold. Then $B \to A$ implies that $a_1 = b_4$, and then $A \to C$ implies that all of b_2, a_3, b_6 and b_9 are the same. Therefore the above tableau is equivalent to:

A	B	C	D
a_1	a_2	a_3	a_4
a_1	b_1	a_3	b_3
a_1	b_1	a_3	b_5
a_1	a_2	a_3	b_7
a_1	b_8	a_3	a_4

By Corollary 2 to Theorem 3, the first row may be eliminated in favor of the second row (or vice-versa), and then the remaining of these may be eliminated in favor of the third row, leaving

A	B	C	D
a_1	a_2	a_3	a_4
a_1	a_2	a_3	b_7
a_1	b_8	a_3	a_4

which implies that the given expression is equivalent to $ABC \bowtie ACD$ in the presence of the dependencies $B \to A \to C$. □

Suppose that T is a tableau and F is a given set of functional dependencies. Let i and j be two rows of T, and let X be the set of all the attributes whose corresponding columns have identical symbols in row i and row j. For every column in X^*, we can equate the symbols that appear in this column in row i and row j wherever they appear in T. This process can be applied recursively until no more symbols can be equated. The result is a tableau T' that is equivalent to T for every instance in which F holds, by Lemma 2. It is easy to show that T' is unique for T up to renaming of variables, since the above transformation on tableaux is a "Finite Church–Rosser System" [3]. Informally, if two symbols can be equated, they will always be equatable, no matter what other symbols are equated.

If no symbols of T may be equated because of a set of functional dependencies F, we say *T satisfies F*. The result T' of equating symbols of any tableau T according to the above rules, until no more can be equated is called the *limit of T with respect to F*. By using the algorithm of [5], [6] to compute X^* for sets of attributes X, we can construct the limit of T in time proportional to the square of the input size (the space needed to write down F and T). The algorithm is essentially that given in [1]. In the next theorem we show that in the presence of functional dependencies there is a weaker necessary and sufficient condition for inclusion or equivalence among tableaux.

THEOREM 6. *Let T_1 and T_2 be tableaux with limits T_1' and T_2' with respect to a set of functional dependencies F. Then $T_1(I) \supseteq T_2(I)$ for all instances I satisfying F if and only if $T_1' \supseteq T_2'$.*

Proof. By Lemma 2, $T_1(I) = T_1'(I)$ for all I satisfying F, and similarly for T_2 and T_2'. Thus the "if" portion is immediate. The converse is similar to the "only if" portion of Theorem 2. Here, we make T_2' into an instance I by assigning distinct constants to all its symbols. As T_2' satisfies F, I satisfies F. If $T_1(I) \supseteq T_2(I)$, then $T_1'(I) \supseteq T_2'(I)$. The existence of a homomorphism ψ from the symbols of T_1' to those of T_2' follows as in that theorem. Thus by Theorem 2, $T_1' \supseteq T_2'$. □

COROLLARY. *$T_1(I) = T_2(I)$ for all instances I satisfying F if and only if $T_1' \equiv T_2'$.*

Example 10. Let us continue Example 9, where the dependencies were $B \to A$ and $A \to C$. Consider the expression $AB \bowtie BC \bowtie AD$, whose tableau is

A	B	C	D
a_1	a_2	a_3	a_4
a_1	a_2	b_1	b_2
b_3	a_2	a_3	b_4
a_1	b_5	b_6	a_4

The limit of this tableau is

A	B	C	D
a_1	a_2	a_3	a_4
a_1	a_2	a_3	b_2
a_1	a_2	a_3	b_4
a_1	b_5	a_3	a_4

which is equivalent to the limiting tableau of Example 9, since the first row may be eliminated by Corollary 2 to Theorem 3. Thus the expression of Fig. 3 is equivalent to $AB \bowtie BC \bowtie AD$ if the dependencies $B \to A$ and $A \to C$ are given. Note that these expressions are not equivalent in general. □

5. NP-Completeness results concerning tableau equivalence. The obvious way to test the equivalence of two tableaux is to consider all possible containment mappings in each direction. Since the number of mappings from n_1 rows to n_2 rows is $n_2^{n_1}$, this procedure takes exponential time. One might therefore be interested in finding a procedure that takes less time. Using recent developments in complexity theory, however, we can prove that a substantially better algorithm is not likely to exist.

We assume the reader is familiar with the notion of an NP-complete problem. This class of problems was first considered in [12], [18]. There is strong evidence that these problems are intractable in general, that is, there is no algorithm for any of these problems which, on every input, will take less than exponential time. References [2], [16] present the methodology and theory behind NP-completeness results, as well as enumerating many of the known NP-complete problems.

In this section we show that the equivalence and containment problems for tableaux are NP-complete even in the following special cases:

(1) The tableaux come from expressions that have no select operators, but there is a set of functional dependencies that must be satisified.

(2) The tableaux come from expressions (including select operators), but no dependencies need be satisfied.

(3) There are no constants in the tableaux, nor are there dependencies, but the tableaux need not come from expressions.

Under the same conditions, the problem of determining whether $T_1 \subseteq T_2$ for two tableaux T_1 and T_2 is also NP-complete. Moreover, even if T_1 is a tableau with the same summary as T_2, and the rows of T_1 are a subset of those of T_2, it is NP-complete to determine whether $T_1 \equiv T_2$. This implies that minimizing the rows of a tableau is also very likely an exponential process in the worst case. Our NP-completeness results

strengthen those in [8] since our restricted relational expressions are a subset of the class of conjunctive queries.

5.1. The satisfiability problem. All the results use almost the same reduction from the 3-satisfiability problem, shown NP-complete in [12]; see also [2], [16]. Let $F = F_1F_2 \cdots F_q$ be a Boolean expression in conjunctive normal form, where the F_i's are clauses of three literals[1] each, and $x_1, x_2, \cdots, x_n$ are all the variables appearing in this expression. We construct two tableaux T_1 and T_2, each with $n+q$ columns, in the following way. T_1 has one row for each clause F_i. Let w_i be the row that corresponds to F_i. Let x_{i_1}, x_{i_2} and x_{i_3} be the variables that appear in F_i, either complemented or uncomplemented. Row w_i has the distinguished variable a_i in the ith column and the nondistinguished variables x_{i_1}, x_{i_2} and x_{i_3} in columns $q+i_1$, $q+i_2$ and $q+i_3$, respectively. The rest of the columns of w_i contain nondistinguished variables that appear nowhere else. The summary of T_1 has a_i in the ith column, $1 \leqq i \leqq q$, and blank in the other columns.

T_2 has seven rows for each row of T_1. Let w_i be a row of T_1. Each of the seven rows of T_2 that correspond to w_i represents some truth assignment to the variables of F_i under which F_i is true. Such a row has the distinguished variable a_i in the ith column and one of the seven lists of constants c_{i_1}, c_{i_2} and c_{i_3} in columns $q+i_1$, $q+i_2$ and $q+i_3$, respectively, such that each c_{i_j} is zero or one, and the assignment of the set of values c_{i_j} to x_{i_j} $(1 \leqq j \leqq 3)$ results in F_i being true. The rest of the columns contain distinct nondistinguished variables. The summary of T_2 is the same as that of T_1.

Example 11. Consider the Boolean expression

$$(x_1 + \bar{x}_2 + x_3)(\bar{x}_3 + x_4 + x_5).$$

Then $F_1 = (x + \bar{x}_2 + x_3)$ and $F_2 = (\bar{x}_3 + x_4 + x_5)$; q is 2 and n is 5. T_1 is:

F_1	F_2	x_1	x_2	x_3	x_4	x_5
a_1	a_2					
a_1	b_1	x_1	x_2	x_3	b_2	b_3
b_4	a_2	b_5	b_6	x_3	x_4	x_5

The seven rows of T_2 that correspond to the first row of T_1 are

$$(a_1, b_7, 1, 1, 1, b_8, b_9)$$
$$(a_1, b_{10}, 1, 1, 0, b_{11}, b_{12})$$
$$(a_1, b_{13}, 1, 0, 1, b_{14}, b_{15})$$
$$(a_1, b_{16}, 1, 0, 0, b_{17}, b_{18})$$
$$(a_1, b_{19}, 0, 1, 1, b_{20}, b_{21})$$
$$(a_1, b_{22}, 0, 0, 1, b_{23}, b_{24})$$
$$(a_1, b_{25}, 0, 0, 0, b_{26}, b_{27}).$$

[1] A *literal* is a variable or negated variable.

The rows of T_2 that correspond to the second row of T_1 are

$$(b_{28}, a_2, b_{29}, b_{30}, 1, 1, 1)$$
$$(b_{31}, a_2, b_{32}, b_{33}, 1, 1, 0)$$
$$(b_{34}, a_2, b_{35}, b_{36}, 1, 0, 1)$$
$$(b_{37}, a_2, b_{38}, b_{39}, 0, 1, 1)$$
$$(b_{40}, a_2, b_{41}, b_{42}, 0, 1, 0)$$
$$(b_{43}, a_2, b_{44}, b_{45}, 0, 0, 1)$$
$$(b_{46}, a_2, b_{47}, b_{48}, 0, 0, 0).$$

Note that the first seven rows do not include the combination 0, 1, 0, because if we assign 0 to x_1 and x_3 and 1 to x_2, then $F_1 = (x_1 + \bar{x}_2 + x)$ gets the truth value 0. Similarly, the last seven rows do not contain the combination 1, 0, 0. □

5.2. A class of tableaux that come from expressions. It happens that T_1 and T_2 are both obtainable from expressions by the construction preceding Theorem 1. These observations are special cases of a more general result, which we state as the next lemma. A *repeated symbol* in a particular column of a tableau is either

(1) a distinguished variable,
(2) a constant appearing in that column of the summary,
(3) a nondistinguished variable appearing in two or more rows.

Notice that a repeated symbol might appear in only one row if it is a distinguished variable or a constant appearing in the summary.

LEMMA 3. *If T is a tableau with at most one repeated symbol in any column, and such that any symbol appearing in the summary appears in at least one row in the same column, then there is a relational expression E, such that Theorem 1 applied to E yields T.*

Proof. For each row i of T, let R_i be the relation scheme consisting of the attributes in whose columns row i has a repeating symbol or other constant. Construct expression E_i by applying $\sigma_{A=c}$ to R_i for all attributes A whose column in row i has a constant c that does not appear in the same column of the summary. The tableau for E_i is a row like row i, but with distinguished variables in place of all repeated symbols, and with a summary containing the distinguished variable in exactly those columns in which row i has a repeated symbol.

Next, join all the E_i's. The result is an expression with a tableau like T, but with distinguished variables for all repeated symbols. Lastly, apply $\sigma_{A=c}$ for all A whose column has in the summary a constant c, and project onto those attributes such that the summary of T has a nonblank. The result is an expression with tableau T. □

COROLLARY. *T_1 and T_2 above come from expressions.*

Proof. T_1 has only the a_i's and (possibly) the x_i's as repeated symbols; T_2 has only the a_i's as repeated symbols.

Example 12. Consider T_1 of Example 11 and suppose the columns correspond to attributes $A_1, A_2, \cdots, A_7$. The repeated symbols are a_1, a_2, and x_3. The relation schemes for the two rows are $R_1 = A_1A_5$ and $R_2 = A_2A_5$. The expression corresponding to T_1 is $\pi_{A_1A_2}(A_1A_5 \bowtie A_2A_5)$. □

5.3. NP-Completeness results for expressions.

LEMMA 4. *Let T_1 and T_2 be constructed from a Boolean expression F as above. Then $T_1 \supseteq T_2$ if and only if F is satisfiable.*

Proof (*If*). Given an assignment that makes F true, we may construct a homomorphism ψ from the symbols of T_1 to those of T_2 as follows.

$\psi(a_i) = a_i$,

$\psi(x_i) = 0$ or 1 depending on the value assigned to x_i to make F true.

We may then map each row w of T_1 to that one of the seven corresponding rows that is $\psi(w)$ when we extend ψ to the rest of the nondistinguished variables. Since each nondistinguished variable of T_1 except for the x_i's appears only once, we can always extend ψ in this manner. Thus $T_1 \supseteq T_2$ by Theorem 2.

(*Only If*). Suppose there is a containment mapping of T_1 to T_2. Because of the a_i's, each row of T_1 must be mapped to one of the seven corresponding rows of T_2. Each x_i is mapped to either 0 or 1 consistently. The values chosen for the x_i's satisfy F, because the combinations of values making clauses false are not available as rows of T_2. Thus F is satisfiable. □

THEOREM 7. *Let U_1 and U_2 be two tableaux that are derived from restricted relational expressions. The following problems are NP-complete*:

(1) *Does $U_1 \supseteq U_2$?*

(2) *Is $U_1 \equiv U_2$?*

(3) *Let U_2 be a tableau that is obtained by deleting some of the rows of U_1. Is $U_1 \equiv U_2$?*

Proof. All these problems are in NP, because all we have to do is to guess a containment mapping and check whether it satisfies all the required conditions. Part (1) is immediate from Lemma 4.

For part (2), let $U_1 = T_1 \bowtie T_2$ and $U_2 = T_2$, where T_1 and T_2 are as above. Recall that the join is defined for tableaux by Theorem 1. Also note that U_1 is obtainable from an expression if T_1 and T_2 are. Since T_1 and T_2 define mappings whose values are relations with the same target relation scheme, the join is really intersection. Thus for any I, $U_1(I) = T_1(I) \cap T_2(I)$, and $U_1 \equiv U_2$ if and only if $T_1 \supseteq T_2$. Thus, equivalence is NP-complete by Lemma 4.

For part (3), simply observe that the rows of U_2 constructed in part (2) are a subset of the rows of U_1 in that part. □

Parts (1) and (2) of Theorem 7 say that the problem of testing equivalence or containment of expressions is almost certainly an intractable one, that is, no general algorithms of less than exponential complexity exist. Part (3) says that the problem of eliminating redundant rows of the tableau derived from one of these expressions is also likely to be intractable.

5.4. NP-Completeness results for tableaux. We should note the critical role played by constants in the proof of Lemma 4 and Theorem 7. However, if we are willing to relax our constraint that the tableaux come from expressions, then constants are not needed.

THEOREM 8. *The problems of Theorem 7 are NP-complete for general tableaux that have no constants.*

Proof. In T_2 defined previously, in each column replace 0 by a nondistinguished variable and 1 by another nondistinguished variable. The proof is then identical to Theorem 7. Note that T_2 does not in general come from any relational expression. □

5.5. NP-Completeness results with functional dependencies. In the presence of functional dependencies, we can prove similar results about tableaux that have only

variables and correspond to expressions with operations project and join only. The key idea is to use tableaux with $q+2n$ columns as follows. The first tableau $\hat{T}_1$ is simply obtained from T_1 by adding another n columns that contain only distinct nondistinguished variables.

To generate the second tableau $\hat{T}_2$, we modify the last n columns of T_2 as follows. First we replace in every column each occurrence of the constant 1 by the same nondistinguished variable, and each occurrence of the constant 0 is replaced by a distinct (for that occurrence) nondistinguished variable. The resulting columns are the $(q+1)$st, $\cdots$, $(q+n)$th columns of $\hat{T}_2$. Columns $q+n+1$ through $q+2n$ of $\hat{T}_2$ are obtained by a similar modification on columns $q+1$ through $q+n$ of T_2; each occurrence of the constant 0 in a particular column is replaced by the same nondistinguished variable, and each occurrence of the constant 1 is replaced by a distinct nondistinguished variable.

Both $\hat{T}_1$ and $\hat{T}_2$ correspond to expressions by Lemma 3. Let A_i be the attribute of the ith column. Suppose that we consider only instances in which the functional dependencies $A_{i+n} \to A_i$ $(q+1 \leqq i \leqq q+n)$ hold. Using these dependencies we can equate all the distinct variables, in the ith column $(q+1 \leqq i \leqq q+n)$, that stand for the truth value 0. Notice that each column between $q+1$ and $q+n$ already has a single symbol representing truth value 1. Therefore, $\hat{T}_1(1) \supseteq \hat{T}_2(I)$, for all instances I satisfying the dependencies, if and only if the Boolean expression F is satisfiable.

As a result of this reduction, we may conclude the following.

THEOREM 9. *Given a set of functional dependencies and two tableaux U_1 and U_2 that come from relational expressions with no select operations (and hence U_1 and U_2 have no constants), it is NP-complete whether, for all instances I satisfying the functional dependencies,*

(1) $U_1(I) \supseteq U_2(I)$

(2) $U_1(I) = U_2(I)$

(3) $U_1(I) = U_2(I)$ *given that the rows of U_2 are a subset of the rows of U_1.*

Proof. Let T_1' and T_2' be the limits of $\hat{T}_1$ and $\hat{T}_2$ above with respect to the functional dependencies given above. Then $T_1' = \hat{T}_1$, and in each of columns $q+1$ through $q+n$ of T_2', there is one nondistinguished variable where T_2, defined previously, has 0, and another where T_2 has 1. Other than this, the first $q+n$ columns of T_2' are the same as T_2. As T_1' has distinct nondistinguished variables in all positions of its last n columns, it follows as in Lemma 4 that there is a containment mapping from T_1' to T_2' if and only if the Boolean expression F is satisfiable. By Theorem 6, $T_1' \supseteq T_2'$ if and only if for all I satisfying the dependencies, $\hat{T}_1(I) \supseteq \hat{T}_2(I)$. Thus F is satisfiable if and only if for all instances I satisfying the dependencies, $\hat{T}_1(I) \supseteq \hat{T}_2(I)$. Parts (2) and (3) follow as in Theorem 7. □

6. A polynomial-time equivalence algorithm for a subclass of tableaux. In this section we define "simple tableaux," a large subclass of tableaux for which we can find a polynomial-time algorithm to decide equivalence.

6.1. Simple tableaux. A tableau is *simple* if in any column with a repeated nondistinguished variable there is no other symbol that appears in more than one row. It is not easy to produce an expression with a nonsimple tableau. The expression $\pi_{AC}(AB \bowtie BC) \bowtie (AB \bowtie BD)$ is in a sense a minimal expression that gives rise to a nonsimple tableau. The tableau is shown in Fig. 3. The rows in the column for B have repeated nondistinguished and distinguished variables.

A	B	C	D
a_1	a_2	a_3	a_4
a_1	b_1	b_2	b_3
b_4	b_1	a_3	b_5
a_1	a_2	b_6	b_7
b_8	a_2	b_9	a_4

FIG. 3. *A nonsimple tableau.*

Note that some simple tableaux do not come from expressions.

Intuitively, the algorithm for equivalence of simple tableaux works as follows. Suppose first that no column has any repeated nondistinguished variables. When we are dealing with equivalence, rather than containment, we can rule out containment mappings in which a distinguished variable maps to a constant. Therefore, to check for the existence of containment mappings in the situation where no nondistinguished variable repeats, we have only to examine each row r to see whether there is another row r' in the other tableau such that r' has a distinguished variable or identical constant wherever r has a distinguished variable or constant.

However, simple tableaux admit repeated nondistinguished variables in a column, provided there is not also another repeated symbol of any sort appearing in two rows of that column. Let T_1 and T_2 be equivalent simple tableaux and A a column of T_1 with repeated nondistinguished variable b_1. As T_1 and T_2 are equivalent, there is a containment mapping θ_1 from T_1 to T_2, and another containment mapping θ_2 from T_2 to T_1. It is easy to check that the composition of containment mappings is a containment mapping, so we may consider the containment mapping $\theta_2 \cdot \theta_1$ from T_1 to itself, as suggested in Fig. 4.

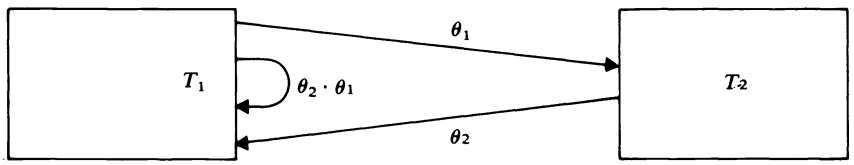


FIG. 4. *Composition of containment mappings.*

Now let us look at the set of rows S of T_1 that have b_1 in column A. There are two cases:

(a) $\theta_2 \cdot \theta_1$ maps rows in S to two or more rows of T_1.

(b) $\theta_2 \cdot \theta_1$ maps all rows in S to a single row r.

In case (b) we can eliminate all rows in S (except r if it is in S) from T_1, and the result will be a tableau equivalent to T_1. In case (a) we know that $\theta_2 \cdot \theta_1(w)$ is in S for all w in S, because by the hypothesis that T_1 is simple, no pair of rows other than those in S have the same symbol in the column for A. Moreover, θ_1 maps S to at least two rows, and these rows must have the same nondistinguished variable in column A. For if they had a distinguished variable or constant, θ_2 could not map them to rows in S. Thus in case (a) there is a repeated nondistinguished variable b_2 in column A of T_2.

Our algorithm works as follows. We search for a column A in which one tableau T_1 has a repeated nondistinguished variable in some set of rows S. If there exists a

containment mapping $\theta_2 \cdot \theta_1$ from T_1 to itself that maps all of S to one row r, then we eliminate S, and perhaps some other rows, to be determined later, in favor of r. If no such mapping exists, then only case (a) can apply, if $T_1 \equiv T_2$. Then θ_1 and θ_2 must map rows with b_1 to rows with b_2, and vice-versa. In this case we may "promote" b_1 and b_2 by treating them as constants. Ultimately, we eliminate all repeated nondistinguished variables, either by row elimination or by promotion. The resulting tableaux meet our earlier requirements for an efficient equivalence test, since they have no repeated nondistinguished variables. We now proceed to formalize the above argument.

6.2. Row covering. We say that row x of a tableau *covers* row w if the following hold.

(a) w and x have the same number of columns.

(b) If w has a distinguished variable in a given column, so does x. If w has a constant in a given column, then x has the same constant in this column.

We say x *covers* a set of rows S if x covers every row in S.

Example 13. Let

$T_1 =$

a_1		0	
a_1	b_2	b_1	b_3
b_4	b_2	0	b_3
a_1	b_2	b_6	b_7

$T_2 =$

a_4		a_5	
a_4	b_8	b_9	a_6
b_{10}	b_8	a_5	a_6

Both T_1 and T_2 are simple tableaux. The third row of T_1 is covered by the first row of T_1 or the first row of T_2. No row of T_1 covers the second row of T_2. □

LEMMA 5. *Let T_1 and T_2 be two simple tableaux without any repeated nondistinguished variables. Then $T_1 \equiv T_2$ if and only if T_1 and T_2 have identical summaries (up to renaming of distinguished variables), every row of T_1 is covered by some row of T_2, and every row of T_2 is covered by some row of T_1.*

Proof (If). We can map each row of T_1 to a row of T_2 that covers it. As there are no repeated nondistinguished variables, this mapping is a containment mapping. Thus $T_1 \supseteq T_2$. In the same way, $T_2 \supseteq T_1$, so $T_1 \equiv T_2$.

(Only if). A containment mapping θ from T_1 to T_2 surely maps distinguished variables to distinguished variables and constants to identical constants. Thus for every row r of T_1, r is covered by $\theta(r)$. The argument for the rows of T_2 is the same. □

6.3. Row closures. Suppose that T is a simple tableau. Let S be the set of all the rows of T that contain a repeated nondistinguished variable in one particular column. Let w be any row of T. The *closure* of S with respect to w, denoted $CL_w(S)$, is the minimal set of rows that contains S and satisfies the following condition:

> if x_1 is in $CL_w(S)$ and x_2 is any row of T such that x_1 and x_2 have the same repeated nondistinguished variable in some column, and w has a different symbol in this column, then x_2 is in $CL_w(S)$.

LEMMA 6. *Let T be a simple tableau and S the set of rows of T that contain a repeated nondistinguished variable in column A. Let w be a row of T, and let θ be defined by*

$$\theta(x) = \begin{cases} w & \text{for all } x \text{ in } CL_w(S), \\ x & \text{otherwise.} \end{cases}$$

Then θ is a containment mapping of T to T if and only if w covers $CL_w(S)$.

Proof (*Only if*). This portion is immediate from the definition of a containment mapping and of the covering relation.

(*If*). Suppose not. By the definition of the covering relation, we know that θ maps distinguished variables to distinguished variables, and constants to identical constants. Thus there exist rows y and z of T such that y and z have the same symbol d in some column B, and differing symbols in rows $\theta(y)$ and $\theta(z)$ of column B. Let us consider three cases.

Case 1. Neither y nor z is in $CL_w(S)$. Clearly $\theta(y)$ and $\theta(z)$ have the same symbol, d, in column B, so no violation occurs.

Case 2. y is in $CL_w(S)$ and z is not (or vice-versa). It is not possible that d is a nondistinguished variable, for it is repeated, and then by the definition of closure, z would be in $CL_w(S)$. (Note that $w = \theta(y)$ must differ in column B from $z = \theta(z)$, so w does not have d in column B.) If d is a distinguished variable or constant, then as w covers $CL_w(S)$ and y is in $CL_w(S)$, it follows that w has d in column B. But then $\theta(y) = w$ and $\theta(z) = z$ each have the same symbol d in column B, contrary to assumption.

Case 3. y and z are in $CL_w(S)$. Then $\theta(y) = \theta(z) = w$, so no violation of the containment mapping condition can be found. □

Let us define a *w-chain* to be a sequence of rows $z_1, z_2, \cdots, z_k$, $k \geqq 1$, such that for $1 \leqq i < k$, there is some column in which z_i and z_{i+1} have the same nondistinguished variable, and in which w does not have this variable. Then, by definition of closure, z_1 is in $CL_w(S)$ if and only if there exists a w-chain $z_1, z_2, \cdots, z_k$, such that z_k is in S.

LEMMA 7. *Suppose A and B are two columns of a simple tableau T with repeated nondistinguished variables in sets of rows* S_1 *and* S_2, *respectively. Suppose x covers* $CL_x(S_1)$ *and* θ_1 *is the containment mapping that sends* $CL_x(S_1)$ *to x and other rows to themselves. Let* T' *be* T *with the rows of* $CL_x(S_1) - \{x\}$ *eliminated. Let* $S_3 = S_2 - (CL_x(S_1) - \{x\})$. *It follows that if* S_3 *contains two or more rows, and in* T', *w covers* $CL_w(S_3)$, *then in T, w covers* $CL_w(S_2)$.

Proof. Case 1. Suppose x is not in $CL_w(S_3)$ in T'. We prove by induction on the length of a w-chain in T from y to some z in S_2, that in T', y is in $CL_w(S_3)$.

Basis. Length = 1. Here y is in S_2, since it has the same nondistinguished variable as z in column B. Suppose y is not in S_3. Then y is in $CL_x(S_1)$. Since S_3 has at least two elements, we may assume that z is in $S_3 - \{x\}$ and, therefore, z is not in $CL_x(S_1)$. Then x must have the same nondistinguished variable as y and z in column B, else z would be in $CL_x(S_1)$. Therefore x is in S_2, and as x is certainly not in $CL_x(S_1) - \{x\}$, it follows that x is in S_3, a contradiction.

Induction. Let there be a chain of length $k > 1$, say $y = z_1, z_2, \cdots, z_k = z$ from y to z. By the inductive hypothesis, z_2 is in $CL_w(S_3)$ in T'. Now there is a column such that y and z_2 have the same nondistinguished variable, and w has a different symbol there. If x has the repeated nondistinguished variable in that column, then x is in $CL_w(S_3)$. As we assume x not to be in $CL_w(S_3)$, if y is in $CL_x(S_1)$, then z_2 is in $CL_x(S_1)$, and therefore not in $CL_w(S_3)$. It follows that y is present in T' and therefore in $CL_w(S_3)$ in T'. Thus w covers $CL_w(S_2)$ in T, and the lemma follows.

Case 2. x is in $CL_w(S_3)$ in T'. Let θ_2 be the containment mapping on T' that sends members of $CL_w(S_3)$ to w and other rows of T' to themselves. Then $\theta_2\theta_1$ is a containment mapping on T. We claim that $\theta_2\theta_1$ maps all of $CL_w(S_2)$ to w. Let y be in $CL_w(S_2)$. If y is in $CL_x(S_1)$, then θ_1 maps y to x, and θ_2 maps x to w.

If y is not in $CL_x(S_1)$ but is in $CL_w(S_2)$, then there is in T a w-chain $y = z_1, z_2, \cdots, z_n$, where z_n is in S_2. An induction similar to the one above shows that y is in $CL_w(S_3)$ in T'. We prove the inductive step. If z_2 is in $CL_w(S_3)$, then so is y. Therefore

assume z_2 is in $CL_x(S_1)-\{x\}$. Consider the column in which y and z_2 have some nondistinguished variable, and w differs. If x does not have the repeated nondistinguished variable in this column, then y is in $CL_x(S_1)$. But in the opposite case, as x is in $CL_w(S_3)$, it follows that y is also, proving the induction.

As $\theta_2\theta_1$ is a containment mapping that sends all of $CL_w(S_2)$ to w, it must be that w covers $CL_w(S_2)$ in T. The present lemma then follows from Lemma 6. □

Consider again a simple tableau T. Let S be the set of all the rows with a repeated nondistinguished variable in a particular column. If we can find a row w that covers every row in $CL_w(S)$, then we can reduce T to an equivalent tableau T' by deleting all the rows of $CL_w(S)$ (except w, if w is in S) from T. This reduction rule can be applied repeatedly, to any column of T' that has a repeated variable, until we get a tableau that cannot be reduced further.

Example 14. Let

$T =$

a_1	a_2			
a_1	a_2	b_1	b_2	b_3
a_1	b_4	b_7	b_5	b_3
b_6	a_2	b_7	b_2	b_8
b_9	a_2	b_{10}	b_{11}	b_3

T is a simple tableau. Let $S=\{1, 3\}$[2] be the set of all the rows with variable b_2. $CL_1(S)=\{1, 2, 3\}$. That is, we begin with $CL_1(S)=S=\{1, 3\}$. Then, as row 2 has in column 3 the same repeated nondistinguished variable as row 3, but row 1 does not have this symbol, we add 2 to $CL_1(S)$. Row 4 has only distinguished variables and nonrepeated nondistinguished variables, except in column 5. But rows 1 and 4 have the same symbol there, so we cannot add 4 to $CL_1(\{1, 2, 3\})$.

The first row covers every row in this closure and, therefore, T can be reduced to

a_1	a_2			
a_1	a_2	b_1	b_2	b_3
b_9	a_2	b_{10}	b_{11}	b_3

.

Now, consider all the rows with the repeated variable b_3—these are all the remaining rows, and the first row covers them. Thus the above tableau is reduced to

a_1	a_2			
a_1	a_2	b_1	b_2	b_3

.

6.4. Promotion of repeated nondistinguished variables. We shall now prove that if for no w can $CL_w(S)$ be eliminated by Lemma 6, then the repeated nondistinguished variable that gave rise to S can be promoted to a constant.

[2] 1 means the first row, etc.

LEMMA 8. *Let T_1 and T_2 be simple tableaux, and let A be a column with a repeated nondistinguished variable b_1 appearing in set of rows S of T_1. Suppose also that there is no row w such that w covers $CL_w(S)$. Then*:

(a) *If $T_1 \equiv T_2$, then there is a repeated nondistinguished variable b_2 in the A column of T_2.*

(b) *If b_2 exists, and T_1' and T_2' are the tableaux that result from T_1 and T_2 by replacing b_1 and b_2 by the same constant, a constant that appears nowhere else, then T_1' and T_2' are simple, and $T_1 \equiv T_2$ if and only if $T_1' \equiv T_2'$.*

Proof (a). As $T_1 \equiv T_2$, let θ_1 and θ_2 be containment mappings from T_1 to T_2 and back, respectively. Let S' be the set of rows of T_2 such that $S' = \{\theta_1(w) | w \text{ is in } S\}$, and let $S'' = \{\theta_2 \cdot \theta_1(w) | w \text{ is in } S\}$. Then S'' has two or more members, and so does S'. The rows of S' have some one symbol d in column A. If d is a distinguished variable or constant, then as θ_2 is a containment mapping, the rows of S'' all have a distinguished variable or constant in column A. As there are at least two rows in S'', we violate our assumption that T_1 is simple. Therefore d is a repeated nondistinguished variable of T_2.

(b) We know containment mappings between T_1 and T_2 exist, if $T_1 \equiv T_2$. If these mappings did not map b_1 to b_2 and vice-versa then there would be a containment mapping from T_1 to itself that mapped S to one row, since no repeated symbols but b_1 and b_2 exist in their columns. We would thus violate our assumption that no w covers $CL_w(S)$. It follows that the containment mappings between T_1 and T_2 also serve for T_1' and T_2'. Conversely, containment mappings between T_1' and T_2' surely serve for T_1 and T_2. The fact that T_1' and T_2' are simple is obvious. □

6.5. The algorithm. We say a simple tableau is in *reduced form* if it has no repeated nondistinguished variables. Lemmas 6 and 8 can be used to put simple tableaux in reduced form, and Lemma 5 can be used to test the equivalence of two such tableaux. The algorithm is summarized in Fig. 5. The procedure REDUCE(T_1, T_2) puts T_1 in reduced form and also returns **false** if $T_1 \not\equiv T_2$ is detected. REDUCE returns **true** if it does not detect that $T_1 \not\equiv T_2$; note that T_1 may still not be equivalent to T_2 in this case.

THEOREM 10. *The algorithm of Fig. 5 correctly decides the equivalence of simple tableaux in $O(s^3t^2)$ time if the tableaux have a maximum of s rows and t columns.*

Proof. Lines (1)–(5) apply Lemma 6. The only important detail is that after looking at each column A and row w once, we need not reconsider A and w if they fail the test of line (4) once. In proof, note that by Lemma 7 applied once for each application of Lemma 6, no new opportunities for reduction are created as reductions are made.

Lines (6)–(10) implement Lemma 8, so the resulting T_1 is in reduced form. The test of line (14) then decides the issue by Lemma 5, if line (7) has not already detected that $T_1 \not\equiv T_2$.

For the running time of Fig. 5, we note that the loop of lines (1)–(5) is executed st times. Computation of $CL_w(S)$ at line (4) takes time $O(s^2t)$, since $O(st)$ is sufficient to check if any rows can be added to the closure, and at most s rows can be added. Thus the loop of (1)–(5) takes $O(s^3t^2)$ time. Clearly $O(st)$ time suffices for the loop of (6)–(10), so REDUCE takes $O(s^3t^2)$ time.

In the main procedure, lines (12) and (13) take $O(s^3t^2)$ time by the foregoing argument. Line (14) takes $O(s^2t)$ time, so the entire algorithm takes $O(s^3t^2)$ time. □

COROLLARY. *If n is the size of the input (i.e., n is the space needed to write down T_1 and T_2), then the algorithm of Fig. 5 takes $O(n^3)$ time.*

Proof. Note that st could be replaced by n in the above analysis, and $s \leqq n$ is obvious. □

```
     procedure REDUCE(T1, T2);
     begin
(1)      for each column A of T1 and row w of T1 do
(2)          if A has a repeated nondistinguished variable b then
                 begin
(3)                  let S be the set of rows in which b appears;
(4)                  if w covers CL_w(S) then
(5)                      remove the rows in CL_w(S)−{w} from T1
                 end
(6)      for each column A of T1 in which a repeated
             nondistinguished variable b1 remains do
             begin
(7)              if the column for A in T2 has no repeated nondistinguished
                 variable then
(8)                  return false; /*T1 ≰ T2*/
(9)              let b2 be the repeated nondistinguished variable in column A of T2;
(10)             make b1 and b2 be the same new constant;
             end;
(11)     return true
     end REDUCE;
     begin /* main procedure */
(12)     if ¬REDUCE(T1, T2) then return false;
         /* as a side effect, T1 is reduced */
(13)     if ¬REDUCE(T2, T1) then return false:
             /* as a side effect, T2 is reduced */
             /* note that lines (6)–(10) of REDUCE are not needed here */
(14)     if every row of T1 is covered by a row of T2, and vice versa then
(15)         return true
         else
(16)         return false
     end
```

FIG. 5. *Polynomial algorithm to test equivalence of simple tableaux.*

Note that the coverage of each row of T_1' by a row T_2' is a sufficient, but not a necessary, condition for $T_2' \subseteq T_1'$ (even when both T_1' and T_2' are in reduced form). Also observe that the results of Section 5 imply that containment is NP-complete for simple tableaux.

7. Extension to strong equivalence. The equivalence and containment results of the previous sections also apply to strong equivalence. We shall state these results here without proof. In each case the proof is analogous to that of the corresponding result about weak equivalence. We can use a modified form of tableau to represent values of expressions as mappings from their operands, rather than from an instance of the universe. The modifications that must be made are:

(1) rows are *tagged* with the relation from which they come,

(2) rows have blanks in columns corresponding to attributes that are not part of the relation with which the row is tagged.

Suppose T is such a tableau, with set of symbols S, summary w_0 and rows $w_1, w_2, \cdots, w_n$. Suppose $R_1, R_2, \cdots, R_k$ are the available relation schemes, and

$r_1, r_2, \cdots, r_k$ are corresponding relations. Let w_i be tagged by R_{j_i}, for $1 \leqq i \leqq n$. Then

$$T(r_1, r_2, \cdots, r_k) = \{\rho(w_0) | \text{for some } \rho : S \to D \text{ we have } \rho(w_i) \text{ in } r_{j_i} \text{ for } 1 \leqq i \leqq n\}.$$

7.1. The strong equivalence test. Tagged tableaux can be constructed from expressions exactly as in Theorem 1. The only modification is that the tableau for a relation scheme R has blank, rather than a nondistinguished symbol, in columns that do not correspond to attributes of R. We shall state the following analog of Corollary 1 to Theorem 2.

THEOREM 11. *Two tableaux are strongly equivalent if and only if containment maps that preserve tags exist in both directions.*

Example 15. Consider the expression $E = \pi_{AB}(AB \bowtie BC)$ from Example 3. The tagged tableau for AB is

A	B	C	
a_1	a_2		
a_1	a_2		(AB)

and for BC it is

A	B	C	
	a_2	a_3	
	a_2	a_3	(BC)

The tagged tableau for $AB \bowtie BC$ is

A	B	C	
a_1	a_2	a_3	
a_1	a_2		(AB)
	a_2	a_3	(BC)

and for E it is

A	B	C	
a_1	a_2		
a_1	a_2		(AB)
	a_2	b_1	(BC)

Note that a tag-preserving containment mapping from the tableau for AB to the above tableau exists, implying that $AB \supseteq \pi_{AB}(AB \bowtie BC)$ in the strong sense. However, no tag-preserving containment mapping exists in the other direction, since the tableau for AB has no row tagged (BC). Thus E is not strongly equivalent to AB, although we saw in Example 5 that these expressions are weakly equivalent. □

7.2. Functional dependencies. We may apply functional dependencies to tagged tableaux exactly as in Theorem 6. The two rows involved need not have the same tag,

provided we understand that functional dependencies apply to two or more relations jointly. For example, suppose that ABC and ABD are relation schemes, and $A \to B$ is a functional dependency. Then it is not permissible to have $a_1b_1c_1$ in ABC and $a_1b_2d_1$ in ABD. If we do not make this prohibition, then functional dependencies may only be applied to rows with the same tag.

7.3. Polynomial-time reductions between weak and strong equivalence. We can prove general results which show that the questions of weak and strong equivalence are almost the same problem.

LEMMA 9. *Let E_1 and E_2 be expressions. Then in time polynomial in the size of E_1 and E_2 we can construct expressions E_1' and E_2' such that E_1' and E_2' are strongly equivalent if and only if E_1 and E_2 are weakly equivalent.*

Proof. Let $R_1, R_2, \cdots, R_k$ be all the arguments of E_1 and E_2, and let $R = \bigcup_{i=1}^{k} R_i$. Construct E_1' and E_2' by replacing each operand R_i by $\pi_{R_i}(R)$. Then T behaves as a univeral relation, and a proof that E_1' is strongly equivalent to E_2' if and only if E_1 and E_2 are weakly equivalent is immediate from definitions. □

LEMMA 10. *Let E_1 and E_2 be expressions. Then in time polynomial in the size of E_1 and E_2 we may construct expressions E_1' and E_2' that are weakly equivalent if and only if E_1 and E_2 are strongly equivalent.*

Proof. Let G be a new attribute and let $R_1, R_2, \cdots, R_k$ be the operands of E_1 and E_2. Let $R_i' = R_i \cup \{G\}$ for all i. Construct E_1' and E_2' from E_1 and E_2 by replacing operand R_i by $\pi_{R_i}(\sigma_{G=i}(R_i'))$. Then the projection of the universal instance onto R_i', followed by selection of $G = i$ and projection to remove the G column yields a relation that is independent of any other relation derived from that instance by selection of another value of G. □

7.4. Complexity results for strong equivalence.

THEOREM 12. *Strong equivalence is NP-complete in each of the following cases.*

(i) *Tableaux are not required to come from expressions but may not have constants, nor may there be functional dependencies.*

(ii) *Tableaux are permitted to have constants, but must come from expressions and there may be no functional dependencies.*

(iii) *Functional dependencies are permitted, but tableaux must come from expressions and may not have constants.*

Proof. The construction of Lemma 9 preserves the absence of constants and the property that a tableau comes from an expression. The absence of functional dependencies is surely preserved. Note that the construction of Lemma 9 may be applied to tableaux as well as expressions, by simply filling out blanks in rows by new nondistinguished variables, so part (i) has meaning. The theorem then follows immediately from Theorems 7, 8, and 9. □

THEOREM 13. *Strong equivalence is decidable in polynomial time for expressions that have simple tableaux.*

Proof. The construction of Lemma 10 preserves simplicity of tableaux, as the column for G has only constants. □

Let T be any tableau tagged with relation schemes. For each repeated symbol s, let TAG(s) be the set of tags of rows that contain s. A *global repeated nondistinguished variable* is a nondistinguished variable b such that TAG(b) contains two or more tags. A tableau is *quasi-simple* if the following hold.

(a) If b is a global repeated nondistinguished variable in column A, then for every repeated symbol s, $s \neq b$, in column A, TAG(b) $\not\subseteq$ TAG(s).

(b) For each tag the set of all the rows with this tag is a simple tableau.

Note that a simple tableau is also quasi-simple. Condition (a) implies that a global repeated nondistinguished variable cannot be eliminated by row covering, and therefore it can be promoted to a constant immediately. Using condition (b), we can now minimize each set of rows with the same tag separately, using the algorithm for simple tableaux.

This approach can also be used whenever a tableau has a pattern of constants and/or distinguished variables that decompose each tableau to several disjoint sets of rows, such that no rows in one set can be mapped to a row in any other set.

8. Conclusions and open problems. Using tableaux, we have developed a "crank" that can be turned to tell whether two expressions over the set of relational operators select project, and (natural) join, are equivalent. The "crank" is capable of accounting for the effect of functional dependencies and works for either weak or strong equivalence. Although the "crank" requires exponential time in the general case, we have isolated an important special case for which a polynomial time equivalence algorithm was developed.

We have not considered the natural next step, which is to develop tools for the efficient optimization of expressions, given an arbitrary cost criterion. Our NP-completeness results suggest that any method involving canonicalization of expressions is likely to require considerable computational effort for general expressions, so the optimization problem appears to be very hard. However, the following problems appear appropriate for examination.

(1) How far can we extend the class of expressions for which equivalence is efficiently decidable?

(2) Can the equivalence test be made to work in even exponential time when there are multivalued dependencies [7], [14], [15], [26] that must be satisfied? A doubly exponential algorithm follows from the techniques of [1] for multivalued dependencies.

(3) Find a complete axiom system to transform an expression into any equivalent one. Note that the number of steps needed to go between equivalent expressions might be polynomial in their size without violating the NP-completeness results or proving P = NP, as finding the right sequence of steps might be hard.

REFERENCES

[1] A. V. AHO, C. BEERI AND J. D. ULLMAN, *The theory of joins in relational databases*, Proc. 18th IEEE Symposium on Foundations of Computer Science, 1977, pp. 107–113.

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

[3] A. V. AHO, R. SETHI AND J. D. ULLMAN, *Code optimization and finite Church–Rosser systems*, Design and Optimization of Compilers, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 89–105.

[4] W. W. ARMSTRONG, *Dependency structures of data base relationships*, Proc. IFIP 74, North Holland, 1974, pp. 580–583.

[5] P. A. BERNSTEIN, *Synthesizing third normal form relations from functional dependencies*, ACM Trans. on Database Sys., 1 (1976), pp. 277–298.

[6] P. A. BERNSTEIN AND C. BEERI, *An algorithmic approach to normalization of relational database schemes*, TR CSRG-73, Computer Science Research Group, University of Toronto, Sept. 1976.

[7] C. BEERI, R. FAGIN AND J. H. HOWARD, *A complete axiomatization for functional and multivalued dependencies*, Proc. ACM SIGMOD International Conference on the Management of Data, August, 1977, pp. 47–61.

[8] A. K. CHANDRA AND P. M. MERLIN, *Optimal implementation of conjunctive queries in relational data bases*, Proc. Ninth Annual ACM Symposium on Theory of Computing, May, 1976, pp. 77–90.

[9] E. F. CODD, *A relational model for large shared data banks*, Comm. ACM 13, (1970), pp. 377–387.
[10] ———, *Further normalization of the data base relational model*, Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 33–64.
[11] ———, *Relational completeness of data base sublanguages*, Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 65–98.
[12] S. A. COOK, *The complexity of theorem proving procedures*, Proc. 3rd Annual ACM Symposium on Theory of Computing, May, 1971, pp. 151–158.
[13] C. J. DATE, *An Introduction to Database Systems*, second ed., Addison-Wesley, Reading, MA, 1977.
[14] C. DELOBEL, *Contributions théoretiques à la conception d'un système d'informations*,' Ph.D. thesis, Univ. of Grenoble, Oct., 1973.
[15] R. FAGIN, *Multivalued dependencies and a new normal form for relational data-bases*, ACM Trans. Database Sys., 2, (1977), pp. 262–278.
[16] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.
[17] P. A. V. HALL, *Optimization of a single relational expression in a relational data-base system*, IBM J. Res. Develop., 20 (1976), pp. 244–257.
[18] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.
[19] J. MINKER, *Performing inferences over relational databases*, Proc. ACM SIGMOD International Conference on Management of Data (May 1975, San Jose, California), pp. 79–91.
[20] F. P. PALERMO, *A database search problem*, Information Systems COINS IV, J. T. Tou, ed., Plenum Press, New York, 1974.
[21] R. M. PECHERER, *Efficient evaluation of expressions in a relational algebra*, Proc. ACM Pacific Conf., April, 1975, pp. 44–49.
[22] J. RISSANEN, *Independent components of relations*, ACM Trans. Database Sys., 2(1977), pp. 317–325.
[23] J. M. SMITH AND P. Y.-T. CHANG, *Optimizing the performance of a relational algebra database interface*, Comm. ACM, 18(1975), pp. 568–579.
[24] M. STONEBRAKER AND L. A. ROWE, *Observations on data manipulation languages and their embedding in general purpose programming languages*, 2 TR UCB/ERL M77-53, University of California, Berkeley, July 1977.
[25] E. WONG AND K. YOUSSEFI, *Decomposition—a strategy for query processing*, ACM Trans. Database Sys. 1, (1976), pp. 223–241.
[26] C. ZANIOLO, *Analysis and design of relational schemata for database systems*, Tech. Rept. UCLA-ENG-7769, Department of Computer Science, UCLA, July, 1976.
[27] M. M. ZLOOF, *Query-by-Example: the invocation and definition of tables and forms*, Proc. ACM International Conf. on Very Large Data Bases, Sept., 1975, pp. 1–24.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0010 $01.00/0

DECISION PROBLEMS FOR MULTIVALUED DEPENDENCIES IN RELATIONAL DATABASES*

KENICHI HAGIHARA†, MINORU ITO†, KENICHI TANIGUCHI† AND TADAO KASAMI†

Abstract. Two decision problems related to multivalued dependencies in a relational database are considered. In this paper, an algorithm is presented for deciding whether or not a multivalued dependency can be derived from sets F of functional dependencies and M of multivalued dependencies on a set U of attributes, whose running time is proportional to $\min(k^2|U|, \|F \cup M\|^2)$ where k and $|U|$ are the numbers of dependencies in $F \cup M$ and attributes in U, respectively, and $\|F \cup M\|$ is the size of description of F and M. A related algorithm is also considered which decides whether or not there exists a nontrivial multivalued dependency that is valid in a projection of the original relation.

Key words. relational database, functional dependency, multivalued dependency, inference rule, membership problem, dependency basis, projection, nontrivial multivalued dependency

1. Introduction. The structure of a relational database is defined by various relationships that hold between its components. The functional dependency (for short, FD) and the multivalued dependency (for short, MVD) are examples of such relationships [4], [5], [6], [7], [9].

It is known that given a set of dependencies, additional dependencies can be derived by using inference rules and that there are complete sets of inference rules [1], [2], [8]. In this paper, we consider the following problems: (1) (membership problem) Given a set of dependencies and an additional dependency, can this dependency be derived from the given set by using the complete inference rules? (2) Given a set of dependencies on a set U of attributes, and a subset U' of U, does there exist a nontrivial MVD in the projection on U' of the original relation? These are fundamental problems in the relational database model and are useful to obtain a set of normalized relations [9]. A linear time membership algorithm for FD's was presented in [4].

In § 3, we present an efficient membership algorithm for the case where the given dependencies are both functional and multivalued. Given sets F of FD's and M of MVD's, it takes at most $O(\|F \cup M\|^2)$ time, where $\|F \cup M\|$ is the size of description of F and M (for more precise upper time bound, see § 3). This algorithm was shown in the authors' previous paper [10].

Beeri [3] also presented a polynomial ($O(\|F \cup M\|^4)$) time membership algorithm for functional and multivalued dependencies. It turned out that the basic idea of our algorithm is essentially the same as the Beeri one and our algorithm is a refinement of the Beeri one.

In § 2, we define the concepts that are used in this paper. In § 3, an algorithm for computing the dependency basis is presented. The dependency basis [2], [3] is used for membership problem. In § 4, we discuss problem (2) and present a decision procedure.

2. Basic concepts.

2.1. Relation, operations and dependencies. *Attributes* are symbols taken from a finite set $U = \{A_1, A_2, \cdots, A_N\}$. With each attribute A is associated a *domain*, denoted by DOM(A), of possible values for that attribute. We use upper case letters near the end of the alphabet for sets of attributes. We may not distinguish between the attribute A and the singleton set $\{A\}$.

* Received by the editors January 9, 1978, and in revised form August 16, 1978.

† Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, Toyonaka, Osaka, Japan.

For a set $X \subseteq U$, an *X-value* is a mapping that assigns to each element $A \in X$ a value from DOM(A). A *relation* R on U, denoted by $R(U)$, is a subset of the cross product $\mathrm{DOM}(A_1) \times \mathrm{DOM}(A_2) \times \cdots \times \mathrm{DOM}(A_N)$. The elements of a relation are called *tuples*.

If u is a tuple in a relation $R(U)$ and A is an attribute in U, then $u[A]$ is the A-component of u. Similarly, if $U' = \{A_{i_1}, A_{i_2}, \cdots, A_{i_j}\}$ is a subset of U, then $u[U']$ is the tuple $(u[A_{i_1}], u[A_{i_2}], \cdots, u[A_{i_j}])$. Given a relation $R(U)$ and a subset U' of U, the *projection* of R on U', denoted by $R[U']$, is defined by:

$$R[U'] = \{u[U'] \mid u \in R(U)\}.$$

Note that $R[U']$ is also a relation on U'.

A *functional dependency* [5] (for short, FD) is a statement f: $X \to Y$ where X and Y are subsets of U. The FD f is *valid* in a relation $R(U)$ if for every two tuples u and v in $R(U)$, $u[X] = v[X]$ implies $u[Y] = v[Y]$. We usually omit the name f of the FD and write $X \to Y$. As usual, we assume, without loss of generality, that the right-hand side of each FD is a singleton set.

Let R be a relation on U, let X and Y be subsets of U (not necessarily disjoint) and let x be an X-value. We define

$$Y_R(x) = \{u[Y] \mid u \in R \wedge u[X] = x\}.$$

A *multivalued dependency* [9] (for short, MVD) on U is a statement m: $X \twoheadrightarrow Y(U)$ where X and Y are subsets of U. Let $Z = U - (X \cup Y)$. The MVD m is valid in a relation $R(U)$ if for every $X \cup Z$-value, xz, that appears in R, we have $Y_R(xz) = Y_R(x)$. We usually omit the name m of the MVD and write $X \twoheadrightarrow Y(U)$.

2.2. Inference rules for dependencies. Suppose that for sets F of FD's and M of MVD's on U, all dependencies in $F \cup M$ are valid in a relation $R(U)$. Then, it is possible to infer additional dependencies that are also valid in $R(U)$. There are three groups of inference rules. The first one contains inference rules for FD's, which were studied in [1], [2] and [8], and are called FD rules. FD rules allow us to infer additional FD's from given FD's. In the rules, X, Y, Z and W are arbitrary subsets of U.

FD rules

FD1 (Reflexivity). If $Y \subseteq X$ then $X \to Y$.

FD2 (Augmentation). If $Z \subseteq W$ and $X \to Y$ then $X \cup W \to Y \cup Z$.

FD3 (Transitivity). If $X \to Y$ and $Y \to Z$ then $X \to Z$.

The second one contains inference rules for MVD's, which were studied in [2] and [9] and are called MVD rules. MVD rules allow us to infer additional MVD's from given MVD's.

MVD rules.

MVD0 (Complementation). If $X \twoheadrightarrow Y(U)$ then $X \twoheadrightarrow U - Y(U)$.

MVD1 (Reflexivity). If $Y \subseteq X$ then $X \twoheadrightarrow Y(U)$.

MVD2 (Augmentation). If $Z \subseteq W$ and $X \twoheadrightarrow Y(U)$ then $X \cup W \twoheadrightarrow Y \cup Z(U)$.

MVD3 (Transitivity). If $X \twoheadrightarrow Y(U)$ and $Y \twoheadrightarrow Z(U)$ then $X \twoheadrightarrow Z - Y(U)$.

MVD4 (Pseudo-Transitivity). If $X \twoheadrightarrow Y(U)$ and $Y \cup W \twoheadrightarrow Z(U)$ then $X \cup W \twoheadrightarrow Z - (Y \cup W)(U)$.

MVD5 (Union). If $X \twoheadrightarrow Y(U)$ and $X \twoheadrightarrow Z(U)$ then $X \twoheadrightarrow Y \cup Z(U)$.

MVD6 (Decomposition). If $X \twoheadrightarrow Y(U)$ and $X \twoheadrightarrow Z(U)$ then $X \twoheadrightarrow Y \cap Z(U)$, $X \twoheadrightarrow Y - Z(U)$ and $X \twoheadrightarrow Z - Y(U)$.

The rules MVD4 ~ MVD6 are implied by the rules MVD0 ~ MVD3. However, those are useful for the manipulation of MVD's, and therefore are listed here.

The last group contains "mixed" inference rules, which were studied in [2] and [9] and are called FD–MVD rules. The rule FD–MVD1 follows from the definitions of FD and MVD. The rules FD–MVD2 and FD–MVD3 show that certain combinations of FD's and MVD's imply additional FD's that cannot be derived by using the rules in the preceding two groups.

FD–MVD rules.

FD–MVD1. If $X \to Y$ then $X \twoheadrightarrow Y(U)$.

FD–MVD2. If $X \twoheadrightarrow Z(U)$ and $Y \to Z'$ where $Z' \subseteq Z$ and Y and Z are disjoint, then $X \to Z'$.

FD–MVD3. If $X \twoheadrightarrow Y(U)$ and $X \cup Y \to Z$, then $X \to Z - Y$.

Given the rules in the preceding two groups, one of the rules FD–MVD2 and FD–MVD3 can be derived from the other.

It is said that a dependency (FD or MVD) d can be *derived* from a set D of dependencies if it can be inferred from D by a sequence of applications of the inference rules. Given a set D of dependencies and a set I of inference rules, the *closure of D under I* is defined to be the set of all dependencies that can be derived from D by using the rules in I (including the dependencies in D).

A set I of inference rules is *complete* for the corresponding family of dependencies (the family of FD's, the family of MVD's or the family of all dependencies) if, for every set D of dependencies from the family and for every dependency d from the family that is not in the closure of D under I, there exists a relation in which all dependencies of D (and hence all dependencies of the closure of D under I) are valid but in which d is not valid.

PROPOSITION 1. (1) *The set of rules* {FD1, FD2, FD3} *is complete for the family of FD's* [1], [2], [8].

(2) *The set of rules* {MVD0, MVD1, MVD2, MVD3} *is complete for the family of MVD's* [2].

(3) *The set of rules* {FD1, FD2, FD3, MVD0, MVD1, MVD2, MVD3, FD–MVD1, FD–MVD2} *is complete for the family of all dependencies* [2].

2.3. Dependency basis. Let F and M be sets of FD's and MVD's on U, respectively. The closure of $F \cup M$ under the set of inference rules {FD1, FD2, FD3, MVD0, MVD1, MVD2, MVD3, FD–MVD1, FD–MVD2} is denoted by $(F \cup M)^+$. Note that this set of rules is complete for the family of all dependencies.

For F, let $\bar{F} = \{X \twoheadrightarrow \{A\}(U) \mid X \to \{A\} \in F\}$ and let $G = \bar{F} \cup M$. In the following, we use the letter "G" to denote $\bar{F} \cup M$ unless stated otherwise. The closure of G under the set of rules {MVD0, MVD1, MVD2, MVD3} is denoted by G^+. Note that this set of rules is complete for the family of MVD's.

The following proposition has been shown in [3] and [10].

PROPOSITION 2. *An MVD m is in $(F \cup M)^+$ if and only if m is in G^+.*

Let Q be a family of some subsets of U that is closed under Boolean operations. (In this paper, for a subset X of U, the complement of X means $U - X$ and is sometimes denoted by X^c.) Q contains a unique subfamily, denoted by $\hat{Q}$, with the following properties:

(a) The sets in $\hat{Q}$ are nonempty.

(b) The sets in $\hat{Q}$ are pairwise disjoint.

(c) Each set in Q is a union of some of sets in $\hat{Q}$.

That is, $\hat{Q}$ consists of all nonempty minimal sets in Q. The subfamily $\hat{Q}$ is called the *basis* of Q.

For a subset X of U, let

$$D(X) = \{Y \mid X \twoheadrightarrow Y(U) \in (F \cup M)^+\}.$$

By Proposition 2, $D(X)$ is equal to $\{Y|X \twoheadrightarrow Y(U) \in G^+\}$. $D(X)$ is closed under Boolean operations. Since for each A in X the singleton set $\{A\}$ is in $\hat{D}(X)$ by MVD1, we consider only the subfamily $\hat{D}(X)-\{\{A\}|A \in X\}$ of $\hat{D}(X)$. Let this subfamily be denoted by DEP(X). DEP(X) is called the *dependency basis* of X [2], [3], [9]. Note that DEP(X) is a partition of X^c.

3. Computation of dependency basis.

3.1. Algorithm and an example. In this section, an algorithm is presented for computing DEP(X). Suppose that we are given a set G of MVD's on U and a subset X of U. Let $G=\{S_1 \twoheadrightarrow T_1(U), \cdots, S_k \twoheadrightarrow T_k(U)\}$. An algorithm for computing DEP(X) is given in Fig. 1 (Algorithm 1). After Algorithm 1 terminates, the collection of all blocks $\{P_1, \cdots, P_l\}$ is equal to DEP (X). Note that $\{P_1, \cdots, P_l\}$ is a partition of X^c. Since the number l of blocks is at most $|X^c|$, Algorithm 1 always terminates.

ALGORITHM 1

```
    begin
1     P_1 ← X^c;
2     l ← 1;
3     for j ← 1 until l do
4       while there exists an MVD S_r ↠ T_r(U) in G such that S_r ∩ P_j = ∅ and ∅ ⊊ T_r ∩
          P_j ⊊ P_j do
          begin
5           l ← l + 1;
6           create a new block P_l;
7           P_j ← T_r ∩ P_j;
8           P_j ← P_j − P_l;
          end
    end
```

FIG. 1. *Algorithm for computing* DEP(X).

PROPOSITION 3. *Algorithm* 1 *correctly computes* DEP(X).

The proof will be given in Appendix A.

Now we present an example. Assume that we are given

$$U=\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\},$$

$$F=\{\{A_8, A_9\} \to \{A_1\}\},$$

$$M=\{\{A_1, A_5, A_6, A_8\} \twoheadrightarrow \{A_1, A_2, A_6\}(U),$$

$$\{A_3, A_6, A_9\} \twoheadrightarrow \{A_5, A_8\}(U),$$

$$\{A_1, A_9\} \twoheadrightarrow \{A_2, A_3, A_4, A_9\}(U),$$

$$\{A_1, A_8, A_9\} \twoheadrightarrow \{A_2\}(U)\} \quad \text{and} \quad X=\{A_8, A_9\}.$$

Then,

$$X^c=\{A_1, A_2, A_3, A_4, A_5, \quad A_6, A_7\} \quad \text{and}$$

$$\bar{F}=\{\{A_8, A_9\} \twoheadrightarrow \{A_1\}(U)\}, \quad \text{so that}$$

$$G=\{\text{rule 1: } \{A_8, A_9\} \twoheadrightarrow \{A_1\}(U)$$

rule 2: $\{A_1, A_5, A_6, A_8\} \twoheadrightarrow \{A_1, A_2, A_6\}(U)$,

rule 3: $\{A_3, A_6, A_9\} \twoheadrightarrow \{A_5, A_8\}(U)$,

rule 4: $\{A_1, A_9\} \twoheadrightarrow \{A_2, A_3, A_4, A_9\}(U)$,

rule 5: $\{A_1, A_8, A_9\} \twoheadrightarrow \{A_2\}(U)\}$.

X^c will be split successively as follows by using Algorithm 1.

First by lines 1–3 in Fig. 1, P_1, l and j are set as follows:

$$P_1 = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7\},$$

$$l = 1, \qquad j = 1.$$

Here l denotes the number of blocks and j denotes the block name under consideration. Suppose that the condition in line 4 is examined for MVD's in G in the order of rule 1, rule 2, $\cdots$. Since $j = 1$ now, the block P_1 is selected. For P_1, the MVD rule 1 in G satisfies the condition in line 4, since $\{A_8, A_9\} \cap P_1 = \varnothing$ ($\{A_8, A_9\}$ is the left-hand side of rule 1) and $\varnothing \subsetneq \{A_1\} \cap P_1 \subsetneq P_1$ ($\{A_1\}$ is the right-hand side of rule 1). Thus, by lines 5–8, l, P_2 and P_1 are set as follows:

$$l = 2,$$

$$P_2 = \{A_1\},$$

$$P_1 = \{A_2, A_3, A_4, A_5, A_6, A_7\}.$$

For P_1, the MVD rule 4 is the first rule satisfying the condition in line 4. (The MVD rule 5 also satisfies the condition.) Thus, by lines 5–8, l, P_3 and P_1 are set as follows:

$$l = 3,$$

$$P_3 = \{A_2, A_3, A_4\},$$

$$P_1 = \{A_5, A_6, A_7\}.$$

For P_1, no MVD in G satisfies the condition in line 4. Thus, by line 3, the value of j is incremented by one, that is, the value of j becomes 2. For P_2 ($= \{A_1\}$), no MVD in G satisfies the condition in line 4, and then the value of j becomes 3. For P_3 ($= \{A_2, A_3, A_4\}$), the MVD rule 2 is the first rule satisfying the condition in line 4. Thus, l, P_4 and P_3 are set as follows:

$$l = 4,$$

$$P_4 = \{A_2\},$$

$$P_3 = \{A_3, A_4\}.$$

For P_3, no MVD in G satisfies the condition in line 4. Then, the value of j becomes 4. For P_4, no MVD in G satisfies the condition in line 4. After that, the value of j becomes 5 and exceeds the number of blocks l. Then, Algorithm 1 terminates. Finally we obtain

$$\begin{aligned} \mathrm{DEP}(X) &= \{P_1, P_2, P_3, P_4\} \\ &= \{\{A_5, A_6, A_7\}, \{A_1\}, \{A_3, A_4\}, \{A_2\}\}. \end{aligned}$$

3.2. Detailed algorithm and its time complexity. We consider in detail the implementation of Algorithm 1 in order to show its running time to be

$$O\left(\max\left\{k|U|, \sum_{r=1}^{k} \sum_{i=1}^{k} |(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|\right\}\right),$$

where k is the number of MVD's in G. (Time complexity is measured according to the uniform cost criterion.) The key of the timing argument is to show how line 4 in Fig. 1 can be executed in $O(k)$ time and how lines 5–8 in Fig. 1 can be executed in

$$O\Big(\sum_{i=1}^{k} |(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c| + k\Big) \text{ time.}$$

Let $U = \{A_1, \cdots, A_N\}$. For a given subset X of U, we can find X^c (as a list with increasing order) in $O(|U| + |X|)$ time. For simplicity, let $X^c = \{C_1, \cdots, C_n\}$. From now on, n denotes the number of attributes in X^c. For two subsets S and T of U and a subset P of X^c, predicate "$S \cap P = \varnothing$ and $\varnothing \subsetneq T \cap P \subsetneq P$" is true if and only if predicate "$(S \cap X^c) \cap P = \varnothing$ and $\varnothing \subsetneq (T \cap S^c \cap X^c) \cap P \subsetneq P$" is true. And, whenever the predicate "$S \cap P = \varnothing$ and $\varnothing \subsetneq T \cap P \subsetneq P$" is true, the set $T \cap P$ is equal to $(T \cap S^c \cap X^c) \cap P$. Thus, for each MVD $S_r \twoheadrightarrow T_r(U)$ in G, we can use $S_r \cap X^c$ and $T_r \cap S_r^c \cap X^c$ instead of S_r and T_r, respectively, in the implementation. In the following, for simplicity, $S_r \cap X^c$ and $T_r \cap S_r^c \cap X^c$ are sometimes denoted by S_r' and T_r', respectively. Each attribute C_m in X^c $(1 \leqq m \leqq n)$ is treated by its suffix m (call it attribute-number). Each MVD $S_i \twoheadrightarrow T_i(U)$ in G $(1 \leqq i \leqq k)$ is also treated by its suffix i (call it rule-number).

(1) We use $2n$ linked lists, $S\_\mathrm{LIST}(1), \cdots, S\_\mathrm{LIST}(n)$, $T\_\mathrm{LIST}(1), \cdots, T\_\mathrm{LIST}(n)$. For each attribute-number m $(1 \leqq m \leqq n)$, $S\_\mathrm{LIST}(m)$ consists of rule-numbers i such that C_m belongs to S_i' and $T\_\mathrm{LIST}(m)$ consists of rule-numbers i such that C_m belongs to T_i'.

In order to execute efficiently line 4 and lines 5–8 in Fig. 1, we use the following lists, array and matrices.

(2) We use l doubly linked lists, $P\_\mathrm{LIST}(1), \cdots, P\_\mathrm{LIST}(l)$, to represent a current partition $\{P_1, \cdots, P_l\}$. $P\_\mathrm{LIST}(j)$ $(1 \leqq j \leqq l)$ consists of attribute-numbers m such that P_j contains C_m.

(3) For each rule-number i $(1 \leqq i \leqq k)$, we use l doubly linked lists, $TP\_\mathrm{LIST}(i, 1), \cdots, TP\_\mathrm{LIST}(i, l)$, to represent $T_i' \cap P_1, \cdots, T_i' \cap P_l$ for a current partition $\{P_1, \cdots, P_l\}$. $TP\_\mathrm{LIST}(i, j)$ $(1 \leqq i \leqq k, 1 \leqq j \leqq l)$ consists of attribute-numbers m such that $T_i' \cap P_j$ contains C_m.

We assume that, given an attribute-number m, we can access in a constant time the "cell" corresponding to m in $P\_\mathrm{LIST}(j)$ and $TP\_\mathrm{LIST}(i, j)$ $(1 \leqq i \leqq k, 1 \leqq j \leqq l)$, respectively, if m belongs to those lists. Thus, given an attribute-number m, deletion of m from such a doubly linked list can be executed in a constant time. A list having such a mechanism can be implemented by using a matrix whose column corresponds to an attribute-number. Let every list have its tail with a special rule-number "0" or a special attribute-number "0".

(4) To represent the numbers of elements in P_j $(1 \leqq j \leqq l)$, $S_i' \cap P_j$ and $T_i' \cap P_j$ $(1 \leqq i \leqq k, 1 \leqq j \leqq l)$, we use an array P_SIZE of length n and two $k \times n$ matrices, SP_SIZE and TP_SIZE, respectively. Assume that a current partition is $\{P_1, \cdots, P_l\}$. Then, for each i and j $(1 \leqq i \leqq k, 1 \leqq j \leqq l)$, $P\_\mathrm{SIZE}[j]$, $SP\_\mathrm{SIZE}[i, j]$ and $TP\_\mathrm{SIZE}[i, j]$ give the numbers of elements in P_j, $S_i' \cap P_j$ and $T_i' \cap P_j$, respectively. For each i and j $(1 \leqq i \leqq k, l+1 \leqq j \leqq n)$, let $P\_\mathrm{SIZE}[j] = 0$, $SP\_\mathrm{SIZE}[i, j] = 0$ and $TP\_\mathrm{SIZE}[i, j] = 0$.

For the above-mentioned data structure, we must do the intialization. To construct $2n$ lists, $S\_\mathrm{LIST}(1), \cdots, S\_\mathrm{LIST}(n)$, $T\_\mathrm{LIST}(1), \cdots, T\_\mathrm{LIST}(n)$, in (1) described above, we first construct two $k \times |U|$ bit matrices, S and T, where the ith row of S (or T) has "1" in each column corresponding to the attribute in S_i' (or T_i'). This can be done in $O(k|U|)$ time.

Next, for each attribute C_m in X^c $(1 \leqq m \leqq n)$, we construct $S\_LIST(m)$ (or $T\_LIST(m)$) by traversing the column of S (or T) corresponding to C_m. Thus, all of such $2n$ lists in (1) can be constructed in $O(\max\{k|U|, kn\})$ time. $P_1(=X^c)$ is the only block in the initial partition, so we can initialize all the lists, array and matrices in (2), (3) and (4) described above by traversing $2n$ lists, $S\_LIST(1), \cdots, S\_LIST(n)$, $T\_LIST(1), \cdots, T\_LIST(n)$ only once. Thus, initialization for all the lists, array and matrices in (2), (3) and (4) can be done in $O(kn)$ time. Therefore, the initialization steps can be done in $O(\max\{k|U|, kn\})$, that is, $O(k|U|)$ time.

```
    begin
1     j ← 1;
2     for j ← 1 until l do
        begin
3         SEARCH (j, R, DIVIDABLE);
4         while DIVIDABLE = "yes" do
            begin
5             l ← l + 1;
6             P_DIVIDE (j, l, R);
7             TP_DIVIDE (j, l);
8             UPDATE (j, l, R);
9             SEARCH (j, R, DIVIDABLE);
            end
        end
    end
```

FIG. 2. *Program for computing* DEP(X).

Next, we shall show a program for computing DEP(X). The procedure corresponding to Fig. 1 is shown in Fig. 2. SEARCH(j, R, DIVIDABLE) at lines 3 and 9 decides whether there is an MVD $S_r \twoheadrightarrow T_r(U)$ in G satisfying the conditions shown in line 4 in Fig. 1, and if there exists such an MVD $S_r \twoheadrightarrow T_r(U)$ in G, then it returns "yes" and r by DIVIDABLE and R, respectively. By $P\_DIVIDE(j, l, R)$ at line 6 in Fig. 2, we split the old P_j into the new P_j and P_l. This corresponds to lines 6–8 in Fig. 1. By $TP\_DIVIDE(j, l)$ at line 7 in Fig. 2, for each i $(1 \leqq i \leqq k)$, we split the old $T'_i \cap P_j$ into the new $T'_i \cap P_j$ and $T'_i \cap P_l$, since the old P_j was split into the new P_j and P_l. By UPDATE(j, l, R) at line 8 in Fig. 2, we update the sizes of sets changed or created by $P\_DIVIDE(j, l, R)$ and $TP\_DIVIDE(j, l)$, that is, sets P_j, P_l, $S'_i \cap P_j$, $S'_i \cap P_l$, $T'_i \cap P_j$ and $T'_i \cap P_l$ $(1 \leqq i \leqq k)$.

The details of the procedure SEARCH are given in Fig. 3. The condition "$S_r \cap P_j = \varnothing$ and $\varnothing \subsetneq T_r \cap P_j \subsetneq P_j$" is replaced by the condition "$|S'_r \cap P_j| = 0$ and $0 < |T'_r \cap P_j| < |P_j|$", and examined at line 4 by using SP_SIZE, TP_SIZE and P_SIZE. Since lines 3–7 are executed at most k times and a constant time is needed at each line, SEARCH terminates in $O(k)$ time.

The details of the procedure P_DIVIDE are given in Fig. 4. By this procedure, P_j is split into two blocks $T'_r \cap P_j$ and $P_j - (T'_r \cap P_j)$, and then the blocks $T'_r \cap P_j$ and $P_j - (T'_r \cap P_j)$ are named new P_l and P_j, respectively. The doubly linked list representing $P_j - (T'_r \cap P_j)$ can be obtained by deleting each element in $T'_r \cap P_j$ from the doubly linked list representing the old P_j. This step, as illustrated in Fig. 5, can be executed by traversing the list representing $T'_r \cap P_j$ only once. Deletion of a particular attribute-number m from a doubly-linked list can be executed in a constant time. Therefore, P_DIVIDE can be executed in $O(|T'_r \cap P_j|)$ time.

```
      procedure SEARCH (j, R, DIVIDABLE);
        begin
1         DIVIDABLE ← "no";
2         i ← 1;
3         while DIVIDABLE = "no" and i ≦ k do
4           if SP_SIZE [i, j] = 0 and 0 < TP_SIZE [i, j] < P_SIZE [j]
              then begin
5                    DIVIDABLE ← "yes";
6                    R ← i;
                   end
7             else i ← i + 1;
8         return R and DIVIDABLE;
      end
```

FIG. 3. *Procedure SEARCH.*

```
    procedure P_DIVIDE(j, l, R);
      begin
1       create a new list P_LIST(l) (newly created list is empty);
2       m ← first attribute-number of TP_LIST(R, j);
3       while m ≠ 0 (m = 0 means that all elements in TP_LIST(R, j) were processed)
          do begin
4           delete m from P_LIST(j);
5           add m to P_LIST(l);
6           m ← succeeding attribute-number of TP_LIST(R, j);
          end
7       add 0 to the tail of P_LIST(l);
        return
      end
```

FIG. 4. *Procedure P_DIVIDE.*

The details of the procedure *TP*_DIVIDE are given in Fig. 6. Since the old P_j was split into the new P_j and P_l, the old $T'_i \cap P_j$ ($1 \leqq i \leqq k$) must be split into the new $T'_i \cap P_j$ and $T'_i \cap P_l$. The doubly linked list representing the new $T'_i \cap P_j$ can be obtained by deleting elements which are inserted into the new P_l (= the old $T'_r \cap P_j$) from the doubly linked list representing the old $T'_i \cap P_j$. The other doubly linked list representing the new $T'_i \cap P_l$ can be constructed by linking all elements which were deleted from the old $T'_i \cap P_j$. These steps can be executed as follows. For each attribute C_m in the new P_l, we delete the attribute-number m from *TP*_LIST(i, j) and add m to *TP*_LIST(i, l) for each rule-number i in *T*_LIST(m), as illustrated in Fig. 7. The execution of the procedure *TP*_DIVIDE can be done in $O(\sum_{i=1}^{k} |(T'_r \cap P_j) \cap (T'_i \cap P_j)| + k)$ time, that is, $O(\sum_{i=1}^{k} |T'_i \cap T'_r \cap P_j| + k)$ time, where P_j denotes the old block before being split.

The details of the procedure UPDATE are given in Fig. 8. For each attribute C_m in the new P_l (= the old $T'_r \cap P_j$), we can update *SP*_SIZE$[i_s, j]$ and *SP*_SIZE$[i_s, l]$ for each rule-number i_s in *S*_LIST(m) (or *TP*_SIZE$[i_t, j]$ and *TP*_SIZE$[i_t, l]$ for each rule-number i_t in *T*_LIST(m)) by traversing *S*_LIST(m) (or *T*_LIST(m)). Thus, UPDATE can be executed in $O(\sum_{i_s=1}^{k} |(T'_r \cap P_j) \cap (S'_{i_s} \cap P_j)| + \sum_{i_t=1}^{k} |(T'_r \cap P_j) \cap (T'_{i_t} \cap P_j)|)$ time, that is, $O(\sum_{i=1}^{k} |(S'_i \cup T'_i) \cap T'_r \cap P_j|)$ time (note that $S'_i \cap T'_i = \varnothing$), where P_j denotes the old block before being split.

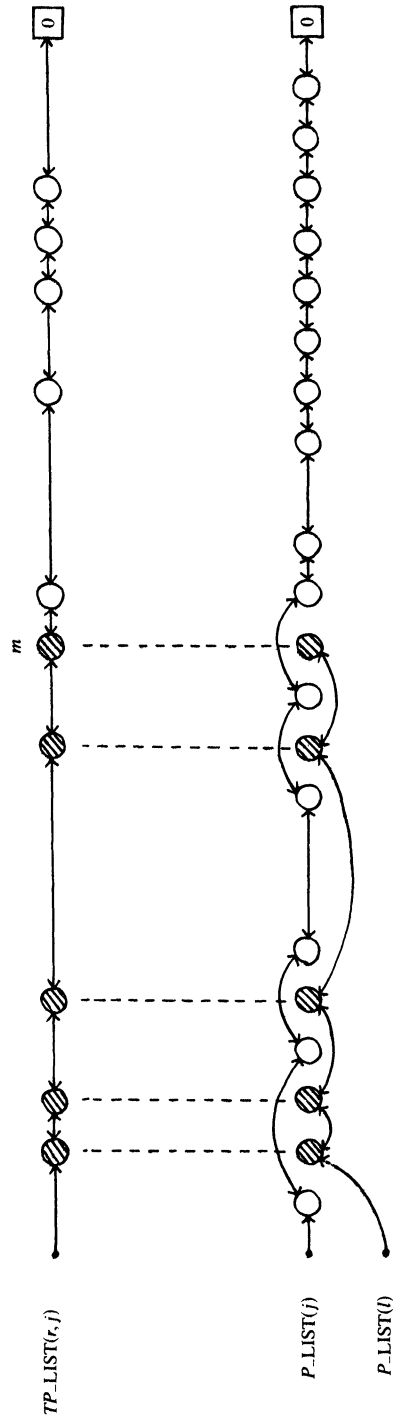


FIG. 5. *Splitting P_j into new P_j and P_l by $T'_r \cap P_j$.*

```
    procedure TP_DIVIDE(j, l);
      begin
1       for each i such that 1 ≦ i ≦ k, create a new list TP_LIST(i, l);
2       m ← first attribute-number of P_LIST(l);
3       while m ≠ 0 do
          begin
4           i ← first rule-number of T_LIST(m);
5           while i ≠ 0 do
              begin
6               delete m from TP_LIST(i, j);
7               add m to TP_LIST(i, l);
8               i ← succeeding rule-number of T_LIST(m);
              end
9           m ← succeeding attribute-number of P_LIST(l);
          end
10      for each i such that 1 ≦ i ≦ k, add 0 to the tail of TP_LIST(i, l);
11      return
      end
```

FIG. 6. *Procedure TP_DIVIDE.*

Now, we show that the whole program in Fig. 2 terminates in $O(\max\{k|X^c|, \sum_{r=1}^{k}\sum_{i=1}^{k}|(T_i \cup S_i) \cap T_r \cap S_r^c \cap X^c|\})$ time. From the above-mentioned discussion, when $R = r$, lines 5–9 in Fig. 2 can be executed in $O(|T_r' \cap P_j|) + O(\sum_{i=1}^{k}|T_i' \cap T_r' \cap P_j| + k) + O(\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_r' \cap P_j|) + O(k)$ time, that is, $O(\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_r' \cap P_j|) + O(k)$ time, where P_j denotes the old block before being split. Lines 5–9 in Fig. 2 are executed at most $|X^c|$ times, thus the total time by the second factor $O(k)$ is $O(k|X^c|)$. In the following, we estimate the total time by the first factor $O(\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_r' \cap P_j|)$.

Once some block P_j is split into two blocks $T_r' \cap P_j$ and $P_j - (T_r' \cap P_j)$ by using an MVD $S_r \twoheadrightarrow T_r(U)$ in G, then any block P which is a subset of P_j cannot be split by using the same MVD $S_r \twoheadrightarrow T_r(U)$ in G, since, for such a block P, either $P \subseteq T_r' \cap P_j$ or $P \subseteq P_j - (T_2' \cap P_j)$ holds and then such a block P cannot satisfy the condition $\emptyset \subsetneq T_r' \cap P \subsetneq P$. Thus, for a fixed r $(1 \leqq r \leqq k)$, any two blocks which are split by using the MVD $S_r \twoheadrightarrow T_r(U)$ must be disjoint. Therefore, the sum of the values $\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_r' \cap P_j|$ for all blocks P_j which are split by using the MVD $S_r \twoheadrightarrow T_r(U)$ is bounded by $\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_r'|$. Hence, the total time spent at lines 5–9 in Fig. 2 is the sum of $O(k|X^c|)$ and $O(\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_1'|), \cdots, O(\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_k'|)$, that is, $O(k|X^c| + \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i' \cup T_i') \cap T_r'|)$. Note that each execution of line 3 in Fig. 2 takes $O(k)$ time and line 3 is executed at most $|X^c|$ times. Then, the program terminates in $O(k|X^c| + \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|)$ time $((S_i' \cup T_i') \cap T_r' = (S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c)$. Considering the time needed in the initialization steps, we have the following theorem.

THEOREM 1. *For a set of MVD's* $G = \{S_1 \twoheadrightarrow T_1(U), \cdots, S_k \twoheadrightarrow T_k(U)\}$ *and a subset* X *of* U, *the dependency basis of* X, DEP(X), *can be computed in* $O(\max\{k|U|, \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|\})$ *time.*

It is known that MVD $X \twoheadrightarrow Y(U)$ in G^+ if and only if Y is a union of some of sets in DEP(X). It is obvious that it can be decided in $O(\max\{k|U|, \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|\})$ time whether or not Y is a union of some of sets in DEP(X). Therefore, the following corollary follows from Theorem 1.

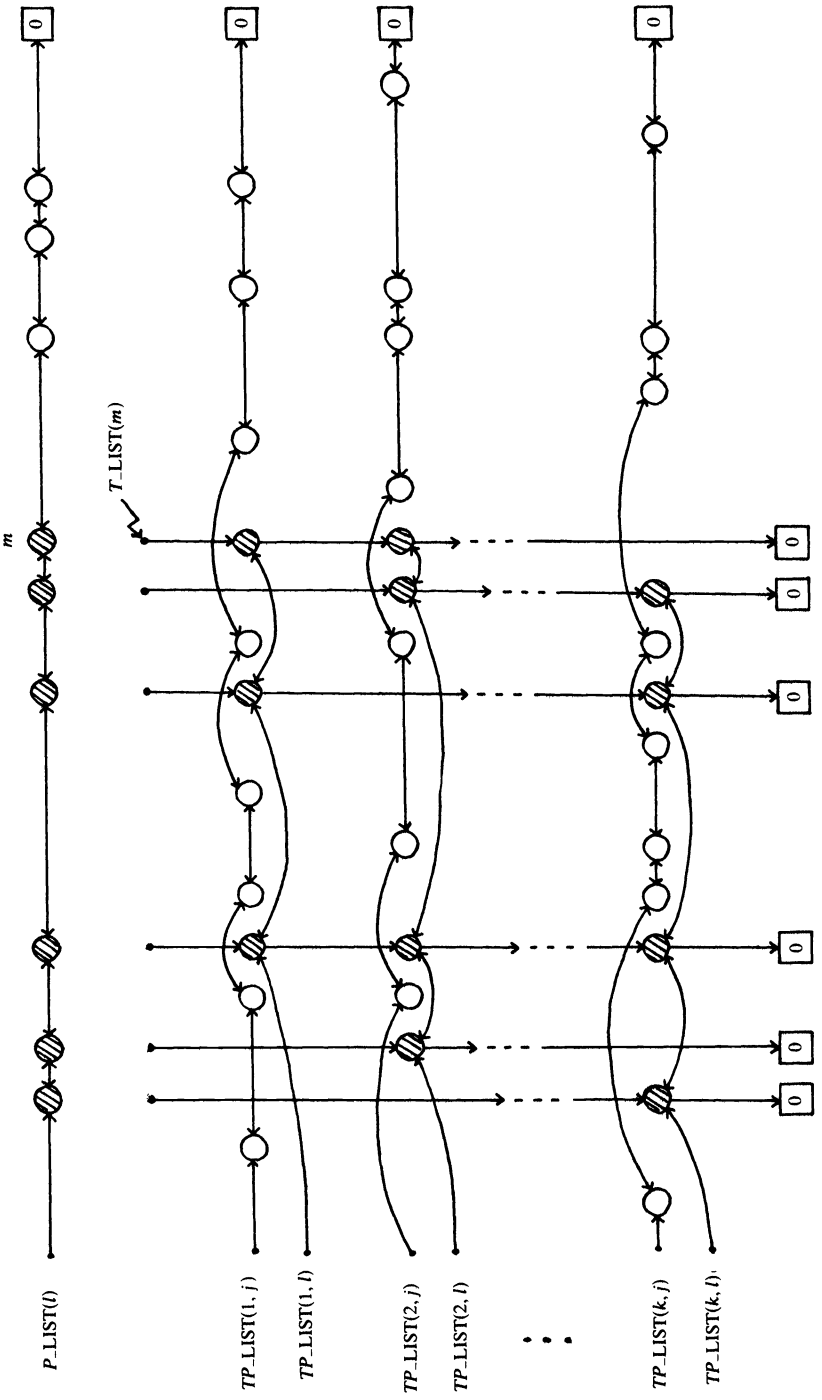


FIG. 7. *Splitting* $T_i' \cap P_j$ $(1 \leqq i \leqq k)$ *into new* $T_i' \cap P_j$ *and* $T_i' \cap P_l$ *by new* P_l.

```
    procedure UPDATE(j, l, R);
      begin
1       P_SIZE[l] ← TP_SIZE[R, j];
2       P_SIZE[j] ← P_SIZE[j] − P_SIZE[l];
3       m ← first attribute-number of P_LIST(l);
4       while m ≠ 0 do
          begin
5           i_s ← first rule-number of S_LIST(m);
6           while i_s ≠ 0 do
              begin
7               SP_SIZE[i_s, j] ← SP_SIZE[i_s, j] − 1;
8               SP_SIZE[i_s, l] ← SP_SIZE[i_s, l] + 1;
9               i_s ← succeeding rule-number of S_LIST(m);
              end
10          i_t ← first rule-number of T_LIST(m);
11          while i_t ≠ 0 do
              begin
12              TP_SIZE[i_t, j] ← TP_SIZE[i_t, j] − 1;
13              TP_SIZE[i_t, l] ← TP_SIZE[i_t, l] + 1;
14              i_t ← succeeding rule-number of T_LIST(m);
              end
15          m ← succeeding attribute-number of P_LIST(l);
          end
16      return
      end
```

FIG. 8. *Procedure UPDATE.*

COROLLARY 1. *For a set of FD's* $F = \{S_1 \to T_1, \cdots, S_l \to T_l\}$, *where each* T_j $(1 \leqq j \leqq l)$ *is a singleton set, and a set of MVD's* $M = \{S_{l+1} \twoheadrightarrow T_{l+1}(U), \cdots, S_k \twoheadrightarrow T_k(U)\}$, *it can be decided in* $O(\max\{k|U|, \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|\})$ *time whether or not a given MVD* $X \twoheadrightarrow Y(U)$ *is derived (by a complete set of inference rules) from F and M.*

Furthermore, from Theorem 1 and the results of [3], we have the following corollary.

COROLLARY 2. *For F and M as described in Corollary* 1, *it can be decided in* $O(\max\{k|U|, \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|\})$ *time whether or not a given FD* $X \to Y$ *is derived (by a complete set of inference rules) from F and M.*

Note that $O(\max\{k|U|, \sum_{r=1}^{k}\sum_{i=1}^{k}|(S_i \cup T_i) \cap T_r \cap S_r^c \cap X^c|\})$ is bounded above by $O(\min\{k^2|U|, \|F \cup M\|^2\})$ where $\|F \cup M\|$ is the size of description of F and M.

4. Decision problem concerning nontrivial MVD.

4.1. Problem and key lemma. For a set G, defined in § 2.3, of MVD's on a set U of attributes and for a subset U' of U, define

$$\text{PROJ}(G^+, U') = \{X \twoheadrightarrow Y \cap U'(U') | X \twoheadrightarrow Y(U) \in G^+ \wedge X \subseteq U'\}.$$

If all MVD's in G (and hence all MVD's in G^+) are valid in a relation $R(U)$, then all MVD's in $\text{PROJ}(G^+, U')$ are valid in the projection $R[U']$, [9].

An MVD $X \twoheadrightarrow Y(U')$ is called a *nontrivial MVD* if Y is a nonempty proper subset of $U' - X$, [9].

In this section, we consider the following decision problem, which is useful for decomposing a relation schema into a Fourth Normal Form family, [9].

"Given a set G of MVD's on U and a subset U' of U, decide whether or not there exists a nontrivial MVD in $\text{PROJ}(G^+, U')$."

ALGORITHM 2

```
          begin
step ①  1   compute π' from G and U';
step ②  2   for each B ∈ π' such that |B| ≧ 2
        3     do begin P1 ← U − (U' − B);
        4           while there exists an MVD S →→ T(U) in G such that S ∩ P1
                                                  = ∅ and ∅ ⊊ T ∩ P1 ⊊ P1
        5             do begin P2 ← T ∩ P1;
        6                  P1 ← P1 − P2;
        7                  if B ⊆ P1 or B ⊆ P2
                             then if B ⊆ P2
                                    then P1 ← P2
       10                  else return "yes";
                           end
                end
step ③ 11   for each distinct two sets B1, B2 ∈ π'
       12     do begin P1 ← U − (U' − B1 − B2);
       13           while there exists an MVD S →→ T(U) in G such that S ∩ P1
                                                  = ∅ and ∅ ⊊ T ∩ P1 ⊊ P1
       14             do begin P2 ← T ∩ P1;
       15                      P1 ← P1 − P2;
       16                      if (B1 ∪ B2) ⊆ P1 or (B1 ∪ B2) ⊆ P2
       17                        then if (B1 ∪ B2) ⊆ P2
       18                               then P1 ← P2
       19                      else return "yes";
                           end
                end
step ④ 20   return "no";
        end
```

FIG. 9. *Algorithm for deciding whether or not there exists a nontrivial MVD in* PROJ(G^+, U').

For G, let π denote the basis of the Boolean closure of the family $\{S | S \twoheadrightarrow T(U) \in G\}$, where complementation is relative to U. For π and a subset U' of U, define

$$\pi' = \{B \cap U' | B \in \pi \wedge B \cap U' \neq \emptyset\}.$$

Note that π (or π') is a partition of U (or of U', respectively).

The following is a key lemma for obtaining an efficient decision procedure for the problem above. The proof is given in Appendix B.

LEMMA. *If there exists a nontrivial MVD in* PROJ(G^+, U'), *then there exists a nontrivial MVD* $X \twoheadrightarrow Y(U')$ *in* PROJ(G^+, U') *satisfying either*

(1) $X = U' - B$ *and* $\emptyset \subsetneq Y \subsetneq B$, *where* $B \in \pi'$, *or*

(2) $X = U' - B_1 - B_2$ *and* $Y = B_2$, *where* $B_1, B_2 \in \pi'$ *and* $B_1 \neq B_2$.

4.2. Decision algorithm and an example. The lemma in § 4.1 yields the algorithm shown in Fig. 9 for deciding, given a set of MVD's on U and a subset U' of U, whether or not there exists a nontrivial MVD in PROJ(G^+, U'). Step ① in Algorithm 2 computes π' from given G and U'. Step ② determines whether or not there exists a nontrivial MVD in PROJ(G^+, U') satisfying condition (1) in the lemma. If condition (1) is satisfied, then "yes" is returned. Otherwise, step ③ is executed which determines whether or not there exists a nontrivial MVD in PROJ(G^+, U') satisfying condition (2). If condition (2) is satisfied, then "yes" is returned. Otherwise, "no" is returned.

Now we present an example (this example is taken from [9]). Assume that we are given

$$U = \{\text{CLASS, SECTION, STUDENT, MAJOR, EXAM, YEAR, INSTRUCTOR, RANK, SALARY, TEXT, DAY, ROOM}\},$$

$$F = \{\{\text{CLASS, SECTION}\} \to \{\text{INSTRUCTOR}\},$$
$$\{\text{CLASS, SECTION, DAY}\} \to \{\text{ROOM}\},$$
$$\{\text{STUDENT}\} \to \{\text{MAJOR}\},$$
$$\{\text{STUDENT}\} \to \{\text{YEAR}\},$$
$$\{\text{INSTRUCTOR}\} \to \{\text{RANK}\},$$
$$\{\text{INSTRUCTOR}\} \to \{\text{SALARY}\}\},$$

$$M = \{\{\text{CLASS, SECTION}\} \twoheadrightarrow \{\text{STUDENT, MAJOR, EXAM, YEAR}\}(U),$$
$$\{\text{CLASS, SECTION}\} \twoheadrightarrow \{\text{INSTRUCTOR, RANK, SALARY}\}(U),$$
$$\{\text{CLASS, SECTION}\} \twoheadrightarrow \{\text{TEXT}\}(U),$$
$$\{\text{CLASS, SECTION}\} \twoheadrightarrow \{\text{DAY, ROOM}\}(U),$$
$$\{\text{CLASS}\} \twoheadrightarrow \{\text{TEXT}\}(U),$$
$$\{\text{CLASS, SECTION, STUDENT}\} \twoheadrightarrow \{\text{EXAM}\}(U)\}, \quad \text{and}$$

$$U' = \{\text{CLASS, SECTION, RANK, SALARY}\}.$$

In this case, we have that $G = \bar{F} \cup M$, where

$$\bar{F} = \{\{\text{CLASS, SECTION}\} \twoheadrightarrow \{\text{INSTRUCTOR}\}(U),$$
$$\{\text{CLASS, SECTION, DAY}\} \twoheadrightarrow \{\text{ROOM}\}(U),$$
$$\{\text{STUDENT}\} \twoheadrightarrow \{\text{MAJOR}\}(U),$$
$$\{\text{STUDENT}\} \twoheadrightarrow \{\text{YEAR}\}(U),$$
$$\{\text{INSTRUCTOR}\} \twoheadrightarrow \{\text{RANK}\}(U),$$
$$\{\text{INSTRUCTOR}\} \twoheadrightarrow \{\text{SALARY}\}(U)\}, \quad \text{and}$$

$$\pi = \{\{\text{CLASS}\}, \{\text{SECTION}\}, \{\text{STUDENT}\}, \{\text{INSTRUCTOR}\}, \{\text{DAY}\},$$
$$\{\text{MAJOR, EXAM, YEAR, RANK, SALARY, TEXT, ROOM}\}\}.$$

It follows from U' and π that

$$\pi' = \{\{\text{CLASS}\}, \{\text{SECTION}\}, \{\text{RANK, SALARY}\}\}.$$

{RANK, SALARY} is the only block B in π' such that $|B| \geqq 2$. For this block B, lines 3–10 of Algorithm 2 are executed. In line 3, P_1 is set as follows:

$$P_1 = \{\text{STUDENT, MAJOR, EXAM, YEAR, INSTRUCTOR, RANK, SALARY, TEXT, DAY, ROOM}\}.$$

MVD {CLASS, SECTION}$\twoheadrightarrow${INSTRUCTOR}(U) satisfies conditions {CLASS, SECTION}$\cap P_1 = \varnothing$ and $\varnothing \subsetneq \{\text{INSTRUCTOR}\} \cap P_1 \subsetneq P_1$. Then P_2 and P_1 are set at lines 5 and 6, respectively, as follows:

$$P_2 = \{\text{INSTRUCTOR}\}$$

$$P_1 = \{\text{STUDENT, MAJOR, EXAM, YEAR, RANK, SALARY, TEXT, DAY, ROOM}\}.$$

Since $B \subseteq P_1$, the while-statement in line 4 is executed again for this new P_1. This time, by using MVD {INSTRUCTOR}$\twoheadrightarrow${RANK}(U), P_1 is split as follows:

$$P_2 = \{\text{RANK}\}$$

$$P_1 = \{\text{STUDENT, MAJOR, EXAM, YEAR, SALARY, TEXT, DAY, ROOM}\}.$$

Since $B \not\subseteq P_1$ and $B \not\subseteq P_2$, "yes" is returned in line 10, that is, there exists a nontrivial MVD in PROJ(G^+, U').

The lemma in § 4.1 shows that in this example at least nontrivial MVD's {CLASS, SECTION}$\twoheadrightarrow${SALARY}(U') and {CLASS, SECTION}$\twoheadrightarrow${RANK}(U') are in PROJ(G^+, U').

4.3. Time complexity of Algorithm 2. We will estimate the time complexity of Algorithm 2. As stated in § 3.2, the time complexity of computing DEP(U') for a subset U' of U is bounded above by $O(\min\{k^2|U|, \|F \cup M\|^2\})$ where k and $|U|$ are the numbers of dependencies in $F \cup M$ and attributes in U, respectively, and $\|F \cup M\|$ is the size of description of F and M.

Step ① requires $O(k|U|)$ time [3]. Let i be the number of sets in π'.

For a block B in π', the do-statement in line 3, which is a modification of the algorithm shown in Fig. 1, determines whether or not there exist at least two sets in DEP($U' - B$) that intersect B, that is, whether there is no set in DEP($U' - B$) of which B is a subset. Almost the same discussion in § 3.2 assures that the do-statement in line 3 can be executed within $O(\min\{k^2|U|, \|F \cup M\|^2\})$ time. Since the loop of lines 2–10 is repeated at most i times, step ② requires at most $O(i \cdot \min\{k^2|U|, \|F \cup M\|^2\})$ time.

Similarly, step ③ requires at most $O(i^2 \cdot \min\{k^2|U|, \|F \cup M\|^2\})$ time since the loop of lines 11–19 is repeated at most $i(i-1)/2$ times.

Step ④ is done in a constant time. Thus, we have the following theorem.

THEOREM 2. *For sets F of FD's and M of MVD's on U and for a subset U' of U, Algorithm 2 determines within $O(i^2 \cdot \min\{k^2|U|, \|F \cup M\|^2\})$ time whether or not there exists a nontrivial MVD in PROJ(G^+, U') where i is the number of blocks B in the basis of the Boolean closure of all left-hand sides of dependencies in $F \cup M$ such that $B \cap U' \neq \phi$, k and $|U|$ are the numbers of dependencies in $F \cup M$ and attributes in U, respectively, and $\|F \cup M\|$ is the size of the description of F and M.*

Note that $i \leqq |U'| \leqq |U|$.

Appendix A. Proof of Proposition 3. It was proved in [10] that Algorithm 1 correctly computes DEP(X) by using derivation trees. In the following, another proof will be shown.

It is proved in [3] that, if a partitioning algorithm satisfies the following conditions, then it correctly computes DEP(X).

(a) For each block P_i in the final partition, $X \twoheadrightarrow P_i(U)$ is in G^+.

(b) For the final partition $\{P_1, \cdots, P_l\}$ and each MVD $S \twoheadrightarrow T(U)$ in G, $T \cap Y^c$ is either empty or a union of some of P_i's, where Y is the union of X and the sets from $\{P_1, \cdots, P_l\}$ that intersect $S \cap X^c$.

We show that condition (a) is satisfied by using induction on the number of executions through lines 5–8 in Fig. 1. X^c is the only block in the initial partition of Algorithm 1 and $X \twoheadrightarrow X^c(U)$ is in G^+ by MVD0 and MVD1. Assume that, for each block P in a current partition, $X \twoheadrightarrow P(U)$ is in G^+. Then, the newly obtained blocks by one pass of Algorithm 1 are $T_r \cap P_j$ and $P_j - (T_r \cap P_j)$, where MVD $S_r \twoheadrightarrow T_r(U)$ is in G, $S_r \cap P_j = \varnothing$ and $\varnothing \subsetneq T_r \cap P_j \subsetneq P_j$. By the assumption, $X \twoheadrightarrow P_j(U)$ is in G^+. By MVD0, $X \twoheadrightarrow P_j^c(U)$ is in G^+. By applying MVD2 to MVD $S_r \twoheadrightarrow T_r(U)$, $P_j^c \twoheadrightarrow T_r(U)$ is in G^+, since $S_r \cap P_j = \varnothing$ implies $S_r \subseteq P_j^c$. Then, by MVD3, $X \twoheadrightarrow T_r \cap P_j(U)$ is in $G^+(T_r \cap (P_j^c)^c = T_r \cap P_j)$. By MVD6, $X \twoheadrightarrow P_j - (T_r \cap P_j)(U)$ is in G^+. Thus condition (a) is satisfied. Next, we show that Algorithm 1 satisfies condition (b). Line 4 in Fig. 1 shows that, if at least one of three conditions $S_r \cap P_j \neq \varnothing$, $T_r \cap P_j = \varnothing$ and $T_r \cap P_j = P_j$ holds for block P_j and every MVD $S_r \twoheadrightarrow T_r(U)$ in G, then block P_j is no longer split, that is, block P_j is in the final partition. Furthermore by the construction of the final partition by Algorithm 1, the final partition consists of only such blocks. Thus, for each MVD

$S \twoheadrightarrow T(U)$ in G and each block P in the final partition, either $S \cap P \neq \emptyset$, $T \cap P = \emptyset$ or $T \cap P = P$ holds, that is, if $S \cap P = \emptyset$, then either $T \cap P = \emptyset$ or $T \cap P = P$ holds. Therefore, condition (b) is satisfied.

Appendix B. Proof of the lemma. The following Proposition 4 and Proposition 5 are used to prove the lemma.

PROPOSITION 4. *If* $W \subseteq X \subseteq U'$ *and* $W \twoheadrightarrow V(U') \in \mathrm{PROJ}(G^+, U')$, *then* $X \twoheadrightarrow V - X(U')$ *is in* $\mathrm{PROJ}(G^+, U')$.

Proof. By the definition of $\mathrm{PROJ}(G^+, U')$, if $W \twoheadrightarrow V(U') \in \mathrm{PROJ}(G^+, U')$, then there is a subset Z of U such that $Z \cap U' = V$ and $W \twoheadrightarrow Z(U) \in G^+$. Since $W \subseteq X$, by using MVD2, MVD1 and MVD3, we have that

$$X \twoheadrightarrow Z - X(U) \in G^+. \tag{B.1}$$

Since $X \subseteq U'$ and $(Z - X) \cap U' = V - X$, it follows from (B.1) that

$$X \twoheadrightarrow V - X(U') \in \mathrm{PROJ}(G^+, U'). \qquad \square$$

PROPOSITION 5. *For a subset* X *of* U', *let* W *be the union of all sets in* $\{B \mid B \in \pi' \bigwedge B \subseteq X\}$. *If there exists a nontrivial MVD in* $\mathrm{PROJ}(G^+, U')$ *whose left-hand side is* X, *then there also exists a nontrivial MVD in* $\mathrm{PROJ}(G^+, U')$ *whose left-hand side is* W.

Proof. By the definition of W, we have

$$W \subseteq X \subseteq U'. \tag{B.2}$$

Suppose that there exists no nontrivial MVD in $\mathrm{PROJ}(G^+, U')$ whose left-hand side is W. Then it follows from the definition of $\mathrm{PROJ}(G^+, U')$ that for any subset Z of U such that $W \cap Z = \emptyset$ and $\emptyset \subsetneq Z \cap U' \subsetneq U' - W$, there exists no MVD $W \twoheadrightarrow Z(U)$ in G^+. Therefore, $U' - W$ is a subset of a set V belonging to $\mathrm{DEP}(W)$. That is, it holds that

$$U' - W \subseteq V, \tag{B.3}$$

$$W \cap V = \emptyset, \tag{B.4}$$

and

$$W \twoheadrightarrow V(U) \in G^+. \tag{B.5}$$

By (B.2), (B.3) and (B.4), we have that

$$V - X = V - (X - W), \tag{B.6}$$

$$(V - X) \cap U' = V \cap U' - X \cap U' = (U' - W) - X = U' - X. \tag{B.7}$$

Next we shall show that $V - X$ is a set in $\mathrm{DEP}(X)$. By (B.2), (B.5) and Proposition 4 (let $U' = U$), we have that

$$X \twoheadrightarrow V - X(U) \in G^+. \tag{B.8}$$

Note that $\mathrm{PROJ}(G^+, U) = G^+$. Let $S \twoheadrightarrow T(U)$ be an arbitrary MVD in G. Since $V \in \mathrm{DEP}(W)$, either (B.9) or (B.10) holds:

$$S \cap V \neq \emptyset. \tag{B.9}$$

$$T \cap V = \emptyset \quad \text{or} \quad V \subseteq T. \tag{B.10}$$

In the case where condition (B.9) holds, suppose $P \in \pi$, $P \subseteq S$ and $P \cap V \neq \emptyset$. There are two cases to consider. If P intersects $V - U'$ then P intersects $V - X$. Otherwise, P must intersect $V \cap U'$. Let P' be $P \cap U'$. If $P' \subseteq X$ then, by the definition of W, $P' \subseteq W$—a contradiction. Hence P' intersects $U' - X$, so P intersects $V - X$.

Therefore, we have that

(B.11) $$S \cap (V - X) \neq \emptyset.$$

In the case where condition (B.10) holds, since $V - X \subseteq V$ we obviously have that

(B.12) $$T \cap (V - X) = \emptyset \quad \text{or} \quad V - X \subseteq T.$$

It follows from (B.8), (B.11) and (B.12) that

(B.13) $$V - X \in \text{DEP}(X).$$

By (B.7) and (B.13), there will exist no nontrivial MVD in $\text{PROJ}(G^+, U')$ whose left-hand side is X. This contradicts the assumption of the proposition.

By Proposition 5, we can assume without loss of generality that

(B.14) $$W \twoheadrightarrow V(U') \in \text{PROJ}(G^+, U'),$$

where W is an empty set or a union of some of sets in π' and $\emptyset \subsetneq V \subsetneq U' - W$.

(i) Consider the case where there exists a set B in π' such that

(B.15) $$\emptyset \subsetneq B \cap V \subsetneq B.$$

It follows from the assumption for MVD $W \twoheadrightarrow V(U')$ that $B \subseteq U' - W$. Since $W \subseteq U'$, we have that $W \subseteq U' - B$. By (B.14), $V - (U' - B) = V \cap B$ and Proposition 4, we have that

$$U' - B \twoheadrightarrow V \cap B(U') \in \text{PROJ}(G^+, U').$$

By (B.15) this MVD is nontrivial. Therefore, condition (1) in the lemma is satisfied.

(ii) Consider the remaining case, where there exist at least two distinct sets B_1 and B_2 in π' such that $B_1 \subseteq U' - W - V$ and $B_2 \subseteq V$. Therefore, since $W \subsetneq U'$ and $V \subsetneq U'$, we have that

(B.16) $$W \subseteq U' - B_1 - B_2,$$

(B.17) $$V - (U' - B_1 - B_2) = V \cap (B_1 \cup B_2) = B_2.$$

By Proposition 4, (B.14), (B.16) and (B.17), we have that

$$U' - B_1 - B_2 \twoheadrightarrow B_2(U') \in \text{PROJ}(G^+, U').$$

Since $U' - ((U' - B_1 - B_2) \cup B_2) = B_1 \neq \emptyset$, this MVD is nontrivial. Thus, condition (2) in the lemma is satisfied. □

REFERENCES

[1] W. W. ARMSTRONG, *Dependency structures of database relationships*, Information Processing 74, J. L. Rosenfeld, ed., North-Holland, Amsterdam, 1974, pp. 580–583.

[2] CATRIEL BEERI, RONALD FAGIN AND JOHN H. HOWARD, *A complete axiomatization for functional and multivalued dependencies in database relations*, Proc. ACM SIGMOD Conf., D. C. P. Smith, ed., Toronto, Canada, 1977, pp. 47–61.

[3] CATRIEL BEERI, *On the membership problem for multivalued dependencies in relational databases*, ACM Trans. Database Syst., to appear.

[4a] CATRIEL BEERI AND PHILIP A. BERNSTEIN, *Computational problems regarding the design of normal form relational schemas*, ACM Trans. Database Syst., to appear.

[4b] PHILIP A. BERNSTEIN AND CATRIEL BEERI, *An algorithmic approach to normalization of relational database schemas*, Tech. Rep. CSRG-73, Computer System Research Group, Univ. of Toronto, Toronto, 1976.

[5] E. F. CODD, *A relational model for large shared data bases*, Comm. ACM, 13 (1970), pp. 377–387.

[6] ———, *Further normalization of the data base relational model*, Courant Computer Science Symposium 6, Data base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1971, pp. 65–98.

[7] ———, *Recent investigations in relational data base systems*, Information Processing 74, J. L. Rosenfeld, ed., North-Holland, Amsterdam, 1974, pp. 1017–1021.

[8] RONALD FAGIN, *Functional dependencies of a relational database and propositional logic*, IBM J. Res. Develop., 21(1977), pp. 534–544.

[9] ———, *Multivalued dependencies and a new normal form for relational databases*, ACM Trans. Database Syst., 2 (1977), pp. 262–278.

[10] KENICHI HAGIHARA, MINORU ITO, KENICHI TANIGUCHI AND TADAO KASAMI, *Decision problems for multivalued dependencies and fourth normal form in relational data base model*, Tech. Rep. of Languages and Automata Symposium, Nagoya (July 1977). (In Japanese.)

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0011 $01.00/0

BOTTLENECKS AND EDGE CONNECTIVITY IN UNSYMMETRICAL NETWORKS*

C. P. SCHNORR†

Abstract. Let $F_{u,v}$ be the maximal flow from u to v in a network $\mathcal{N} = (V, E, c)$. We construct the matrix $(\min\{F_{u,v}, F_{v,u}\} | u, v \in V)$ by solving $|V| \log 2|V|$ individual max-flow problems for $\mathcal{N}$. There is a tree network $\bar{\mathcal{N}} = (V, \bar{E}, \bar{c})$ that stores minimal cuts corresponding to $\min\{F_{u,v}, F_{v,u}\}$ for all u, v. $\bar{\mathcal{N}}$ can be constructed by solving $|V| \log 2|V|$ individual max flow problems for the given network which can be done within $O(|V|^4)$ steps using the Dinic–Karzanov algorithm. We design an algorithm that computes the edge connectivity k of a directed graph within $O(k \cdot |E| \cdot |V|)$ steps.

Key words. maximum flow, minimum cut, Gomory–Hu algorithm, multiterminal network flow, maximum flow matrix, edge connectivity

1. Introduction.

A *network* (V, E, c) consists of 1) a set V of vertices, 2) a set $E \subset V \times V$ of edges and 3) a function $c: E \to R_+$ which associates to each $e \in E$ a positive real number $c(e)$ called the *capacity* of e.

For $s, t \in V$, $s \neq t$ *a flow from* s *to* t *in* (V, E, c) is a function $f: E \to R_+$ such that

(f1) $\forall e \in E: 0 \leqq f(e) \leqq c(e)$

(f2) $\forall u \in V - \{s, t\}: \sum_{(u,v)\in E} f(u, v) = \sum_{(v,u)\in E} f(v, u)$

i.e. the outcoming flow equals the incoming flow at u.

The *value* $\Phi(f)$ of f is defined by

$$\Phi(f) := \sum_{(s,v)\in E} f(s, v) - \sum_{(v,s)\in E} f(v, s).$$

A flow f from s to t is called *maximal* if $\Phi(f) \geqq \Phi(g)$ for all flows g from s to t. Let $F_{s,t}$ be the *value of a maximal flow* from s to t. We define $F_{u,u} := \infty$.

An (s, t)-*cut* is a pair $(S, \bar{S})$ such that $S \subset V \wedge s \in S \wedge \bar{S} = V - S \wedge t \in \bar{S}$. One defines the *capacity* $c(S, \bar{S})$ of a cut $(S, \bar{S})$ as

$$c(S, \bar{S}) := \sum_{(u,v)\in S\times\bar{S}\cap E} c(u, v).$$

An (s, t)-cut $(S, \bar{S})$ is called *minimal* if $c(S, \bar{S}) \leqq c(A, \bar{A})$ for all (s, t)-cuts $(A, \bar{A})$.

Maximal flows are characterized by the fundamental theorem of Ford and Fulkerson:

THEOREM 1.1. *Let f be a maximal flow from s to t and let $(S, \bar{S})$ be a minimal (s, t)-cut; then $\Phi(f) = c(S, \bar{S})$.*

The problem of designing efficient algorithms which to a given network construct a maximal flow from s to t has been successfully attacked over many years. Now, the best-known algorithm is Karzanov's improvement [5] of Dinic's algorithm [1] which runs in $O(|V|^3)$ RAM-steps when additions of real numbers are counted as single steps.

Whereas the Dinic–Karzanov algorithm solves an individual max flow problem for a given network we are interested in the construction of the *max-flow matrix* $(F_{u,v} | u, v \in V)$ of a given network. Clearly the max-flow matrix can be constructed by solving $|V|(|V| - 1)$ individual max-flow problems for the given network. We shall obtain a significant reduction of this number. In § 2 we construct the matrix $\mathcal{B}$ which is defined as

$$\mathcal{B} := (\min\{F_{u,v}, F_{v,u}\} | u, v \in V)$$

* Received by the editors November 1, 1977.

† Fachbereich Mathematik, Universität Frankfurt, 6000 Frankfurt a.M., West Germany.

by solving $|V|\log 2|V|$ individual max-flow problems for the given network. By Theorem 1.1 $\min\{F_{u,v}, F_{v,u}\}$ equals the minimum capacity of a cut that separates u and v. These minimal cuts are called *bottlenecks*; thus $\mathcal{B}$ is the matrix of bottlenecks. This problem has already been solved for *symmetrical* networks, i.e. if $c(u, v) = c(v, u)$ for all $u, v \in V$. For symmetrical networks we have $F_{u,v} = F_{v,u}$ and the algorithm of Gomory and Hu [4] constructs the max-flow matrix by solving $|V|-1$ individual max-flow problems for the given network.

The algorithm of Gomory and Hu associates to any symmetrical network a tree network that stores minimal cuts for all pairs of nodes. In § 3 we extend this construction to unsymmetrical networks. We associate to a given network $\mathcal{N} = (V, E, c)$ a tree network $\bar{\mathcal{N}} = (V, \bar{E}, \bar{c})$ such that (1), (2) hold for all $u, v \in V$. (1) $\min\{F_{u,v}, F_{v,u}\}$ equals the minimal capacity of the edges on the unique undirected path that connects u and v in $\bar{\mathcal{N}}$. (A sequence of edges is called an *undirected path* if it is possible to change the orientation of the edges such that the sequence forms a path).

(2) if among the minimal capacity edges on the path connecting u and v in $\bar{\mathcal{N}}$ some edge e is directed from u to v, then $F_{u,v} = \min\{F_{u,v}, F_{v,u}\}$ and the weak components of $\bar{\mathcal{N}} - \{e\}$ yield a minimal (u, v)-cut in $\mathcal{N}$.

This implies that $\bar{\mathcal{N}}$ also informs on which of $F_{u,v}, F_{v,u}$ is the minimum of both. The algorithm that associates the tree network $\bar{\mathcal{N}}$ to $\mathcal{N}$ requires the solution of $|V|\log 2|V|$ individual max-flow problems for the given network which can be done within $O(|V|^4)$ steps using the Dinic–Karzanov algorithm.

In § 4 we consider the edge connectivity k of a directed graph $G = (V, E)$ which is defined to be the minimal number of edges that must be eliminated from E in order to disconnect G, i.e. after this elimination there is no directed path from u to v for some node pair (u, v). It is known from Menger's theorem that the minimal number $F_{u,v}$ of edge-disjoint paths from u to v equals the minimal number of edges that must be eliminated from E in order to destroy all paths from u to v. Clearly $F_{u,v}$ is the value of the maximal flow from u to v in the network (V, E, c) with unit edge capacities $c(e) = 1$ for all $e \in E$. Therefore $k = \min\{F_{u,v} | u, v \in V\}$ can be determined by solving $|V|(|V|-1)$ individual max-flow problems and this number can be reduced to $|V|\log 2|V|$ by the above results. Moreover, Lemma 2.1 gives an extremely easy reduction of the problem to the solution of $|V|$ individual max-flow problems. This reduction leads to an algorithm which determined k within $O(k|V||E|)$ steps on a storage manipulation machine, see Schönhage [10], or equivalently on a RAM machine with ± 1 addition/subtraction, see Schnorr [9] for the equivalence proof. Our $O(k|V||E|)$-algorithm competes with an edge connectivity algorithm of Even and Tarjan [2] which runs in $O(\min(|V|^{2/3}, |E|^{1/2})\cdot|V||E|)$ steps. Our time bound beats this time bound provided $k \leqq |V|^{2/3}$ and clearly $k \leqq |V|^{2/3}$ should hold in most of the examples with practical interest.

2. Computing by solving $|V|\log 2|V|$ individual max-flow problems. Throughout the paper let $\mathcal{N} = (V, E, c)$ be a fixed network.

LEMMA 2.1. *Let $u_1, u_2, \cdots, u_r \in V$, then $F_{u_1,u_r} \geqq \min\{F_{u_i,u_{i+1}} | i = 1, \cdots, r-1\}$.*

Proof. Let $(S, \bar{S})$ be a minimal (u_1, u_r)-cut. Since $u_1 \in S$, $u_r \in \bar{S}$ there exists $u_i \in S$ such that $u_{i+1} \in \bar{S}$. Hence $(S, \bar{S})$ is a (u_i, u_{i+1})-cut. Therefore 1.1 implies $F_{u_i,u_{i+1}} \leqq c(S, \bar{S}) = F_{u_1,u_r}$. □

Let us call $u = (u_0, u_1, \cdots, u_r)$ a *U-cycle* if $U = \{u_1, u_2, \cdots, u_r\}$ and $u_0 = u_r$. Let $U \subset V$, then $(S, \bar{S})$ is called a *U-cut* if $S \subset V \wedge \bar{S} = V - S \wedge S \cap U \neq \emptyset \wedge \bar{S} \cap U \neq \emptyset$. A U-cut $(S, \bar{S})$ is called *minimal* if

$$c(S, \bar{S}) = \min\{c(D, \bar{D}) | \text{all } U\text{-cuts } (D, \bar{D})\}.$$

LEMMA 2.2. *Let* $(u_0, u_1, \cdots, u_r)$ *be a* U*-cycle,* $U \subset V$. *Then*
(1) $\forall u, v \in U : F_{u,v} \geqq \min \{F_{u_i,u_{i+1}} | i = 0, 1, \cdots, r-1\}$
(2) $\exists j$: *all minimal* (u_j, u_{j+1})*-cuts are minimal* U*-cuts.*
Proof. (1). Let $u = u_k$, $v = u_j$ and let $(S, \bar{S})$ be a minimal (u_k, u_j)*-cut.*

$$k < j \Rightarrow F_{u_k,u_j} \geqq \min \{F_{u_i,u_{i+1}} | i = k, k+1, \cdots, j-1\}$$

$$j < k \Rightarrow F_{u_k,u_j} \geqq \min \{F_{u_i,u_{i+1}} | i = k, k+1, \cdots, r-1, 0, 1, \cdots, j-1\}.$$

(2). It follows from (1) that for some j

$$F_{u_j,u_{j+1}} = \min \{F_{u,v} | u, v \in U\}.$$

Let $(D, \bar{D})$ be any minimal (u_j, u_{j+1})-cut and let $(S, \bar{S})$ be any minimal U-cut. Then $(S, \bar{S})$ is a (u, v)-cut for some $u, v \in U$. Then 1.1 implies

$$c(D, \bar{D}) = F_{u_j,u_{j+1}} \leqq F_{u,v} \leqq c(S, \bar{S})$$

which proves that $(D, \bar{D})$ is a minimal U-cut. □

We now describe a multiterminal maximal flow algorithm MMF which for a given network $\mathcal{N} = (V, E, c)$ computes the matrix $\mathcal{B} := (\min \{F_{u,v}, F_{v,u}\} | u, v \in V)$. MMF uses a subprogram IMF for solving individual maximal flow problems for $\mathcal{N}$: IMF $(u, v, B, \bar{B})$ computes a minimal (u, v)-cut $(B, \bar{B})$ for $\mathcal{N}$. This can be done within $O(|V|^3)$ RAM-steps by applying the Dinic–Karzanov algorithm. In this section we count the total number of IMF-calls during the execution of MMF since the execution of the IMF-calls dominates all other steps.

The inputs of the recursive procedure MMF $(n, \bar{u}, \bar{A})$ are a natural number $n \geqq 2$, a U-cycle $\bar{u} = (u_0, u_1, \cdots, u_n)$ for some $U \subset V$ and a sequence $\bar{A} = ((A_i, \bar{A}_i) | i = 1, 2, \cdots, n)$ such that $(A_i, \bar{A}_i)$ is a minimal (u_{i-1}, u_i)-cut for $\mathcal{N}$. MMF $(n, \bar{u}, \bar{A})$ computes $\min \{F_{u,v}, F_{v,u}\}$ for all $u, v \in U = \{u_1, \cdots, u_n\}$. At first MMF determines a minimal U-cut $(A_j, \bar{A}_j)$ according to Lemma 2.2. This yields $F_{u,v} = \min \{F_{u,v}, F_{v,u}\} = c(A_j, \bar{A}_j)$ for all $u \in A_j \cap U$ and $v \in \bar{A}_j \cap U$. Then the problem of computing the remaining values of $\min \{F_{u,v}, F_{v,u}\}$ is split into two subproblems which are solved by two recursive calls for MMF with input parameters of smaller size.

The recursive procedure MMF $(n, \bar{u}, \bar{A})$ 2.3.

begin $U := \{u_i | 1 \leqq i \leqq n\}$
Determine j with $c(A_j, \bar{A}_j) = \min \{c(A_i, \bar{A}_i) | i = 1, 2, \cdots, n\}$
comment according to 2.2 $(A_j, \bar{A}_j)$ is a minimal U-cut
$V_1 := A_j \cap U$, $V_2 := \bar{A}_j \cap U$, $n_1 := |V_1|$, $n_2 := |V_2|$
for all $u \in V_1$, $v \in V_2$ **do** $F_{u,v} := \min \{F_{u,v}, F_{v,u}\} := c(A_j, \bar{A}_j)$
for $\nu = 1, 2$ **do**
begin if $n_\nu = 1$ **then for** $u \in V_\nu$ **do** $[F_{u,u} := \infty$ **return**]
compose a V_ν-cycle $\bar{u}^\nu = (u_0^\nu, u_1^\nu, \cdots, u_{n_\nu}^\nu)$
comment Lemma 2.6 below describes how to form $\bar{u}^1$, $\bar{u}^2$ such that the number of IMF-calls in the following block becomes minimum.
for all edges (u_i^ν, u_{i+1}^ν) in $\bar{u}^\nu$ **do**
if a minimal (u_i^ν, u_{i+1}^ν)-cut has been stored
then call this cut $(A_i^\nu, \bar{A}_i^\nu)$
else IMF $(u_i^\nu, u_{i+1}^\nu, A_i^\nu, \bar{A}_i^\nu)$

$\bar{A}^\nu := ((A_i^\nu, \bar{A}_i^\nu) | i = 1, 2, \cdots, n_\nu)$
MMF $(n_\nu, \bar{u}^\nu, \bar{A}^\nu)$
end
end

THEOREM 2.4. *MMF* $(n, \bar{u}, \bar{A})$ *correctly computes* $(\min \{F_{u,v}, F_{v,u}\} | u, v \in U)$.

Proof. We proceed by induction on n. According to Lemma 2.2 $(A_j, \bar{A}_j)$ is a minimal U-cut. It follows $\forall u \in A_j \cap U : \forall v \in \bar{A}_j \cap U$:

$$F_{u,v} \leqq c(A_j, \bar{A}_j) \quad \text{since } (A_j, \bar{A}_j) \text{ is a } (u, v)\text{-cut},$$

$\min \{F_{u,v}, F_{v,u}\} \geqq c(A_j, \bar{A}_j)$ since $(A_j, \bar{A}_j)$ is a minimal U-cut. Hence $F_{u,v} = \min \{F_{u,v}, F_{v,u}\} = c(A_j, \bar{A}_j)$ and $\min \{F_{u,v}, F_{v,u}\}$ is correctly computed for $u \in A_j \cap U$, $v \in \bar{A}_j \cap U$. Since $n_\nu < n$ it follows that the remaining values $\min \{F_{u,v}, F_{v,u}\}$ are correctly computed by the induction hypothesis provided $n_\nu \geqq 2$. Moreover, if $n_\nu = 1$ then MMF correctly determines $F_{u,u} = \infty$ for $\{u\} = V_\nu$. □

An immediate consequence of algorithm 2.3 is the following

COROLLARY 2.5. *For any network* (V, E, c) *and any* $U \subset V$ *the matrix* $(\min \{F_{u,v}, F_{v,u}\} | u, v \in U)$ *has at most* $|U| - 1$ *distinct finite values.*

Proof. We prove by induction on n that MMF $(n, \bar{u}, \bar{A})$ yields at most $n - 1$ finite values. MMF $(n, \bar{u}, \bar{A})$ yields the values $c(A_j, \bar{A}_j)$ and by induction hypothesis $\leqq n_\nu - 1$ finite values which are obtained by MMF $(n_\nu, \bar{u}^\nu, \bar{A}^\nu)$. Since $n_1 + n_2 = n$ this yields at most $n - 1$ distinct finite values in total. Observe that for $n = 1$ MMF $(n, \bar{u}, \bar{A})$ only yields the infinite value ∞. □

The *first stage* of MMF $(n, \bar{u}, \bar{A})$ consists of all those operations which are not part of the recursive calls MMF $(n_\nu, \bar{u}^\nu, \bar{A}^\nu)$ for $\nu = 1, 2$. Let $(A_j, \bar{A}_j)$ be the minimal U-cut which is determined within the first stage of MMF $(n, \bar{u}, \bar{A})$. An A_j-segment ($\bar{A}_j$-segment, resp.) S in the cycle $\bar{u}$ is a maximal segment $S = (u_i, u_{i+1}, \cdots, u_k)$ such that all vertices of S are in A_j ($\bar{A}_j$ resp.). Obviously the number of A_j-segments of $\bar{u}$ equals the number of $\bar{A}_j$-segment.

LEMMA 2.6. *Suppose* $\bar{u}$ *has exactly* m A_j*-segments. Then* $\bar{u}^1, \bar{u}^2$ *can be chosen such that* $2m$ *IMF-calls are to be executed within the first stage of MMF* $(n, \bar{u}, \bar{A})$.

Proof. Consider a cycle $\bar{u}$ with vertices in A_j denoted as □ and vertices in $\bar{A}_j$ denoted as ○ in Figs. 1 and 2.

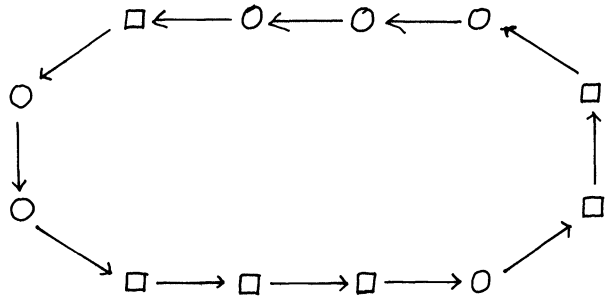


FIG. 1

Then MMF $(n, \bar{u}, \bar{A})$ can construct an $A_j \cap U$-cycle ($\bar{A}_j \cap U$-cycle, resp.) by substituting each A_j-segment S ($\bar{A}_j$-segment, resp.) in $\bar{u}$ by a new edge that connects the neighboring $\bar{A}_j$-segments (A_j-segments, resp.) of S. For instance, the new edges - - → in the $A_j \cap U$-cycle are as shown in Fig. 2.

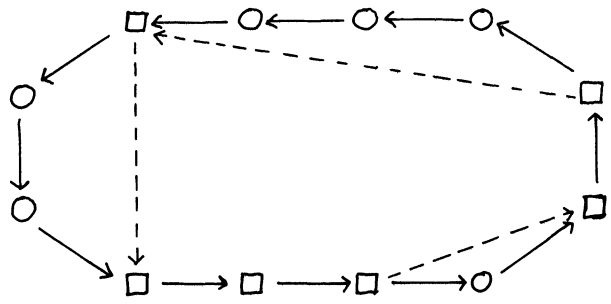


FIG. 2

In the following we suppose that MMF is executed according to Lemma 2.6, i.e. $2m$ IMF-calls are executed within the first stage of MMF $(n, \bar{u}, \bar{A})$ provided $\bar{u}$ consists of m A_j-segments and m $\bar{A}_j$-segments.

Let $\psi(n)$ be the maximal number of IMF-calls that are executed within the entire procedure MMF $(n, \bar{u}, \bar{A})$ for any $\bar{u}$ and any $\bar{A}$. Algorithm 2.3 and Lemma 2.6 imply the following recursion formula and initial values of ψ:

$$\psi(n) \leqq \max(\{\psi(n-k)+\psi(k)+2k \mid 2 \leqq k \leqq n/2\} \cup \{\psi(n-1)+1\}) \quad \text{for } n > 2 \tag{2.7}$$

(2.8)

n	1	2	3	4	5	6	7	8
$\psi(n)$	0	0	1	4	5	8	11	16

LEMMA 2.9 (Hanke Bremer). $\psi(n) \leqq n \log n$.

Proof. The assertion holds for $n = 1, 2$. We proceed by induction on n. Suppose $\psi(j) \leqq j \log j$ for $j < n$. In order to prove $\psi(n) \leqq n \log n$ we have to show:

(1) $\psi(n-1)+1 \leqq n \log n$ (which trivially holds),

(2) $\psi(n-k)+\psi(k)+2k \leqq n \log n$ for $2 \leqq k \leqq n/2$.

By the induction hypothesis:

$$\begin{aligned}
\psi(n-k)+\psi(k)+2k &\leqq (n-k)\log(n-k)+k\log k+2k \\
&= (n-k)\log\left(\frac{n-k}{n}\right)+k\log(k/n)+2k+n\log n \\
&= n\left(\frac{n-k}{n}\log\left(\frac{n-k}{n}\right)+k/n\log(k/n)\right)+2k+n\log n \\
&= n\left(-H\left(\frac{n-k}{n},\frac{k}{n}\right)+2k/n\right)+n\log n
\end{aligned}$$

where $H(x, 1-x) = -x \log x - (1-x)\log(1-x)$ is the Shannon entropy. As is well known $H(x, 1-x)$ is convex. Therefore $H(0, 1) = 0$, $H(\frac{1}{2}, \frac{1}{2}) = 1$ implies

$$H(x, 1-x) \geqq 2x \quad \text{for } x \leqq \tfrac{1}{2}.$$

This yields $-H((n-k)/n, k/n)+2k/n \leqq 0$ which proves $\psi(n-k)+\psi(k)+2k \leqq n \log n$. □

This yields

THEOREM 2.10. *The computation of $\mathfrak{B}$ via program 2.3 requires the solution of at most $|V| \log 2|V|$ individual max flow problems for the given network (V, E, c).*

Proof. The computation of a vector $\bar{A}$ of minimal cuts along a V-cycle $\bar{u}$ requires $|V|$ IMF-calls. By Lemma 2.9 there are at most $|V| \log |V|$ IMF-calls during the execution of MMF $(|V|, \bar{u}, \bar{A})$. □

3. The tree-network that represents all bottlenecks. Our further improvements to algorithm 2.3 are based on the following Lemma.

LEMMA 3.1. *Let $U \subset V$ and let $(A, \bar{A})$ be a minimal U-cut. Then*

(1) $\forall \bar{u}, \bar{v} \in \bar{A}, \bar{u} \neq \bar{v}: \exists$ *minimal* $(\bar{u}, \bar{v})$*-cut* $(S, \bar{S})$: $A \cap U \subset S \vee A \cap U \subset \bar{S}$;

(2) $\forall u, v \in A, u \neq v: \exists$ *minimal* (u, v)*-cut* $(S, \bar{S})$: $\bar{A} \cap U \subset S \vee \bar{A} \cap U \subset \bar{S}$.

Proof. For any cut $(B, \bar{B})$ we set

$$B_A := B \cap A, \quad B_{\bar{A}} := B \cap \bar{A},$$
$$\bar{B}_A := \bar{B} \cap A, \quad \bar{B}_{\bar{A}} := \bar{B} \cap \bar{A}.$$

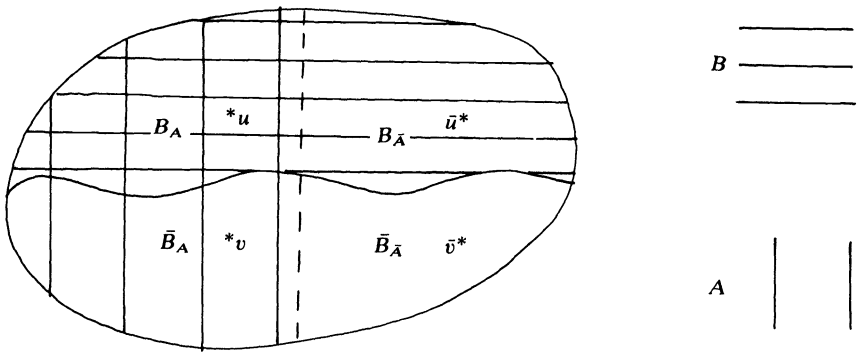


FIG. 3

For $D, H \subset V$ we define $C_{DH} := \sum_{(u,v) \in D \times H} c(u, v)$.

(1) Suppose $(B, \bar{B})$ is a minimal $(\bar{u}, \bar{v})$-cut. If $A \cap U \subset B$ or $A \cap U \subset \bar{B}$ then set $(S, \bar{S}) := (B, \bar{B})$ and we are finished. Otherwise set $(S, \bar{S}) := (B \cup \bar{B}_A, \bar{B} - \bar{B}_A) = (B \cup \bar{B}_A, B_{\bar{A}})$. $(S, \bar{S})$ is a $(\bar{u}, \bar{v})$-cut and $A \cap U \subset S$. Moreover, it can be seen from the above figure that

$$\begin{aligned} c(B, \bar{B}) - c(B \cup \bar{B}_A, \bar{B} - \bar{B}_A) &= C_{B_A \bar{B}_A} + C_{B_{\bar{A}} \bar{B}_A} - C_{\bar{B}_A \bar{B}_{\bar{A}}} \\ &\geqq C_{B_A \bar{B}_A} - C_{\bar{B}_A B_{\bar{A}}} - C_{\bar{B}_A \bar{B}_{\bar{A}}} \\ &= c(A - \bar{B}_A, \bar{A} \cup \bar{B}_A) - c(A, \bar{A}). \end{aligned}$$

$(A - \bar{B}_A, \bar{A} \cup \bar{B}_A)$ is a U-cut since $A \cap U \cap B \neq \emptyset$ and $A \cap U \cap \bar{B} \neq \emptyset$. Since $(A, \bar{A})$ is a minimal U-cut the term in the last line above is $\geqq 0$. This proves that $(S, \bar{S}) := (B \cup \bar{B}_A, \bar{B} - \bar{B}_A)$ is a minimal $(\bar{u}, \bar{v})$-cut.

(2) Suppose that $(B, \bar{B})$ is a minimal (u, v)-cut. If $\bar{A} \cap U \subset B$ or $\bar{A} \cap U \subset \bar{B}$ then define $(S, \bar{S}) := (B, \bar{B})$. Otherwise set $(S, \bar{S}) := (B - B_{\bar{A}}, \bar{B} \cup B_{\bar{A}})$. $(S, \bar{S})$ is a (u, v)-cut and $\bar{A} \cap U \subset \bar{S}$. Moreover, it can be seen from Fig. 3 that

$$\begin{aligned} c(B, \bar{B}) - c(B - B_{\bar{A}}, \bar{B} \cup B_{\bar{A}}) &= C_{B_{\bar{A}} \bar{B}_{\bar{A}}} + C_{B_{\bar{A}} \bar{B}_A} - C_{B_A B_{\bar{A}}} \\ &\geqq C_{B_{\bar{A}} \bar{B}_{\bar{A}}} - C_{\bar{B}_A B_{\bar{A}}} - C_{B_A B_{\bar{A}}} \\ &= c(A \cup B_{\bar{A}}, \bar{A} - B_{\bar{A}}) - c(A, \bar{A}). \end{aligned}$$

$(A \cup B_{\bar{A}}, \bar{A} - B_{\bar{A}})$ is a U-cut since $\bar{A} \cap U \cap \bar{B} \neq \emptyset$ and $\bar{A} \cap U \cap B \neq \emptyset$ in this case. Since $(A, \bar{A})$ is a minimal U-cut the term in the last line above is $\geqq 0$. This proves that $(S, \bar{S}) := (B - B_{\bar{A}}, \bar{B} \cup B_A)$ is a minimal (u, v)-cut. □

It follows from Lemma 3.1 that, given a minimal U-cut $(A, \bar{A})$ in order to determine $F_{\bar{u},\bar{v}}$ with $\bar{u}, \bar{v} \in \bar{A} \cap U$, one can first contract $A \cap U$, and in order to determine $F_{u,v}$ with $u, v \in A \cap U$ one can first contract $\bar{A} \cap U$. Here contracting a subset $B \subset V$ means the following operations:

for all $a \in V - B$ **do**
$[c(a, B) := \sum_{b \in B} c(a, b), \qquad c(B, a) := \sum_{b \in B} c(b, a)]$
$V := V - B \cup \{B\}$.

Such a new node B that is formed by contraction is called a *supernode.*

Inserting these contractions into the algorithm 2.3 means that at the beginning of the interior block the subset V_2 is contracted if $\nu = 1$ and V_1 is contracted if $\nu = 2$. At the end of this block the contraction is reversed by a decompose statement. Here decompose (B) substitutes the supernode B by its previous subset of nodes. With these changes the interior block in algorithm 2.3 looks as follows:

for $\nu = 1, 2$ **do**
begin contract $V_{2-\nu}$
old statements of the block
decompose $V_{2-\nu}$
end

Lemma 3.1 immediately yields the following

THEOREM 3.2. *The modified procedure MMF* $(n, \bar{u}, \bar{A})$ *correctly computes* $(\min \{F_{u,v}, F_{v,u}\} | u, v \in U)$.

Let $(B_i, \bar{B}_i)$ $i = 1, \cdots, |V| - 1$ be system of minimal U-cuts that is determined within the different stages of the modified algorithm MMF $(|V|, \bar{u}, \bar{A})$ with a V-cycle $\bar{u}$ and an appropriate $\bar{A}$ as inputs. Let $(B_i, \bar{B}_i)$ be computed before $(B_{i+1}, \bar{B}_{i+1})$. For instance $(B_1, \bar{B}_1)$ is the minimal V-cut $(A_j, \bar{A}_j)$ which is determined within the first stage of MMF $(|V|, \bar{u}, \bar{A})$. Then $(B_2, \bar{B}_2)$ either is a minimal A_j-cut or is a minimal $\bar{A}_j$-cut which is determined within one of the recursive calls MMF $(n_\nu, \bar{u}^\nu, \bar{A}_\nu)$ $\nu = 1, 2$. In general $(B_i, \bar{B}_i)$ is a minimal U-cut with $U = B_j$ or $U = \bar{B}_j$ for some $j < i$.

We are now able to represent the cuts $(B_i, \bar{B}_i)$ $i = 1, \cdots, k$ by a tree network $\mathcal{N}_k$. This representation extends the construction of Gomory and Hu [4] to unsymmetrical networks.

DEFINITION 3.3. The network $\mathcal{N}_k$ *represents* the cuts $(B_i, \bar{B}_i)$ $i = 1, \cdots, k$ if

(1) the nodes of $\mathcal{N}_k$ constitute a partition of V into $k + 1$ blocks and

(2) for each cut $(B_i, \bar{B}_i)$ there is an edge e in $\mathcal{N}_k$ that *represents* $(B_i, \bar{B}_i)$, i.e. $\mathcal{N}_k$-$\{e\}$ splits into two weak components that partition V into B_i and $\bar{B}_i$, e is directed from B_i to $\bar{B}_i$ and has capacity $c(B_i, \bar{B}_i)$; (the classes of nodes that are connected by undirected paths are called *weak components*).

We describe the construction of the tree network $\mathcal{N}_k$ that represents the cuts $(B_i, \bar{B}_i)$ $i = 1, \cdots, k$. The minimal V-cut $(B_1, \bar{B}_1)$ is represented by the network $B_1 \xrightarrow{c(B_1, \bar{B}_1)} \bar{B}_1$.

Suppose that $(B_2, \bar{B}_2)$ is a minimal B_1-cut which has been determined after contracting $\bar{B}_1$. We distinguish two cases:

(a) $\bar{B}_1 \in B_2$: then

$$\bar{B}_2 \xleftarrow{c(B_2, \bar{B}_2)} B_2 \cap B_1 \xrightarrow{c(B_1, \bar{B}_1)} \bar{B}_1$$

represents the cuts $(B_1, \bar{B}_1)$, $(B_2, \bar{B}_2)$;

(b) $\bar{B}_1 \in \bar{B}_2$: then

$$B_2 \xrightarrow{c(B_2,\bar{B}_2)} \bar{B}_2 \cap B_1 \xrightarrow{c(B_1,\bar{B}_1)} \bar{B}_1$$

represents the cuts $(B_1, \bar{B}_1)$, $(B_2, \bar{B}_2)$.

The induction step that constructs $\mathcal{N}_{k+1}$ from $\mathcal{N}_k$ operates as follows: Observe that $(B_{k+1}, \bar{B}_{k+1})$ is a minimal U-cut for some supernode U of $\mathcal{N}_k$ and has been constructed after contracting each B_i with $i \leqq k$ and $U \subset \bar{B}_i$ and after contracting each $\bar{B}_i$ with $i \leqq k$ and $U \subset B_i$. This series of contractions can equivalently be obtained by eliminating U from $\mathcal{N}_k$ and by contracting each weak component of $\mathcal{N}_k - \{U\}$. $\mathcal{N}_{k+1}$ is obtained from $\mathcal{N}_k$ by splitting U into

$$U \cap B_{k+1} \xrightarrow{c(B_{k+1},\bar{B}_{k+1}} U \cap \bar{B}_{k+1}.$$

The weak components of $\mathcal{N}_k - \{U\}$ are attached either to $U \cap B_{k+1}$ provided that the component is contained in B_{k+1}, or to $U \cap \bar{B}_{k+1}$ provided that the component is contained in $\bar{B}_{k+1}$. This ensures that the new edge represents the cut $(B_{k+1}, \bar{B}_{k+1})$. Obviously the construction implies that the old edges in $\mathcal{N}_{k+1}$ still represent their corresponding cuts.

This proves that $\mathcal{N}_k$ represents the cuts $(B_1, \bar{B}_1)$ through $(B_k, \bar{B}_k)$. Moreover, it can easily be seen that $\mathcal{N}_k$ is the unique tree-like network that represents $(B_1, \bar{B}_1)$ through $(B_k, \bar{B}_k)$.

In particular $\bar{\mathcal{N}} := \mathcal{N}_{|V|-1}$ has node set $\{\{v\} \mid v \in V\}$ and stores $\mathcal{B}$ as follows:

THEOREM 3.4. *$\min\{F_{u,v}, F_{v,u}\}$ equals the minimal capacity of the edges on the path that connects $\{u\}$ and $\{v\}$ in $\bar{\mathcal{N}}$. If among the minimal capacity edges on the undirected path connecting u and v in $\bar{\mathcal{N}}$ some edge e is directed from $\{u\}$ to $\{v\}$ then $F_{u,v} = \min\{F_{u,v}, F_{v,u}\}$ and the weak components of $\bar{\mathcal{N}} - \{e\}$ yield a minimal (u, v)-cut.*

Proof. Let $|V| = n$. $\mathcal{N}_{n-1}$ represents all cuts $(B_1, \bar{B}_1)$ through $(B_{n-1}, \bar{B}_{n-1})$ that appear during the computation of $\mathcal{B}$ by the modified procedure MMF $(n, \bar{u}, \bar{A})$. Each edge e on the path that connects $\{u\}$, $\{v\}$ in $\mathcal{N}_{n-1}$ represents some (u, v)-cut if e is directed from u to v and represents some (v, u)-cut otherwise. Hence $c(e) \geqq \min\{F_{u,v}, F_{v,u}\}$ for all edges e on the undirected path that connects $\{u\}$ and $\{v\}$ in $\mathcal{N}_{n-1}$. On the other hand some cut $(B_i, \bar{B}_i)$ occurs during the modified procedure MMF $(|V|, \bar{u}, \bar{A})$ with $c(B_i, \bar{B}_i) = \min\{F_{u,v}, F_{v,u}\}$ and $(B_i, \bar{B}_i)$ is either a (u, v)-cut or a (v, u)-cut. We know that this cut is represented by some edge e in $\mathcal{N}_{n-1}$. Moreover, this edge e must lie on the undirected path that connects $\{u\}$ and $\{v\}$ in $\mathcal{N}_{n-1}$ since by eliminating e from $\mathcal{N}_{n-1}$, $\mathcal{N}_{n-1}$ splits such that $\{u\}$ and $\{v\}$ fall in different weak components. This altogether proves that $\min\{F_{u,v}, F_{v,u}\}$ equals the minimal capacity of the edges on the path that connects $\{u\}$ and $\{v\}$ in $\mathcal{N}_{n-1}$.

Now suppose that some minimal capacity edge e on the undirected path that connects $\{u\}$ and $\{v\}$ in $\mathcal{N}_{n-1}$ is directed from $\{u\}$ to $\{v\}$. Then e represents a (u, v)-cut $(B_i, \bar{B}_i)$ with capacity $c(e)$. Hence $F_{u,v} \leqq c(e) = \min\{F_{u,v}, F_{v,u}\}$ which implies $F_{u,v} = \min\{F_{u,v}, F_{v,u}\}$. □

Next we bound the total number of steps that are sufficient to compute $\mathcal{B}$ for networks with n nodes on a Random Access Machine with full addition of real numbers. We suppose that the Dinic–Karzanov algorithm is used for a single IMF-call which runs in $O(m^3)$ steps for networks with m nodes. Let $|V| = n$ then $O(n^4)$ steps are sufficient in order to compute a sequence $\bar{A}$ of minimal cuts along a V-cycle $\bar{u}$. It remains to bound

the execution time of the modified procedure MMF $(n, \bar{u}, \bar{A})$. Let $\delta(n, m)$ be the maximal number of RAM-steps that are used by the modified version of MMF$(n, \bar{u}, \bar{A})$ for any $\bar{u}$ and $\bar{A}$ with respect to networks with m vertices.

Remember that MMF $(n, \bar{u}, \bar{A})$ determines a minimal V-cut $(A_j, \bar{A}_j)$ with $V = \{u_0, \cdots, u_n\}$. Let $\bar{u}$ have $k \leqq n/2$ A_j-segments and k $\bar{A}_j$-segments. Then k IMF-calls are executed on a network where A_j has been contracted and k IMF-calls are executed on a network where $\bar{A}_j$ has been contracted. Finally MMF $(n_\nu, \bar{u}^\nu, \bar{A}^\nu)$, $\nu = 1, 2$, is executed on networks of size $m - p + 1$ and $p + 1$ with $k \leqq p \leqq m/2$. We have $n_1 + n_2 = n$ with $n_1, n_2 \geqq k$. Therefore we obtain an upper bound $\bar{\delta}$ for δ by the following recursion scheme with some suitable $a, b \in N$:

$$\bar{\delta}(n, m) = am^2 + \max\{\bar{\delta}(n - n_1, m - p + 1) + \bar{\delta}(n_1, p + 1) + bk[(m - p + 1)^3 + (p + 1)^3] \mid$$
$$1 \leqq k \leqq n/2,\ k \leqq n_1 \leqq p \leqq m/2\}.$$

Here am^2 bounds all side computations and b is the linear factor in the $O(m^3)$ bound for the Dinic–Karzanov algorithm. For a sufficiently large $c \in N$ the bound $\bar{\delta}(n, m) \leqq cnm^3$ can be proved inductively on n using the above recursion. In particular we have $\delta(n, n) = O(n^4)$. This proves

THEOREM 3.5. *The modified algorithm* 2.3 *constructs* $\bar{\mathcal{N}}$ *from* $\mathcal{N}$ *within* $O(\min(T|V| \log |V|, |V|^4))$ *steps where* T *is the best time bound for solving individual max flow problems for* $\mathcal{N}$. *Thus* $\mathcal{B}$ *can be computed within the same time.*

4. Determining the edge connectivity k within $O(k|V||E|)$ steps. Let $G = (V, E)$ be the given directed graph. Then k is the minimal number of edges that must be eliminated from E in order to disconnect G. Let $\mathcal{N} = (V, E, c)$ be the network with unit edge capacities $c(e) = 1$ for all $e \in E$. For this network $F_{u,v}$ is the maximal number of edge disjoint paths from u to v. It follows from the max-flow-min-cut Theorem 1.1 that

$$k = \min\{F_{u,v} | u, v \in V\}.$$

Let $\bar{u} = (u_0, u_1, \cdots, u_n)$ with $n = |V|$ be any V-cycle; then Lemma 2.2 implies

$$k = \min\{F_{u_i, u_{i+1}} | i = 0, \cdots, n - 1\}.$$

Using this equality we determine k by computing flows f_i from u_i to u_{i+1} such that within stage j the value $\Phi(f_i)$ of f_i is increased from $j - 1$ to j for all i. This can be done by constructing an augmenting path with respect to f_i. We assume that the reader is familiar with the concept of augmenting paths.

THE CONNECTIVITY ALGORITHM 4.1.

$f_i \equiv 0$ for $i = 0, 1, \cdots, n - 1$
stage $:= 0$
Marke: **for** $i = 0, 1, \cdots, n - 1$ **do**
if there is an augmenting path with respect to f_i
then increase f_i by 1 along this path
else goto End
stage $:=$ stage $+ 1$ **goto** Marke
End: $k :=$ stage

At stage j the algorithm tests whether $\Phi(f_i)$ can be increased to $j + 1$ for all i. By the previous remarks k is correctly computed and it remains to bound the running time of a suitable implementation of the algorithm on a reference machine. Let $V =$

$\{1, 2, \cdots, n\}$ and suppose that the adjacency lists $E_i = \{j | (i, j) \in E\}$ $i = 1, \cdots, n$ are given as inputs. Then the construction of an augmenting path with respect to f_i can be done within $O(|E|)$ steps by standard methods. Hence each stage of the algorithm can be done within time $O(|V|\,|E|)$ and therefore the connectivity k is computed within $O(k|V|\,|E|)$ steps.

Finally we compare the time bound $O(k|V||E|)$ with the time bound of the edge connectivity algorithm of Even and Tarjan [2, p. 514] which runs in $O(\min(|V|^{2/3}, |E|^{1/2})|V||E|)$ steps. Obviously $k \leqq |E|/|V|$ since each vertex must have outdegree $\geqq k$ in order to ensure edge connectivity k. Therefore $k|V|\,|E| \leqq |E|^2$. On the other hand $|V|^2 \geqq |E|$ implies $|E|^{1/2}|V||E| \geqq |E|^2$. Hence $k \cdot |V||E| \leqq \min(|V|^{2/3}, |E|^{1/2})|V||E|$ provided $k \leqq |V|^{2/3}$. Thus our time bound beats the time bound of Even and Tarjan provided $k \leqq |V|^{2/3}$ and this condition should hold for most practical interesting examples.

Probably our algorithm can still be improved by increasing f_i via Dinic's algorithm [1] instead of using the technique of augmenting paths. Increasing the flow f_i by one phase of Dinic's algorithm requires $O(|E|)$ steps, but this might yield a considerable increase of $\Phi(f_i)$. Unfortunately we are unable to express this improvement by a better time bound in terms of k, $|V|$ and $|E|$.

Acknowledgment. I thank H. Bremer for reading the manuscript and S. Even, Z. Galil and the referee for useful comments.

REFERENCES

[1] E. A. DINIC, *Algorithm for solution of a problem of maximum flow in a network with power estimation*, Soviet Math. Dokl., 11 (1970), pp. 1277–1280.

[2] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, this Journal 4 (1975), pp. 507–518.

[3] L. R. FORD AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, N.J., 1962.

[4] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, J. Soc. Indust. Appl. Math., 19 (1961), pp. 551–570.

[5] A. V. KARZANOV, *Determining the maximal flow in a network by the method of preflows*, Soviet Math. Dokl., 15 (1974), pp. 434–437.

[6] D. J. KLEITMAN, *Methods for investigating connectivity of large graphs*, IEEE Trans. Circuit Theory, CT-16 (1969), pp. 232–233.

[7] K. MENGER, *Zur allgemeinen Kurventheorie*, Fund. Math., 10 (1927), pp. 96–115.

[8] C. P. SCHNORR, *Multiterminal network flow and edge connectivity in unsymmetrical networks*, Proceeding of the 5th Colloquium on automata. Languages and Programming (Udine, 1978), Lecture Notes in Computer Science 62, 1978, pp. 425–439.

[9] ———, *Rekursive Funktionen und ihre Komplexität*, Teubner, Stuttgart, 1974.

[10] A. SCHÖNHAGE, *Universelle Turingspeicherung*, Automatentheorie und formale Sprachen, Dörr, Hotz, eds., Bibliographisches Institut, Mannheim, 1970.

SIAM J. COMPUT.
Vol. 8, No. 2, May 1979

0097-5397/79/0802-0012 $01.00/0

NEARLY ON LINE SCHEDULING OF A UNIFORM PROCESSOR SYSTEM WITH RELEASE TIMES*

SARTAJ SAHNI† AND YOOKUN CHO†

Abstract. An $O(m^2n + mn \log n)$ nearly on line algorithm to preemptively schedule n independent tasks on m uniform processors is presented. It is assumed that there is a release time associated with each task. No task may be started before its release time. All tasks must be completed by a common due time (if possible). Our algorithm generates schedules having $O(nm)$ preemptions in the worst case. The algorithm can also be used to minimize maximum lateness even for the case when all jobs have the same release time but different due times.

Key words. independent tasks, uniform processors, preemptive schedule, release time, common due time, complexity

1. Introduction. A *uniform processor system* $P = \{P_1, P_2, \cdots, P_m\}$ is a set of m processors (machines). Associated with each processor, P_i, is a speed s_i, $s_i > 0$, $1 \leqq i \leqq m$. Processor P_i can perform s_i units of processing in one unit of time. When $s_i = s_{i+1}$, $1 \leqq i < m$, P is said to be a system of *identical processors*. Let T be a set of n independent tasks. Let t_i, r_i and d_i respectively be the processing requirement, release time and due time of task i, $1 \leqq i \leqq n$.

A *DD-schedule* for T is an assignment of tasks to processors such that (i) no processor is required to process more than one task at any time, (ii) no task is simultaneously processed on more than one processor, (iii) the processing of no task begins before its release time and (iv) all tasks are completed by their due times. Note that not all task sets have *DD*-schedules on a given processor system.

A nearly on line algorithm to find a *DD*-schedule (if one exists) is an algorithm which, for every distinct release time r_i, determines the schedule from 0 to r_i without knowledge of the jobs released on or after r_i.

Many researchers have studied the problem of obtaining *DD*-schedules (when they exist). Rinnooy Kan [6] shows that the problem of determining the existence of nonpreemptive *DD*-schedules is *NP*-Complete. McNaughton's algorithm [8] can be used to obtain preemptive *DD*-schedules for systems of identical processors when the task set T has only one distinct release time and one distinct due time. Gonzalez and Sahni [3] present an $O(n + m \log m)$ algorithm that works for uniform processor systems when T has only one distinct release time and one distinct due time. For the case when all tasks have the same release time (but may have different due times), Horn [4] presents an $O(n^3)$ algorithm to obtain preemptive *DD*-schedules for identical processors. A faster algorithm ($O(n \log mn)$) for this case may be found in [9]. Under the same assumptions on T, Sahni and Cho [10] obtain an $O(n \log n + mn)$ algorithm for uniform processors. Since, in all the cases cited so far all tasks are released at the same time, all the algorithms obtained are, of necessity, on line.

For the case when no restriction is placed on the task set T, Horn [4] presents an $O(n^3)$ algorithm for preemptive schedules on identical processors. Bruno and Gonzalez [1] present a similar algorithm for a system of two uniform processors. Neither of these two algorithms is on line. In fact, it is known [9] that no nearly on line algorithm exists when tasks are allowed to have arbitrary release and due times.

* Received by the editors October 10, 1977. This work was supported in part by the National Science Foundation under Grant MCS 76-21024.

† Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455.

Another special case that has been studied is when all tasks have the same due time. While, this case is symmetric to the case when all tasks have the same release time, the algorithms for the latter case do not result in on line algorithms for the former. Gonzalez and Johnson [2] have obtained an $O(nm)$ nearly on line algorithm for identical processors when all tasks have the same due time. Their algorithm generates DD-schedules (when they exist) having at most $O(nm)$ preemptions. In this paper we extend their result to the case of uniform processors. Our algorithm has time complexity $O(m^2n + mn \log n)$ and generates schedules with at most $O(nm)$ preemptions. In [10] we demonstrated the existence of task sets for which every DD-schedule (even those generated by off line algorithms) had at least $O(nm)$ preemptions. This algorithm can also be used to obtain schedules minimizing lateness when all jobs have the same release time but differing due times. To do this we just change the roles of due times and release times.

2. The algorithm. We first present a nearly on line algorithm that generates schedules with $O(mn + n^2)$ preemptions. Later, we shall show how to modify this algorithm so that the number of preemptions is $O(nm)$.

Assume that the n tasks to be scheduled have v distinct release times r_i, $1 \leqq i \leqq v$. Assume $r_i < r_{i+1}$, $1 \leqq i < v$ and $r_1 = 0$. Our algorithm works in v phases. In the ith phase tasks are scheduled from time r_i to time r_{i+1}, $1 \leqq i < v$. In the vth (and last) phase scheduling is done for the interval $[r_v, d]$ where d is the common due time of all tasks. The tasks available for scheduling in the ith phase are those released at or before time r_i and which haven't yet been completed, i.e., all released tasks with a nonzero remaining processing time (RPT). For each phase, i, algorithm EQUAL determines the amount each available task is to be processed. It does this by using an equalizing rule that attempts to equalize the RPTs of all tasks at the end of the phase. EQUAL utilizes the following fact which is due to Liu and Liu [7] and Horvath, Lam and Sethi [5]:

Fact 1. Let $a_1 \geqq a_2 \geqq \cdots \geqq a_l$ be a set of task times. Let w be the minimum finish time of any preemptive schedule for these l tasks on a system $P = \{P_1, P_2, \cdots, P_m\}$. Let $s_1 \geqq s_2 \geqq \cdots \geqq s_m$. Then,

$$w = \max\left\{\sum_1^l a_i \Big/ \sum_1^m s_i, \max_{1 \leqq j < m}\left\{\sum_1^j a_i \Big/ \sum_1^j s_i\right\}\right\}. \tag{1}$$

Using (1), EQUAL ensures that the amount of processing it is assigning for each task in phase i is such that all the phase i processing can be completed in $\Delta = r_{i+1} - r_i$ time (we may assume $r_{v+1} = d$). The basic strategy in EQUAL is to preferentially process the longest tasks so that the tasks remaining at the next release time are as small as possible. This is comparable to the level strategy used in [5]. The actual schedule for each phase can be constructed using the algorithm of Gonzalez and Sahni [3].

EQUAL has six parameters. Δ is the length of the interval for which scheduling is to be carried out in this phase (it is the time between two successive release times). S is an array such that $S(i) = \sum_{j=1}^{i} s_j$, $1 \leqq i \leqq m$ and $S(0) = 0$. It is assumed that $s_i \geqq s_{i+1}$, $1 \leqq i < m$. At the start of EQUAL, $t(i)$ is the RPT for task i, $1 \leqq i \leqq p$. p is the number of available jobs with nonzero RPT. For convenience, a fictitious job $p+1$ with $t(p+1) = 0$ is assumed. t' is an output array. At termination of EQUAL, $t'(i)$ is the amount job i is to be processed in the interval Δ. Also, $t(i)$ is modified to reflect the RPTs at the end of the interval. EQUAL determines $t'(i)$ such that $t'(i) \geqq t'(i+1)$, $1 \leqq i < p$ and $\max\{\sum_{i=1}^{p} t'(i)/S(m), \max_{1 \leqq j < m}\{\sum_{i=1}^{j} t'(i)/S(j)\}\} \leqq \Delta$. Hence, the assignments determined by EQUAL for the Δ interval can be scheduled. Furthermore, at termination we shall have $t(i) \geqq t(i+1)$, $1 \leqq i < p$.

EQUAL begins by initializing t' and $t(0)$ to zero. When $p < m$, only the fastest p processors need be used. Hence, in line 3, *mused* is initialized to be the actual number of processors to be used. *mlo* is the least index such that P_{mlo} is still available for processing. Processors 1 through $mlo-1$ become unavailable when $\sum_{i=1}^{mlo-1} t'(i)/S(mlo-1) = \Delta$. *nhi* is the index of the next job to be considered. Initially, $mlo = nhi = 1$. $mhi = \min\{nhi-1, mused\}$. AMT is the amount of processing that has so far been assigned to processors P_{mlo} to P_{mhi}. At the start of the loop of lines 6–34, it will be the case that the RPTs of the jobs indexed $mlo, \cdots, nhi$ is the same. This RPT is *tnow*. Initially, $tnow = t(1)$ (line 5). EQUAL sequences through tasks with higher index than *nhi* determining the next smaller task i (line 8). Note that since $t(p+1) = 0$ and $t(p) > 0$, the loop of lines 7–9 will always terminate from line 8. *nhi* is updated in line 10 and *mhi* (line 11) is set so that processors *mlo* to *mhi* may be used for the processing of jobs *mlo* to $nhi-1$. We now attempt to equalize the RPTs of jobs *mlo* to $nhi-1$ with $t(nhi)$. Recall that the present RPTs of all these jobs is *tnow*. Hence, equalization calls for an additional total processing of $(tnow - t(nhi))*(nhi - mlo)$ units. However, only $(S(mhi) - S(mlo-1))*\Delta - \text{AMT}$ units are still unassigned on P_{mlo} to P_{mhi}. INCR is the maximum amount by which each $t'(i)$, $mlo \leqq i < nhi$ is to be increased. This will result either in equalization with $t(nhi)$ or complete utilization of the processors.

ALGORITHM 2.1. The Equalizing Rule.

```
line   procedure EQUAL(Δ, S, t, m, p, t')
       //S(j) = Σ_1^j s_i and S(0) = 0 is assumed. Also t_i and s_i are in nonincreasing//
       // order and t(p+1) = 0//
 1     real t(0:p+1), t'(0:p), S(0:m)
 2     t' ← 0; t(0) ← 0 //t'(0) and t(0) needed in ADJUST; t' ← 0 initializes//
                        //all of t'//
 3     mused ← min{m, p}
 4     mlo ← nhi ← 1; AMT ← 0
 5     tnow ← t(1)
 6     while mlo ≦ mused and tnow ≠ 0 do
 7           for i ← nhi + 1 to p + 1 do //t(p+1) = 0//
 8                 if t(i) < tnow then exit; endif
 9           repeat
10           nhi ← i
11           mhi ← min(nhi - 1, mused)
12           NEXTAMT ← min{(tnow - t(nhi)) * (nhi - mlo)
                                          + AMT, (S(mhi) - S(mlo - 1)) * Δ}
13           INCR ← (NEXTAMT - AMT)/(nhi - mlo)
14           TSUM ← 0
15           for i ← mlo to nhi - 1 do //test if INCR feasible//
16                 t'(i) ← t'(i) + INCR
17                 TSUM ← TSUM + t'(i); i' ← min {mhi, i}
18                 if TSUM > (S(i') - S(mlo - 1)) * Δ then//use up P_mlo, ..., P_i//
19                       DECR ← (TSUM - Δ * (S(i) - S(mlo - 1)))/(i - mlo + 1) //i = i'//
20                       for j ← mlo to i do
21                             t'(j) ← t'(j) - DECR
22                       repeat
23                       call ADJUST
```

```
24                  AMT ← AMT − Δ * (S(i) − S(mlo − 1)) + (INCR − DECR)
                                                        * (i − mlo + 1)
25                  mlo ← i + 1; TSUM ← 0
26                  NEXTAMT ← min {(tnow − t(nhi)) * (nhi − mlo)
                                    + AMT, (S(mhi) − S(mlo − 1)) * Δ}
27                  INCR ← (NEXTAMT − AMT)/(nhi − mlo)
28              endif
29          repeat
30          tnow ← t(nhi)
31          if t(nhi − 1) − t'(nhi − 1) ≠ tnow then AMT ← 0; mlo ← mhi + 1
32                                          else AMT ← NEXTAMT
33          endif
34      repeat
35      for i ← 1 to nhi − 1 do //update RPTs//
36          t(i) ← t(i) − t'(i)
37      repeat
38  end EQUAL
```

ALGORITHM 2.2. Subalgorithm for EQUAL.

```
line  procedure ADJUST
          //All variables used in EQUAL are available inside ADJUST//
 1        if t(i) − t'(i) ≦ t(mlo − 1) − t'(mlo − 1) then return endif
 2        low ← mlo; RPT ← t(i) − t'(i)
 3        jobs ← i − low + 1 //Number of jobs with equal RPT//
 4        while low > 1 and RPT > t(low − 1) − t'(low − 1) do
 5              PRPT ← t(low − 1) − t'(low − 1)
 6              pjobs ← 1
 7              while t(low − pjobs − 1) − t'(low − pjobs − 1) = PRPT do
 8                    pjobs ← pjobs + 1
 9              repeat
10              RPT ← (RPT * jobs + PRPT * pjobs)/(jobs + pjobs)
11              jobs ← jobs + pjobs
12              low ← low − pjobs
13        repeat
14        for q ← low to i do
15            t'(q) ← t(q) − RPT
16        repeat
17    end ADJUST
```

In the loop of lines 15–29 we check to see that increasing the $t'(i)$'s, $mlo \leqq i < nhi$ by INCR still leaves us with $t'(i)$'s that can be scheduled in Δ. This condition is easily tested for by the use of equation (1) and our assertion that $t'(i) \geqq t'(i+1)$ for all i, $mlo \leqq i < mhi$. If the conditional of line 18 is true then, the $t'(i)$'s cannot be increased by INCR. We compute DECR such that all $t'(j)$, $mlo \leqq j \leqq i$ can be increased by INCR − DECR and this is the maximum possible increase. This completely utilizes processors P_{mlo} to P_i. In line 23 the procedure ADJUST is invoked. This procedure ensures that the RPTs of the jobs already assigned to processors of index smaller than *mlo* are not less than those of the newly completed processors *mlo* to *i*. In case this is not true, ADJUST reduces the assigned processing of lower indexed jobs (thus increasing their RPTs) and

increases the $t'(j)$'s of the higher indexed jobs. Now that processors 1 through i have been completely assigned, AMT is updated to reflect the processing assigned only to processors P_{i+1} through P_{mhi}. *mlo* is updated to $i+1$ (the next available processor is P_{i+1}). INCR is recomputed in line 27 and an attempt is made to increase the $t'(j)$ of the jobs $mlo \leqq j < nhi$.

When the loop of lines 15–29 is exited, we will be in one of two conditions. Either, the RPTs of jobs *mlo* to $nhi-1$ have been equalized to $t(nhi)$ or they haven't. In the latter case, from the working of lines 12–29, it must be that all the processors with index less than $mhi+1$ have been fully assigned. Lines 31–33 do the necessary bookkeeping.

The subalgorithm ADJUST used by EQUAL is fairly straightforward. It begins with the knowledge that $t(j)-t'(j) \geqq t(j+1)-t'(j+1)$, $1 \leqq j < mlo$ and $t(j)-t'(j) = t(j+1)-t'(j+1)$, $mlo \leqq j < i$. If the condition of line 1 is true then, no adjustments need be made. Otherwise, in lines 4–13, the algorithm goes through blocks of jobs with an equal RPT. Each such block is identified by the loop of lines 7–9. The processing assignments in this block will be changed so that the RPT of this block together with all jobs seen up to index i will be the same. The new RPT is computed in line 10. One may easily verify that when the $t'(q)$'s are set as in line 15 they can still be processed in Δ units on P_{low} through P_i

From the description of ADJUST, it should be clear that when EQUAL terminates, $t(i) \geqq t(i+1)$, $1 \leqq i < p$.

Now we give an example to show how EQUAL works.

Example 2.1. Assume we have 5 jobs with RPTs 20, 19, 18, 17 and 16 respectively and 5 machines with speeds 20.1, 19.1, 17.7, 16.8 and 16.3 respectively. Further, assume $\Delta = 1$. Then $S(i) = (0, 20.1, 39.2, 56.9, 73.7, 90.0)$.

The job with RPT 20 has highest priority and will be assigned for processing. *tnow* = 20 at line 5. The **for** loop of lines 7–9 will be exited with $i = 2$. NEXTAMT = 1, INCR = 1 in lines 12 and 13. The condition of line 18 doesn't hold. *tnow* is set to 19 and we start a new iteration of the **while** loop of lines 6–34. We have two jobs with RPT 19 now and these two jobs will be processed until their RPTs become 18. We shall follow the same procedure until we reduce the RPTs of all 5 jobs to 16. Then, we will have *mlo* = 1, *tnow* = 16 and AMT = 10. We start the 5th iteration of the **while** loop (lines 6–34). *nhi* becomes 6 at line 10 and $t(nhi) = 0$. NEXTAMT = 90 (line 12) and INCR = 16 (line 13). The **for** loop of lines 15–29 is entered and $t'(1) = 20$, $t'(2) = 19$ and $t'(3) = 18$. Now the condition of line 18 holds and DECR = 0.1/3. The amount of processing is reduced to $t'(1) = 20 - 0.1/3$, $t'(2) = 19 - 0.1/3$ and $t'(3) = 18 - 0.1/3$ (lines 20–22). We execute algorithm ADJUST (line 23) for the first time. The condition of line 1 holds and we return to EQUAL immediately. After executing lines 24–27, we have AMT = 1, *mlo* = 4, NEXTAMT = 33 and INCR = 16. Then the **for** loop (lines 15–29) is reiterated with $i = 4$. $t'(4)$ becomes 17 at line 16. Again the condition of line 18 holds. DECR becomes 0.2 and $t'(4)$ reduces to 16.8. ADJUST is called again and the **while** loop (lines 4–13 in ADJUST) is executed. RPT (line 10 in ADJUST) becomes 0.075 and the amount of processing for each job is adjusted to $t'(1) = 19.925$, $t'(2) = 18.925$, $t'(3) = 17.925$ and $t'(4) = 16.925$. Now we shall have AMT = 0 (line 24), *mlo* = 5, NEXTAMT = 16 and INCR = 16 (line 27). We get $t'(5) = 16$ at line 16. The condition of line 18 doesn't hold and we complete the **while** loop (lines 6–34). The final values are $t(i) = (0.075, 0.075, 0.075, 0.075, 0)$ and $t'(i) = (19.925, 18.925, 17.925, 16.925, 16)$. □

Next we prove some facts about EQUAL. These facts are needed to establish the validity of the final algorithm.

LEMMA 2.1. *Assume we have n jobs with processing time* t_i, $1 \leqq i \leqq n$, *and m machines of speed* s_i, $1 \leqq i \leqq m$. *Assume* $t_i \geqq t_{i+1}$, $1 \leqq i < n$, *and* $s_i \geqq s_{i+1}$, $1 \leqq i < m$. *Let* a_i

be the remaining processing time (RPT) of job i, $1 \leq i \leq n$, after using EQUAL *during a certain interval Δ. Note that $a_i \geq a_{i+1}$, $1 \leq i < n$. Let b_i, $1 \leq i \leq n$ be the RPTs after using any other valid assignment rule during the interval Δ. Let $\sigma(\cdot)$ be such that $b_{\sigma(j)} \geq b_{\sigma(j+1)}$, $1 \leq j < n$. Then $\sum_1^j a_i \leq \sum_1^j b_{\sigma(i)}$, $1 \leq j \leq n$.*

Proof. We first show $\sum_1^n a_i \leq \sum_1^n b_{\sigma(i)}$. This is trivially true if $\sum_{i=1}^n (t_i - a_i) = \Delta * \sum_{i=1}^m s_i$. So, assume $\sum_{i=1}^n (t_i - a_i) < \Delta * \sum_{i=1}^m s_i$. Let *mlo* be as defined at termination of EQUAL. From lines 12, 13, 19, 26 and 27, it follows that $\sum_{i=1}^{mlo-1} (t_i - a_i) = \Delta * \sum_{i=1}^{mlo-1} s_i$. Also, since $\sum_{i=1}^n (t_i - a_i) < \Delta * \sum_{i=1}^m s_i$, it follows that the RPT of jobs *mlo* to *p* is zero. I.e., $a_i = 0$, $mlo \leq i \leq n$. Since, by Fact 1 no more than $\Delta * \sum_{i=1}^{mlo-1} s_i$ of any set of $mlo - 1$ jobs can be processed in Δ, it follows that for every valid assignment $\sum_{i=1}^{mlo-1} (t_i - b_i) \leq \Delta * \sum_{i=1}^{mlo-1} s_i = \sum_{i=1}^{mlo-1} (t_i - a_i)$. Hence, $\sum_{i=1}^{mlo-1} b_i \geq \sum_{i=1}^{mlo-1} a_i = \sum_1^n a_i$. So $\sum_{i=1}^n b_{\sigma(i)} = \sum_{i=1}^n b_i \geq \sum_1^n a_i$.

Now, we shall show that $\sum_{i=1}^j a_i \leq \sum_{i=1}^j b_{\sigma(i)}$ for $1 \leq j < n$. Assume this is not true for some *j*. Let *j* be the least index such that $\sum_{i=1}^j a_i > \sum_{i=1}^j b_{\sigma(i)}$. Since $\sum_{i=1}^{j-1} a_i \leq \sum_{i=1}^{j-1} b_{\sigma(i)}$, it follows that $a_j > b_{\sigma(i)}$. Let *l* be the least integer such that $j < l \leq n$ and $a_j \neq a_l$. If no such *l* exists then, since $a_j = a_{j+1} = \cdots = a_n$ and $a_j > b_{\sigma(j)} \geq b_{\sigma(j+1)} \geq \cdots \geq b_{\sigma(n)}$, $\sum_{i=1}^n a_i > \sum_{i=1}^n b_{\sigma(i)}$. But we have just shown $\sum_{i=1}^n a_i \leq \sum_{i=1}^n b_{\sigma(i)}$. Hence, such an *l* must exist. We observe that $a_j = a_{j+1} = \cdots = a_{l-1} > a_l$ and $\sum_{i=1}^{l-1} a_i > \sum_{i=1}^{l-1} b_{\sigma(i)}$. Thus $\sum_{i=1}^{l-1} (t_i - a_i) < \sum_{i=1}^{l-1} (t_i - b_{\sigma(i)})$. One easily observes that $\sum_{i=1}^{l-1} b_i \leq \sum_{i=1}^{l-1} b_{\sigma(i)}$. So, $\sum_{i=1}^{l-1} (t_i - b_{\sigma(i)}) \leq \sum_{i=1}^{l-1} (t_i - b_i)$. If $l - 1 < m$ then, since EQUAL has failed to equalize jobs $l-1$ and *l*, it follows that $\sum_{i=1}^{l-1} (t_i - a_i) = \Delta * \sum_{i=1}^{l-1} s_i$. Furthermore, from Fact 1 it follows that in all valid assignments for Δ, the sum of the $l-1$ largest assignments is no greater than $\Delta * \sum_{i=1}^{l-1} s_i$. Hence, $\sum_{i=1}^{l-1} (t_i - b_i) \leq \Delta * \sum_{i=1}^{l-1} s_i$. This contradicts our earlier claim that $\sum_{i=1}^{l-1} (t_i - a_i) < \sum_{i=1}^{l-1} (t_i - b_i)$. Hence, *l* must be greater than *m*. But in this case job $l-1$ can always be equalized to job *l* unless this equalization requires more processing than available. Since $a_{l-1} > a_l$, it must be that $\sum_{i=1}^{l-1} (t_i - a_i) = \Delta * \sum_{i=1}^m s_i$. Since $\sum_{i=1}^{l-1} (t_i - b_i) \leq \Delta * \sum_{i=1}^m s_i$ we again obtain a contradiction. Thus, there can be no *l* for which $\sum_{i=1}^l a_i > \sum_{i=1}^l b_{\sigma(i)}$. □

LEMMA 2.2. *Let C be a set of n jobs with processing times c_i, $1 \leq i \leq n$. Let D be another set of of n jobs with processing times d_i, $1 \leq i \leq n$. Assume c_i and d_i are in nonincreasing order and $\sum_1^j c_i \leq \sum_1^j d_i$, $1 \leq j \leq n$. Let c_i' be the RPT of job i, $1 \leq i \leq n$, when set C is scheduled for a period Δ using* EQUAL. *Let d_i' be the RPT of job i, $1 \leq i \leq n$, when set D is scheduled for a period Δ using* EQUAL. *(Note that $c_i' \geq c_{i+1}'$ and $d_i' \geq d_{i+1}'$, $1 \leq i < n$.) Then,*

$$\sum_1^j c_i' \leq \sum_1^j d_i' \quad \text{for } 1 \leq j \leq n.$$

Proof. Assume the lemma is not true. Let *j* be the least index for which $\sum_1^j c_i' > \sum_1^j d_i'$. Then, $c_j' > d_j'$. Let *k* be the least index such that $j < k \leq n$ and $c_k' \neq c_j'$. There are two cases.

Case 1. There is no such *k*. In this case $c_j' = c_l'$ for $j < l \leq n$ and $\sum_1^n c_i' > \sum_1^n d_i'$. $c_n' > 0$ since $c_n' = c_j' > d_j' \geq 0$. Also, $\sum_1^n (c_i - c_i') < \sum_1^n (d_i - d_i')$. This means that EQUAL has assigned more total processing of the jobs in *D* than it has for the jobs in *C*. Let $x = \min\{n, m\}$. Since $c_n' > 0$, EQUAL must assign $\Delta * \sum_{i=1}^x s_i$ amount of processing for job set *C*. Also, no more than this amount can be assigned for *D*. Hence $\sum_{i=1}^n (c_i - c_i') \geq \sum_{i=1}^n (d_i - d_i')$.

Case 2. *k* as above exists. In this case $c_j' = c_{j+1}' = \cdots = c_{k-1}' > c_k'$ and $c_j' > d_j'$. Thus $\sum_{i=1}^l c_i' > \sum_1^l d_i'$ and $\sum_{i=1}^l (c_i - c_i') < \sum_{i=1}^l (d_i - d_i')$, $j \leq l < k$. $c_{k-1}' > c_k'$ can happen only if $\sum_{i=1}^{k-1} (c_i - c_i') = \Delta * \sum_{i=1}^{k-1} s_i$. But, in this case, $\sum_{i=1}^{k-1} (c_i - c_i') \geq \sum_{i=1}^{k-1} (d_i - d_i')$. □

LEMMA 2.3. *Let A and B be two sets of jobs. Let r_i, $1 \leqq i \leqq v$, be the distinct release times of the jobs in A and B. Assume that $r_i < r_{i+1}$ and that n_i jobs have release time r_i in both A and B, $1 \leqq i \leqq v$. Further, assume that the set of jobs with release time r_i in A is identical to that with release time r_i in B, $2 \leqq i \leqq v$. Let $C \cup D$ and $C \cup E$ be the job sets with release time r_1 in A and B respectively. Let $|D| = |E| = l$ and let d_i, e_i, $1 \leqq i \leqq l$, be the processing times for the jobs in D and E respectively. Assume $d_i \geqq d_{i+1}$ and $e_i \geqq e_{i+1}$, $1 \leqq i < l$. Also assume that $\sum_1^j d_i \geqq \sum_1^j e_i$, $1 \leqq j \leqq l$. If A has a DD-schedule then B also has one.*

Proof. Assume A has a *DD*-schedule S. The proof is by induction on the number of release times v. Let α_i and β_i, $1 \leqq i \leqq n$, be the processing times of the jobs in $C \cup D$ and $C \cup E$ respectively. Assume that $\alpha_i \geqq \alpha_{i+1}$ and $\beta_i \geqq \beta_{i+1}$, $1 \leqq i < n_1$.

Induction base. $v = 1$. Since $\sum_1^j d_i \geqq \sum_1^j e_i$, $1 \leqq j < l$, it follows that $\sum_1^j \alpha_i \geqq \sum_1^j \beta_i$, $1 \leqq j \leqq n_1$. Hence, from Fact 1 we conclude that the job set $C \cup E$ can be completed using no more time than $C \cup D$. So, if A has a *DD*-schedule then B must have one too.

Induction hypothesis. Assume the lemma is true for all job sets with v distinct release times for $1 \leqq v < u$.

Induction step. We shall show the lemma is true when $v = u$. A and B have n_1 jobs each at time r_1. Schedule both job sets $C \cup D$ and $C \cup E$ using EQUAL during the interval $[r_1, r_2]$. Let α_i' and β_i', $1 \leqq i \leqq n_1$ be the RPTs of these jobs at time r_2. Since $\sum_1^j \alpha_i \geqq \sum_1^j \beta_i$, $1 \leqq j \leqq n_1$, it follows from Lemma 2.2 that $\sum_1^j \alpha_i' \geqq \sum_1^j \beta_i'$, $1 \leqq j \leqq n_1$. Let A' and B' be the set of jobs remaining to be processed in A and B at time r_2. Let A'' be the corresponding set at time r_2 in the *DD*-schedule S. Further, let α_i'', $1 \leqq i \leqq n_1$, be the RPTs in S of the jobs in the set $C \cup D$ at r_2. Then, $\sum_1^j \alpha_i' \leqq \sum_1^j \alpha_i''$, $1 \leqq j \leqq n_1$, by Lemma 2.1. Since A'' has a *DD*-schedule, it follows that A' has one too (induction hypothesis). Now, since A' has a *DD*-schedule and both A' and B' have $u - 1$ release times and satisfy the conditions of the lemma, it follows that B' has a *DD*-schedule too. Hence, B has a *DD*-schedule. □

Our first nearly on line algorithm ONEDT utilizes EQUAL to schedule each phase $[r_i, r_{i+1}]$, $1 \leqq i \leqq v (r_{v+1} = d)$. It also uses two other subalgorithms ORDER and UNIFORM. ORDER sorts the jobs to be processed in each phase into nonincreasing order of their processing times. UNIFORM performs the actual scheduling of each job for the amount of processing time $t'(i)$ computed by EQUAL. Note that the conditional of lines 13 and 19 of EQUAL guarantees that this can be done. The algorithm UNIFORM is formally given in Gonzalez and Sahni [3].

THEOREM 2.1. ONEDT *generates a DD-schedule for every job set J for which such a schedule exists.*

Proof. The proof is by induction on the number of distinct release times in J and is very similar to that of Lemma 2.3.

Analysis of EQUAL. The loop of lines 7–9 contributes at most $O(p)$ to the algorithms complexity. The total number of iterations of the loop of lines 15–29 is at most p. The conditional of line 18 can be true at most m times. Each time this happens, $O(m)$ time may be spent in lines 19–27. Hence, the total contribution of lines 15–29 is $O(p + m^2)$. The complexity of EQUAL is also $O(p + m^2)$.

Analysis of ONEDT.

Time complexity. *Let n_i, $1 \leqq i \leqq v$,* be the number of jobs released at the distinct release times r_i, $1 \leqq i \leqq v$. Line 5 takes $O(n_i \log n_i + \sum_{j=1}^{i-1} n_j)$ time since we only sort the jobs released at r_i and merge these n_i jobs with the uncompleted jobs released from r_1 through r_{i-1}. Note that these jobs are already in nonincreasing order of their RPTs. Line 6 takes $O(N_i + m^2)$ where $N_i = \sum_1^i n_j$. Line 7 takes $O(N_i)$. Therefore the time complexity of ONEDT is $O(n \log n + nv + vm^2) = O(n^2 + nm^2)$.

Number of preemptions. We have v distinct release times. UNIFORM generates at most $2(m-1)$ preemptions [3] during the interval $[r_i, r_{i+1}]$. Additional preemptions are introduced since some of the jobs to be processed in $[r_i, r_{i+1}]$ may have been processed for some time in a previous interval. There are at most N_{i-1} such additional preemptions. Therefore, the total number of preemptions is at most $\sum_1^v (2(m-1)+N_{i-1}) = O((m+n)v) = O(mn+n^2)$. □

ALGORITHM 2.3. First algorithm for *DD*-schedules.

```
line    procedure ONEDT(d)
            //d is the common due time for all the jobs//
 1          R ← set of jobs released at time 0
 2          t ← 0
 3          loop
 4           r ← min {next release time, d}
 5           call ORDER(R)
 6           call EQUAL (r − t, S, R, m, p, T)
                    //S : cumulative speed of processors//
                    //m: number of processors//
                    //p : number of jobs//
                    //T : computed processing time//
 7           call UNIFORM (m, p, T)
 8           if r = d then exit endif
 9           Q ← set of jobs released at r
10           R ← Q ∪ R
11           t ← r
12          repeat
13          if all the jobs are completed then print ("DD-schedule exists")
14          else print ("No DD-schedule for the job set")
15          endif
16      end ONEDT
```

The total number of preemptions may be reduced by using the equalization strategy only until $2m - mlo + 1$ jobs have an equal RPT. The new algorithm, MEQUAL, to determine the scheduling assignments for each phase is described informally. MEQUAL behaves as EQUAL until *nhi* becomes greater than $2m - mlo + 1$. At this time jobs $mlo, mlo+1, \cdots, nhi-1$ have the same RPT. In case all the processing time has been assigned (line 2) then MEQUAL terminates. In this case MEQUAL is identical to EQUAL. In line 3 we increase $t'(i)$, $mlo \leqq i \leqq m$ using the equalizing rule and keeping in mind that $t'(i)$, $m < i < nhi$ has also been assigned to this interval. This step behaves as if $t(m+1) = 0$. If at the end of this step, $t(m) \neq t'(m)$ then it must be that $\sum_{i=1}^{nhi-1} t'(i) = \Delta * \sum_{i=1}^{m} s_i$ and $(t(i) - t'(i)) \geqq (t(i+1) - t'(i+1))$, $1 \leqq i \leqq m$. If $t'(m) = t(m)$ then the RPT of job m is zero. In lines 6–12 we assign as much of jobs *nhi* to p as possible. Note that since $t'(m) = t(m) \geqq t(nhi)$, the assignments made in lines 6–12 cannot violate Fact 1 with $w = \Delta$. If all of jobs *nhi* to p can be assigned then, the remaining time of the processors is allocated equally to jobs $m+1$ to $nhi-1$. Thus, whenever MEQUAL behaves differently from EQUAL the RPTs of jobs $m+1$ to $nhi-1$ is the same at termination. There are at least $m - mlo + 1$ such jobs.

ALGORITHM 2.4. Final equalizing rule for *DD*-schedules.

```
line    procedure MEQUAL
 1          use the equalizing rule, EQUAL, until it either terminates or
            nhi > 2m − mlo + 1 (line 10). If EQUAL terminates then return.
```

```
2       if ∑_{i=1}^{nhi-1} t'(i) = Δ * S(m) then t(i) ← t(i) − t'(i), 1 ≦ i < nhi;
        return; endif
        nhi ← 2m − mlo + 2
3       continue to increase the assignments t'(i), mlo ≦ i < m using
        the equalizing rule with the assumption t(m + 1) = 0. Now, the
        equalizing rule will terminate either with t'(m) = t(m) or
        t'(m) ≠ t(m). If t'(m) ≠ t(m) then ∑_{i=1}^{nhi-1} t'(i) = Δ * S(m)
4       if t'(m) ≠ t(m) then t(i) ← t(i) − t'(i), 1 ≦ i < nhi
                             return
                        endif
5       AMTLFT ← Δ * S(m) − ∑_{i=1}^{nhi-1} (t(i) − t'(i))
6       for i ← nhi to p do
7           if AMTLFT > t(i) then t'(i) ← t(i); AMTLFT ← AMTLFT − t(i)
8                            else t'(i) ← AMTLFT
                                  t(j) ← t(j) − t'(j), 1 ≦ j ≦ i
10                                return
11                           endif
12      repeat
13      for i ← m + 1 to nhi − 1 do
14          t'(i) ← t'(i) + AMTLFT/(nhi − m − 1)
15      repeat
16      t(i) ← t(i) − t'(i), 1 ≦ i ≦ p
17      return
18  end MEQUAL
```

Example 2.2. Assume we have 3 machines with speeds $S = \{3, 2, 1\}$, 7 jobs with processing times $t = \{10, 9, 8, 7, 6, 5, 4\}$ and $\Delta = 7$.

If we use EQUAL, the RPTs of these jobs after scheduling are $\{1, 1, 1, 1, 1, 1, 1\}$. When we use MEQUAL, we will first equalize the 6 largest jobs to have an RPT of 5. Next, MEQUAL will continue to use EQUAL but with only the first 3 jobs. The RPTs of these 3 jobs will be 0 after this step. We have used up 30 units of processing time out of 42 units we are given for the interval. The last job with RPT 4 is now assigned (lines 6–10) and is completed. We have 8 units of processing time left. We execute the jobs with initial RPTs 7, 6 and 5 this time (lines 3–5) for 8/3 more units. The final RPTs of the jobs by MEQUAL are $\{0, 0, 0, 7/3, 7/3, 7/3, 0\}$. Note that the minimum finish time for both sets is the same and is 7/6. □

The following lemma shows that the processing assignments made by MEQUAL can be completed within the given interval.

LEMMA 2.4. *$t'(i)$, $1 \leqq i \leqq p$, generated by* MEQUAL *are such that they can be completed in Δ units of processing time.*

Proof. We have 2 different cases according to when return of control is made by MEQUAL.

Case 1. The return of the control is made at one of steps 1, 2 or 4. We have assigned only using EQUAL thus far and EQUAL generates only valid sets of processing assignments.

Case 2. Return of control is made at either line 10 or line 17. In this case $t'(j) < t'(i)$, $j > m$ and $i \leqq m$. Since, $\max_{j \leqq m} \{\sum_{i=1}^{j} t'(i)/\sum_{i=1}^{j} s_i\} \leqq \Delta$ is guaranteed by EQUAL and from lines 6–15 it follows that $\sum_{i=1}^{p} t'(i) \leqq \Delta * \sum_{1}^{m} s_i$, Fact 1 guarantees that the $t'(i)$'s are a valid assignment set. □

Let t_i, $1 \leqq i \leqq p$, $t_i \geqq t_{i+1}$, $1 \leqq i < p$, be any set of p processing times. If the corresponding task set is used by EQUAL then let a_i, $1 \leqq i \leqq p$, be the RPTs. Let b_i,

$1 \leqq i \leqq p$, be the RPTs when MEQUAL is used. We have seen, earlier, that $a_i \geqq a_{i+1}$, $1 \leqq i < p$. Assume that MEQUAL does not terminate in lines 1 or 2. It is easy to see that $a_i = b_i$, $1 \leqq i < mlo$. Also, there may exist a k, $mlo \leqq k \leqq m$, such that $b_k > b_{m+1}$. If such a k exists then we can show that $b_{mlo} \geqq b_{mlo+1} \geqq \cdots \geqq b_k$ and $a_i \geqq b_i$, $mlo \leqq i \leqq k$. This follows from the observation that MEQUAL tries to use all remaining space to increase the assignments of only jobs mlo to m whereas EQUAL uses this same space to increase the assignments of many more tasks. Let $\sigma(\cdot)$ be such that $b_{\sigma(i)} \geqq b_{\sigma(i+1)}$, $1 \leqq i < p$. From the preceding discussion and the knowledge that $b_{m+1} = b_j$, $m+1 \leqq j \leqq 2m - mlo + 1$, we can conclude that $a_i \geqq b_{\sigma(i)}$, $1 \leqq i \leqq j$, and $b_{\sigma(j+1)} = b_{\sigma(j+2)} = \cdots = b_{\sigma(m)}$.

LEMMA 2.5. *Let A, B, C, D, E, d_i, e_i and l be as defined in Lemma 2.3. Assume that $d_i \geqq e_i$, $1 \leqq i \leqq j$, and $e_{j+1} = e_i$, $j+1 < i \leqq m$. Further, assume that $\sum_1^l d_i = \sum_1^l e_i$ and $l > m$. If A has a DD-schedule then B also has one.*

Proof. First assume that the number of distinct release times in A and B is 1. Sort $A = C \cup D$ and $B = C \cup E$ into nonincreasing order of processing times. Let these times respectively be α_i and β_i, $1 \leqq i \leqq n_1$. $\alpha_i \geqq \alpha_{i+1}$ and $\beta_i \geqq \beta_{i+1}$, $1 \leqq i < n_1$. Let r be the largest index such that $\alpha_i \geqq \beta_i$, $1 \leqq i \leqq r$. If $r < m$ then from the assumptions on D and E, it follows that $\beta_{r+1} = \beta_j$, $r+1 < j \leqq m$. When $r \geqq m$, the lemma is proved by using Fact 1 and the knowledge, $\alpha_i \geqq \beta_i$, $1 \leqq i \leqq r$, and $\sum_1^{n_1} \alpha_i = \sum_1^{n_1} \beta_i$. When $r < m$, we use the additional information $\sum_1^j \beta_i / \sum_1^j s_i \leqq \sum_1^{j+1} \beta_i / \sum_1^{j+1} s_i$, $r < j < m$, and $\sum_1^m \beta_i / \sum_1^m s_i < \sum_1^l \beta_i / \sum_1^l s_i = \sum_1^l \alpha_i / \sum_1^l s_i$.

Now assume the lemma is true for all job sets A and B with $u < v$ release times. We shall show the lemma is true for all job sets with v release times. Let α_i and β_i be as above. Use EQUAL to compute the assignments for $C \cup D$ and $C \cup E$ in the interval $\Delta = r_2 - r_1$. Let α_i' and β_i' be the respective RPTs. From the working of EQUAL, it follows that $\alpha_i' \geqq \beta_i'$, $1 \leqq i \leqq r$ and if $r < m$ then $\beta_{r+1}' = \beta_j'$, $r+1 < j \leqq m$. Let A' and B' be the job sets remaining at r_2 following the use of EQUAL in $[r_1, r_2]$. It follows that A' and B' satisfy the conditions of the lemma and have only $v-1$ release times. Also, A' has a DD-schedule. So, B' and hence B have DD-schedules. □

Let MONEDT be the algorithm resulting when line 6 of ONEDT is replaced by a call to MEQUAL.

THEOREM 2.2. MONEDT *generates a DD-schedule for every job set J for which such a schedule exists.*

Proof. The proof is similar to that of Lemma 2.5 and uses the results of the discussion preceding this lemma. □

Analysis of MONEDT. The complexity of MONEDT, is the same as that of ONEDT, i.e., $O(n^2 + nm^2)$. Its complexity can be easily reduced to $O(m^2 n + mn \log n)$ by using a heap. The changes needed to ONEDT to get the improved MONEDT are:

(i) delete line 5;
(ii) change EQUAL to MEQUAL in line 6;
(iii) maintain a max-heap of jobs with nonzero RPT;
(iv) in line 10 insert the Q into this heap;

The heap insertion of change (iv) requires $O(|Q| \log n)$ time per iteration. Hence its overall contribution to the computing time is $O(\sum (|Q| \log n)) = O(n \log n)$. We now need to modify EQUAL so that when it is called from line 1 of MEQUAL, the job times are obtained from a heap. This requires the insertion of an instruction between lines 7 and 8 to delete an element from the max-heap and to set $t(i)$. A check for $i > m - mlo + 1$ is also made. When this happens a jump to line 10 followed by a return to MEQUAL is made. Since only $O(m)$ items are deleted from the heap, the total time spent on this call to EQUAL is $O(m^2 + m \log n)$. When EQUAL is used from line 3 of

MEQUAL it works as before (i.e. using the times $t(i)$, $mlo \leqq i \leqq m$ rather than extracting times from the heap.) Hence, the time for line 3 of MEQUAL is $O(m^2)$. Lines 4 and 5 take $O(m)$ time. If the loop of lines 6–12 is iterated k_i times on the ith call to MEQUAL then the time needed is $O(k_i \log n)$ to extract the next k_i times from the heap plus $O(nhi \log n) = O(m \log n)$ time to insert the nonzero RPTs back into the heap (line 9) in case a return is made from this loop. If a return is made from line 17 instead then the total time spent in lines 13–15 is $O(m)$. Line 16 requires reinsertion of the nonzero RPTs into the heap. There can be at most $nhi - 1$ such RPTs as lines 13–17 are executed only when all jobs indexed nhi to p are fully allocated in lines 6–12. So the time needed in lines 13–17 is $O(m \log n)$. Hence, the ith call to MEQUAL takes times $O(m^2 + (m + k_i) \log n)$. If there are v release times then the total time spent in MEQUAL is $O(m_2 v + (mv + \sum k_i) \log n)$. Note that when the loop of lines 6–12 of MEQUAL are iterated, at least $k_i - 1$ jobs have a zero RPT and so are not considered in future iterations. Hence, $\sum k_i = O(n)$. Also $v \leqq n$. Therefore the time spent in MEQUAL is $O(m^2 n + mn \log n)$. The total time spent in UNIFORM is less than this. So, the overall complexity of MONEDT is $O(m^2 n + mn \log n)$.

If v is the number of release times then UNIFORM introduces at most $O(mv)$ preemptions. At most $2m + 1$ of the jobs scheduled in any interval may remain uncompleted by the end of the interval. This results in at most $2m + 1$ additional preemptions per phase (except the last phase when all jobs must be completed). The total number of preemptions is therefore $O(mv) = O(mn)$. This bound of $O(mn)$ agrees with the lower bound established in [10] on the worst case number of preemptions. □

Acknowledgment. We are grateful to an anonymous referee for suggesting the use of a heap to reduce the complexity of MONEDT from $O(n^2 + nm^2)$ to $O(m^2 n + mn \log n)$. The corresponding analysis was also provided by the referee.

REFERENCES

[1] J. BRUNO AND T. GONZALEZ, *Scheduling independent tasks with release dates and due dates on parallel machines*, Technical Report No. 213, Pennsylvania State University, State College, Dec. 1976.

[2] T. GONZALEZ AND D. JOHNSON, *A new algorithm for preemptive scheduling of trees*, Technical Report No. 222, Pennsylvania State University, State College, June 1977.

[3] T. GONZALEZ AND S. SAHNI, *Preemptive scheduling of uniform processor systems*, J. Assoc. Comput. Mach., 25 (1978), pp. 92–101.

[4] W. HORN, *Some simple scheduling algorithms*, Naval Res. Logist. Quart., 21 (1974), pp. 177–185.

[5] E. HORVATH, S. LAM AND R. SETHI, *A level algorithm for preemptive scheduling*, J. Assoc. Comput. Mach., 24 (1) (1977), pp. 32–43.

[6] A. RINNOOY KAN, *Machine scheduling problems*, Ph.D. thesis, Mathematische Centrum, Amsterdam, 1976.

[7] J. LIU AND C. LIU, *Bounds on scheduling algorithms for heterogeneous computing systems*, IFIP Proceedings, August 1974, pp. 349–353.

[8] R. MCNAUGHTON, *Scheduling with deadlines and loss functions*, Management Sci., 12 (1959), pp. 1–12.

[9] S. SAHNI, *Preemptive scheduling with due dates*, Technical Report #77-4, University of Minnesota, Minneapolis, April 1977; Operations Res., to appear.

[10] S. SAHNI AND Y. CHO, *Scheduling independent tasks with due times on a uniform processor system*, Technical Report #77-7, University of Minnesota, Minneapolis, May 1977.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0001 $01.00/0

AUTOMATIC ASYMPTOTIC AND BIG-*O* CALCULATIONS VIA COMPUTER ALGEBRA*

DAVID R. STOUTEMYER†

Abstract. A computer program able to approximate symbolic expressions asymptotically is described. More precisely, the program determines, as requested, simpler expressions which are either asymptotically equal, of the same exact order, of at least the same order, of greater order, of lesser order, or of at most the same order as the given expression. Additional features allow the combination of exact and approximate subexpressions in the proper manner. The program is intended as a tool for complexity analysis, numerical analysis, and wherever approximations are used.

Key words. asymptotic, *O*, *o*, order, complexity analysis, numerical analysis, approximation theory, computer algebra, MACSYMA, simplification

1. Introduction. Complexity analysis has long benefited computer algebra by serving as a tool for the analysis and a guide for the design of symbolic algorithms. This paper describes a computer-algebra program which returns the favor by providing a tool for complexity analysis.

Among the mathematical tasks of complexity analysts are:

1. The discovery of closed-form representations for indefinite finite sums.
2. The discovery of closed-form or asymptotic solutions to recurrence relations.
3. The simplification of complicated expressions by the reduction of expressions to simpler ones which are asymptotically equal, of the same exact order, of at least the same order, of greater order, of lesser order, or of at most the same order.
4. The simplification of expressions containing one or more approximate subexpressions of the above types.

The MACSYMA computer-algebra system, described by the Mathlab group [9], has two built-in functions which are helpful for task 1, one of which is described by Gosper [3], [4]. Moenck [10] describes another algorithm for this task. Ivie [7] describes a MACSYMA program which is helpful for task 2, whereas Cohen and Katcoff [2] describe an analogous REDUCE computer-algebra program.

A MACSYMA program for tasks 3 and 4 is described here. Section 2 outlines the mathematical and programming techniques, while § 3 contains a brief demonstration of the program. Section 4 summarizes the performance for some more complicated examples, with conclusions in § 5.

2. Mathematical and programming techniques. The notion of approximation is ubiquitous and invaluable in mathematics. Among the reasons are:

1. Approximations often enable analysis to proceed when it would otherwise be impossible or impractical.
2. Approximations often permit a concise summary of the most crucial features of a result which is incomprehensibly complicated in its exact form.
3. Physical system models are frequently approximate anyway.

2.1. Asymptotic simplification via limits. Many concepts and notations of varying rigor have been devised to facilitate a concise treatment of approximations. One of the most useful of such concepts is that of asymptotic equality:

* Received by the editors May 23, 1978. This work was supported by the National Science Foundation under grants MCS75-22983 and MCS7802234, by the United States Energy Research and Development Administration under contract number E(11-1)-3070, and by the National Aeronautics and Space Administration under Grant NSG 1323.

† Electrical Engineering Department, University of Hawaii, Honolulu, Hawaii 96822.

DEFINITION. For a given vector of limit points $\hat{\mathbf{x}} = (\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_k)$, we say that $g(x_1, x_2, \cdots, x_k) = g(\mathbf{x})$ is **asymptotically equal** to $f(\mathbf{x})$, denoted $g(\mathbf{x}) \sim f(\mathbf{x})$, if

$$\lim_{\mathbf{x} \to \hat{\mathbf{x}}} (g(\mathbf{x})/f(\mathbf{x})) = 1,$$

provided the limit exists.

$\mathbf{x}$ and $\hat{\mathbf{x}}$ are often contextually implied rather than explicitly stated. For example in complexity analysis $\mathbf{x}$ is frequently a vector of integer-valued variables with $\hat{\mathbf{x}} = (\infty, \infty, \cdots, \infty)$, whereas for numerical analysis $\mathbf{x}$ is frequently a vector of real or complex variables with $\hat{\mathbf{x}} = (0, 0, \cdots, 0)$.

As an approximation, it is often desirable to replace a complicated expression by the simplest asymptotically equal expression. Moreover, it is often desirable to mix such approximations with exact quantities in a disciplined fashion. Accordingly, it is frequently convenient to replace a complicated subexpression $g(\mathbf{x})$ by a functional form such as ASYMP($f(\mathbf{x})$), where $f(\mathbf{x})$ is simpler than $g(\mathbf{x})$, and for a contextually implied limit point, is asymptotically equal. For example, we could approximate the infinite Taylor series for e^t by

$$1 + t + \text{ASYMP}(t^2/2) \tag{1}$$

as t approaches 0. Thus, ASYMP($f(\mathbf{x})$) denotes some $g(\mathbf{x})$ such that $g(\mathbf{x}) \sim f(\mathbf{x})$. Alternatively, it is sometimes more suitable to let ASYMP($f(\mathbf{x})$) denote the set of all $g(\mathbf{x})$ such that $g(\boldsymbol{x}) \sim f(\mathbf{x})$. For example, under this interpretation expression (1) denotes the set of all functions $\{1 + t +$ any function asymptotically equal to $t^2/2\}$. The techniques described below are applicable to either interpretation.

MACSYMA programs have been developed which attempt to simplify expressions of the form ASYMP($g(\mathbf{x})$) to simpler forms ASYMP($f(\mathbf{x})$) where $f(\mathbf{x}) \sim g(\mathbf{x})$ but $f(\mathbf{x})$ is in some structural sense simpler than $g(\mathbf{x})$. The user establishes $\hat{\mathbf{x}}$ by commands of the form PUT(x_j, $\hat{x}_j$, LIMIT) before using "ASYMP". Any variables without such established limits are assumed to represent constants.

Simplicity is in the eye of the beholder, and the notion of simplicity imbedded in the algorithm for ASYMP is implicit in the following description of that algorithm:

First, the function uses the MACSYMA built-in RATSIMP function together with appropriate settings of the RATEXPAND and RATDENOMDIVIDE flags to express the argument as a ratio of two relatively-prime fully-expanded expressions. Then, separately for the numerator and denominator, the algorithm discards terms which are strictly dominated by any other terms. A term u **strictly dominates** a term v if

$$|L| = \left| \lim_{\mathbf{x} \to \hat{\mathbf{x}}} (u/v) \right| = \infty. \tag{2}$$

Hardy [6] gives a thorough discussion of the orders of infinity, and Wang [11] describes how these considerations are incorporated into the built-in MACSYMA LIMIT function. This function computes limits for a single real variable and must sometimes ask the user for sign information about any other variables that occur in its arguments. Consequently, in order to utilize this function for multivariate limits, the program attempts to partition u/v into factors $r_0, r_1, \cdots, r_q$, such that:

1. r_0 is the product of all factors which do not contain any asymptotic variables.
2. Each asymptotic variable occurs in at most one of the factors r_1 through r_q.

Construction of r_0 through r_q from the factors of u/v is an application of the well-known "equivalencing" or "disjoint-union-find" problem, as discussed by Aho, Hopcroft and Ullman [1].

Computation of multivariate limits as nested univariate limits is not necessarily valid, so if any of these factors contains more than one asymptotic variable, all terms are retained and the user is advised to try again using the series-expansion alternative described in § 2.2. Otherwise, without loss of generality, let x_j denote the asymptotic variable in r_j, having the limit point $\hat{x}_j$, and let

$$l_j = \lim_{x_j \to \hat{x}_j} r_j.$$

Equation (2) can be regarded as true if and only if

$$l_j \neq 0 \quad \text{for } j = 1, 2, \cdots, q;$$

and

$$|l_k| = \infty \quad \text{for some } 1 \leqq k \leqq q.$$

Conversely, v strictly dominates u if $|l_j| \neq \infty$ for $j = 1, 2, \cdots, q$; $l_j = 0$ for some $1 < k < q$.

However, if this technique results in the retention of more than one term, then asymptotic cancellation may make some of any discarded terms significant. For example, as $n \to \infty$,

$$\sqrt{n^4+1} - n^2 + n - 1 \not\sim \sqrt{n^4+1} - n^2.$$

Consequently, the set of terms retained by the above dominance testing is partitioned into equivalence classes, where in each class $0 < |L| < \infty$ for every pair of terms in the class. Let the members of such a class be designated $u_1, u_2, \cdots, u_p$, and let

$$L_j = \lim_{\mathbf{x} \to \hat{\mathbf{x}}} u_j / u_1, \quad \text{for } j = 1, 2, \cdots, p.$$

Then cancellation occurs if $\sum_{j=1}^{p} L_j = 0$.

When cancellation occurs, all terms are retained and the user is advised to try again using the alternative series expansion technique described below. Otherwise the sum of the terms in the class is represented by $u_1 \sum_{j=1}^{p} L_j$, choosing for u_1 the member which makes this representation least complex. Complexity is measured as the number of atoms in the internal representation of an expression, and the complexity measure function is easily changed if the user prefers another measure. Consequently, ASYMP($\sqrt{n^4+1} - n^2 + n - 1$) remains as is, whereas ASYMP($2\sqrt{n^4+1} - n^2 + n - 1$) simplifies to ASYMP($n^2$).

As a final step, the ratio of the simplified numerator and denominator is reduced to lowest terms.

For example, if m and n asymptotically approach ∞ while ε asymptotically approaches 0, then

$$\text{(3)} \qquad \text{ASYMP}\left(\frac{am^3n + \pi m^2n^2 + m^2n + \log^3 n}{5m\varepsilon + 3\varepsilon^2}\right) \to \text{ASYMP}\left(\frac{am^2n + \pi mn^2}{5\varepsilon}\right),$$

meaning that the expression left of the arrow simplifies to the expression on the right.

2.2. Asymptotic simplification via series expansions. The built-in TAYLOR function described by Zippel [12] provides an alternative technique. This function yields multivariate power-series expansions of a specified order for a large range of expressions. TAYLOR can accommodate poles, logarithmic and fractional-power singularities, but not essential singularities. Moreover, TAYLOR permits expansions about infinity provided the value is finite and without an essential singularity there. The

restriction to a specified order, as opposed to a specified number of nonzero terms, means that TAYLOR may in some cases have to be invoked with increasing order until a nonzero term first appears. To prevent an infinite or prohibitively expensive loop, the number of attempts is limited to the value of a global variable named MAXTAYLOR. The limitations on expansions at infinity means that if TAYLOR initiates an error interrupt due to an infinite value at infinity then TAYLOR should be reattempted on the reciprocal of the given expression (returning the reciprocal of the series answer). Because essential singularity will provoke error returns during our attempt to expand expressions such as e^n or e^{-n} about ∞, we use the built-in MACSYMA ERRCATCH function to trap and recover gracefully from these "errors". For these reasons, the TAYLOR approach is scheduled before the dominance techniques, but the default setting of MAXTAYLOR is 0. If the default setting yields an insufficiently simplified result, the user can try again with MAXTAYLOR set to some modest positive integer. Despite its present limitations and difficulties, the TAYLOR technique is sometimes capable of generating simpler answers than the other technique described here, particularly when the expanded numerator or denominator of the given expression contains multi-term functional subexpressions such as $\log(m+1)$ or $\sqrt{m^2+3m+1}$.

Perhaps the TAYLOR program could be generalized to suffice alone.

2.3. Mixtures of exact and approximate expressions. The simplifications done by the ASYMP function can be called intra-simplifications, because they occur within the argument of ASYMP. There are also applicable inter-simplifications between asymptotic subexpressions and exact subexpressions or other asymptotic subexpressions. Subexpressions of the form ASYMP(u) can be treated algebraically just as any other functional form, with some important exceptions:

$$\frac{\text{ASYMP}(u)}{\text{ASYMP}(u)} \to \text{ASYMP}(1), \tag{4}$$

rather than simplifying to 1. Even more dramatic is the rule

$$\text{ASYMP}(u) - \text{ASYMP}(u) \to o(u), \tag{5}$$

rather than simplifying to zero, where we have the following

DEFINITION. For a given $\hat{\mathbf{x}}$ and $f(\mathbf{x}) > 0$, $o(f(\mathbf{x}))$ denotes some $g(\mathbf{x})$ (or the set of all $g(\mathbf{x})$) such that

$$\lim_{\mathbf{x}\to\hat{\mathbf{x}}} (g(\mathbf{x})/f(\mathbf{x})) = 0,$$

provided the limit exists. Besides the ordinary algebraic simplification rules, modified by rules (4) and (5), there are the additional consolidation rules:

$$\text{ASYMP}(u) + \text{ASYMP}(v) \to \text{ASYMP}(u+v), \quad \text{for } u \neq -v, \tag{6}$$

$$\text{ASYMP}(u)^v \to \text{ASYMP}(u^v) \quad \text{for } u, v > 0, \tag{7}$$

$$\text{ASYMP}(u)v \to \text{ASYMP}(uv), \tag{8}$$

$$\text{ASYMP}(u)\text{ASYMP}(v) \to \text{ASYMP}(uv), \tag{9}$$

$$\text{ASYMP}(\text{ASYMP}(u)) \to \text{ASYMP}(u), \tag{10}$$

$$\text{ASYMP}(u)/\text{ASYMP}(v) \to \text{ASYMP}(u/v). \tag{11}$$

These rules are reversible, yielding corresponding decomposition rules, but the program uses only consolidation.

Regrettably, the MACSYMA pattern-matcher rejects attempts to establish rules (4) and (5). Therefore, the user is advised not to generate expressions containing a cancellable pair of two instances of the same asymptotic subexpression. The same unwanted transformations would probably occur on almost any computer-algebra system, and it is unclear how to esthetically inhibit them without extensive tampering with the built-in simplifier.

To achieve simplifications (6) through (11), the program includes a function named ASYMPSIMP, which scans expressions from the bottom up, applying these simplifications at every opportunity.

Sometimes the constants within a result returned by ASYMP are complicated, or they are of no interest. The θ notation introduced by Knuth [8] usually permits us to suppress these constants, retaining only the more important information indicating the exact order of an approximation:

DEFINITION. For a given $\hat{\mathbf{x}}$ and positive $f(\mathbf{x})$, $\theta(f(\mathbf{x}))$ denotes some $g(\mathbf{x})$ (or the set of all $g(\mathbf{x})$) such that there exist positive constants c and C together with a neighborhood of $\hat{\mathbf{x}}$ where $cf(\mathbf{x}) \leqq g(\mathbf{x}) \leqq Cf(\mathbf{x})$ for all $\mathbf{x}$ in this neighborhood.

The asymptotic analysis package contains a corresponding function named THETA, analogous to ASYMP. THETA attempts to simplify expressions of the form THETA($g(\mathbf{x})$) to forms THETA($f(\mathbf{x})$) when $f(\mathbf{x}) = \theta(g(\mathbf{x}))$ but $f(\mathbf{x})$ is in some sense structurally simpler than $g(\mathbf{x})$.

THETA uses the same algorithm as ASYMP, except THETA also removes the content with respect to all asymptotic variables in the numerator and denominator of its argument.

For example, if m and n asymptotically approach ∞,

$$\text{THETA}(2m^2n + 4mn^2) \to \text{THETA}(m^2n + 2mn^2).$$

There are inter-simplifications for THETA analogous to rules (4) and rules (6) through (11). However, the analog of rule (5) is

$$\text{THETA}(u) - \text{THETA}(u) \to O(u), \tag{12}$$

where we have the following

DEFINITION. For a given $\hat{\mathbf{x}}$ and positive $f(\mathbf{x})$, $O(f(\mathbf{x}))$ denotes some $g(\mathbf{x})$ (or the set of all $g(\mathbf{x})$) such that there exist a positive constant C and a neighborhood of $\hat{\mathbf{x}}$ with $|g(\mathbf{x})| \leqq Cf(\mathbf{x})$ for all $\mathbf{x}$ in the neighborhood.

Whenever THETA and ASYMP interact, such as in ASYMP(u)THETA(v) or THETA(ASYMP(u)), we can demote the ASYMP to the less precise THETA, then proceed with any applicable further simplifications.

The aforementioned ASYMPSIMP function also performs these demotions together with the THETA analogs of rules (6) through (11). For similar reasons, rule (12) and the analog of rule (4) are not implemented. Consequently, there is a similar caveat to avoid generating expressions containing a cancellable pair of two instances of the same THETA subexpressions.

We have already defined little-oh and big-oh, and the program package contains corresponding functions respectively named LO and O. Here is the story of O:

MACSYMA has no built-in facilities which encourage direct use of the definition of O, so the program used LIMIT in an attempt to simplify the subset of problems for which appropriate limits can be computed.

First O uses THETA; then for each asymptotic variable together with each term of the numerator of the returned argument of THETA, O uses LIMIT to compare the products of all explicit factors containing the variable and no other asymptotic variable,

retaining the dominant one of these products. If any factor contains more than one asymptotic variable, then all terms containing those variables are retained. The same is done for the denominator, except the most subordinate of these products is retained for each variable. For example if m and n approach ∞,

$$O\left(\frac{am^2n + mn^2 + mn}{\log^3 n + \log\log m}\right) \to O(m^2n^2).$$

There are inter-simplifications for O analogous to rules (6) through (10). However, the analog of rules (4), (5) and (11) are respectively

$$O(u) - O(u) \to O(u), \tag{13}$$

$$O(u)/O(u) \to \text{undefined}, \tag{14}$$

$$O(u)/O(v) \to \text{undefined}. \tag{15}$$

Also, whenever THETA or ASYMP interact with O, THETA or ASYMP can be demoted to O, after which further simplification can proceed.

For ASYMP, THETA, and O we have used the same function name to designate simplification and to designate the simplified result, because the simplification processes are idempotent: ASYMP(ASYMP(u)) $\equiv$ ASYMP(u), etc. However, when a user inputs $o(g(\mathbf{x}))$, he might want either of two things returned:

1. An expression $o(f(\mathbf{x}))$, where $f(\mathbf{x})$ is the simplest expression which strictly dominates $g(\mathbf{x})$.
2. An expression $o(f(\mathbf{x}))$, where $f(\mathbf{x})$ is the simplest expression which strictly dominates any function that $g(\mathbf{x})$ strictly dominates.

I have chosen the second interpretation for its idempotency and its ease of implementation, but it is not clear yet whether or not most users would prefer the first alternative or an opportunity to impose either one.

With the second alternative, the intra-simplification for o is similar to that for O.

The inter-simplifications for o are analogous to those for O. Also in composition or products with o, we can absorb ASYMP, θ, or O. For example:

$$\theta(o(u)) \to o(u), \tag{16}$$

$$o(\theta(u)) \to o(u), \tag{17}$$

$$o(u)O(v) \to o(uv). \tag{18}$$

Otherwise, we can demote o to O in interactions with ASYMP, θ, and O. For example:

$$o(u) + \theta(v) \to O(u) + \theta(v) \to O(u + v). \tag{19}$$

O and o respectively provide an upper bound and a strict upper bound on the order of an approximation. In addition or instead, we are sometimes interested in a lower bound or a strict lower bound on the order. Halton [5] and Knuth [8] have independently suggested the following notations for this purpose:

DEFINITIONS. For a given $\hat{\mathbf{x}}$ and a positive $f(\mathbf{x})$, $\omega(f(\mathbf{x}))$ denotes some $g(\mathbf{x})$ (or the set of all $g(\mathbf{x})$) such that

$$\lim_{\mathbf{x}\to\hat{\mathbf{x}}} (f(\mathbf{x})/g(\mathbf{x})) = 0.$$

DEFINITION. For a given $\hat{\mathbf{x}}$ and positive $f(\mathbf{x})$, $\Omega(f(\mathbf{x}))$ denotes some $g(\mathbf{x})$ (or the set of all $g(\mathbf{x})$) such that there exist a positive constant c and a neighborhood of $\hat{\mathbf{x}}$ with $cf(\mathbf{x}) \leqq |g(\mathbf{x})|$ for all $\mathbf{x}$ in the neighborhood.

The intra and inter simplifications of ω and Ω are quite similar to their respective o and O counterparts, as are the interactions of ω and Ω with each other and with ASYMP or θ.

The reciprocal nature of O with Ω and o with ω permits the following conversions:

$$1/(\Omega(1/u)) \leftrightarrow O(u), \tag{20}$$

$$1/(\omega(1/u)) \leftrightarrow o(u). \tag{21}$$

However, it is uncertain which direction, if any, is universally best, so these conversions are not included in the program.

Consolidation rules (7) and (8) combine exact with approximate operands of multiplication and exponentiation, which can also be done for addition. For $u \neq -v$:

$$u + \text{ASYMP}(v) \to \text{ASYMP}(u+v), \tag{22}$$

$$u + \theta(v) \to \theta(u+v), \tag{23}$$

$$u + O(v) \to O(u+v), \tag{24}$$

$$u + \Omega(v) \to \Omega(u+v). \tag{25}$$

However, these are of dubious desirability unless $u+v$ simplifies, such as when $u = \alpha v$, with α free of all asymptotic variables, for which we have

$$\alpha v + \theta(v) \to \theta(v), \tag{26}$$

$$\alpha v + O(v) \to O(v), \tag{27}$$

$$\alpha v + \Omega(v) \to \Omega(v), \tag{28}$$

and also

$$\alpha v + o(v) \to \text{ASYMP}(\alpha v), \tag{29}$$

$$\alpha v + \omega(v) \to \omega(v). \tag{30}$$

2.4. Asymptotic series. One way of developing an n-term asymptotic expansion for an expression u_0 is to let $u_{-1} = 0$; then for $j = 1, 2, \cdots, n$, use ASYMP to successively compute simplified terms u_j such that $u_{j-2} - u_{j-1} \sim u_j$. The desired asymptotic expansion is then

$$\sum_{j=1}^{n} u_j.$$

This expansion is the most natural one in the sense that successive basis functions are selected automatically according to the nature of u_0, rather than artificially imposed by the user. Accordingly, a function which performs such expansions is included in the package.

3. Demonstration. MACSYMA is an interactive computer system which prompts the user for successive commands with a unique label beginning with the letter C. The user then types an expression terminated by a semicolon, after which MACSYMA generates and prints a corresponding simplified expression having a correspondingly numbered label beginning with the letter D. D-label results can be referred to in subsequent expressions.

At the MIT MACSYMA-consortium Decsystem 10(KL model), the asymptotic simplification package resides on a publicly accessible disk file. To load that file:

(C1) BATCH(ASYMP, ">", DSK, SHARE);

(D47) BATCH DONE.

To establish that N asymptotically approaches ∞ and that X asymptotically approaches 0:

(C48) PUT(*N*, *INF*, *LIMIT*);

(D48) *INF*

(C49) PUT(*X*, 0, *LIMIT*);

(D49) 0.

Now, consider the expression

(C50) (3*SQRT(*X*)*LOG(*N*)+6*(*X***N*)**2+*X***2**N*)/(*N*!*3***X*+*X***(*N*+*X*)+5);

$$\text{(D50)}\qquad \frac{6\ N^2\ X^2+N\ X^2+3\ \text{LOG}(N)\ \text{SQRT}(X)}{N!\ 3^X+X^{X+N}+5}.$$

To asymptotically simplify the expression:

(C51) ASYMP(D50);

$$\text{(D51)}\qquad \text{ASYMP}\left(\frac{6\ N^2\ X^2+3\ \text{LOG}(N)\ \text{SQRT}(X)}{N!\ 3^X}\right).$$

To find a simpler expression of the exact same order:

(C52) THETA(D50);

$$\text{(D52)}\qquad \text{THETA}\left(\frac{2\ N^2\ X^2+\text{LOG}(N)\ \text{SQRT}(X)}{N!\ 3^X}\right).$$

To find still a simpler expression of at least the same order:

(C53) O(D50);

$$\text{(D53)}\qquad \text{O}\left(\frac{N^2\ \text{SQRT}(X)}{N!\ 3^X}\right).$$

To find a simple lower bound for the order

(C54) OMEGA(D50);

$$\text{(D54)}\qquad \text{OMEGA}\left(\frac{\text{LOG}(N)\ X^2}{N!\ 3^X}\right).$$

Now, suppose that we wish to combine two of the above approximate expressions as follows:

(C55) D53**2*D51;

$$\text{(D55)}\qquad \text{O}^2\left(\frac{N^2\ \text{SQRT}(X)}{N!\ 3^X}\right)\ \text{ASYMP}\left(\frac{6\ N^2\ X^2+3\ \text{LOG}(N)\ \text{SQRT}(X)}{N!\ 3^X}\right).$$

To consolidate the approximate subexpressions:

(C56) ASYMPSIMP(D55);

(D56) $$O\left(\frac{N^6\ X^{3/2}}{N!^3\ 3^{3X}}\right).$$

Finally, consider the expression

(C57) (N!+3**N)**2/LOG(N)+(N*LOG(N)+2**N)**3;

(D57) $$\frac{(N!+3^N)^2}{\mathrm{LOG}(N)}+(N\ \mathrm{LOG}(N)+2^N)^3.$$

To get a natural 2-term asymptotic expansion of this expression:

(C58) ASYMPSERIES(D57, 2);

(D58) $$\frac{N!^2}{\mathrm{LOG}(N)}+\frac{2\ 3^N\ N!}{\mathrm{LOG}(N)}.$$

4. Test results. The intent was to implement a package which can asymptotically simplify a broad spectrum of complicated expressions in a reasonable amount of computer time. Probably the most relevant test of the performance relative to this goal is to collect examples and reactions from users over the next year or two. Until then, artificially-contrived tests must suffice as a brief indication of the performance.

For application of the functions THETA, *O*, *LO*, LOMEGA to expressions having many terms, most of the work is usually done by an internal use of the function ASYMP. Consequently, test results are summarized only for ASYMP. Accordingly, the following random expression generator was designed to produce a class of expressions which is both plausible and suitable as inputs to these asymptotic simplifiers:

$$\text{expression}\ (p, q, r) ::= \sum_{j=1}^{p} \text{term}_j(q, r)$$

$$\text{term}\ (q, r) ::= \prod_{k=1}^{q} \text{factor}_k(r)$$

factor (r) : : = variable(r) | nonvariable(r)

variable (r) : : = variable$_1$ | variable$_2$ | $\cdots$ | variable$_r$

nonvariable (r) : : = number | log(factor(r)) | factor(r)↑exponent(r)

number : : = 2 | 3 | $\cdots$ | 9

exponent (r) : : = number | factor(r)

variable$_1$ ::= N

variable$_2$ ::= M

$\cdots$,

where random elements of each set of alternatives are chosen with equal probability, and where variables $M, N, \cdots$ asymptotically approach ∞. For example, a typical

instance of expression (16, 4, 2) is

$$504\ \mathrm{LOG}(\mathrm{LOG}(5))\ N^{25}\ \mathrm{LOG}(N)+M\ N^{2}\ \mathrm{LOG}(N)+M\ \mathrm{LOG}^{5}(M)\ N\ \mathrm{LOG}(N)$$
$$+4\ \mathrm{LOG}(5)\ N\ \mathrm{LOG}(N)+147\ \mathrm{LOG}(N)+\mathrm{LOG}(M)\ N^{2N+2}+6\ M^{2}\ N^{36}$$
$$+42\ \mathrm{LOG}(M)\ N^{5}+M\ N^{3}+M\ \mathrm{LOG}(M)^{\mathrm{LOG}(\mathrm{LOG}(6))}\ N^{2}+2\ M\ N^{2}$$
$$+M^{M}\ 3^{M^{4}}\ \mathrm{ABS}(M)^{4MM}\ N+M^{2}\ 7^{M}\ N$$
$$+9\ \mathrm{LOG}(6)\ M\ N+6\ M^{2}\ \mathrm{LOG}(M)+M^{4}.$$

Due to collection of similar terms and other routine simplifications, this expression has an average of 3 rather than 4 factors per term. Using the program's complexity measure of the number of atoms in the internal representation, this expression has a complexity of 125. Using an untranslated, uncompiled version of the program, ASYMP required 19 seconds to simplify this expression to

$$\mathrm{ASYMP}(\mathrm{LOG}(M)\ N^{2N+2}+6M^{2}N^{36}+M^{4MM+M}\ 3^{M^{4}}\ N),$$

for which the complexity of the argument is 35.

This example is from a systematic sequence of tests log-log plotted in Fig. 1, from data summarized in Table 1.

This sequence of tests was conducted using the built-in random number generator with its default initialization. Most complexity analyses entail only one asymptotic

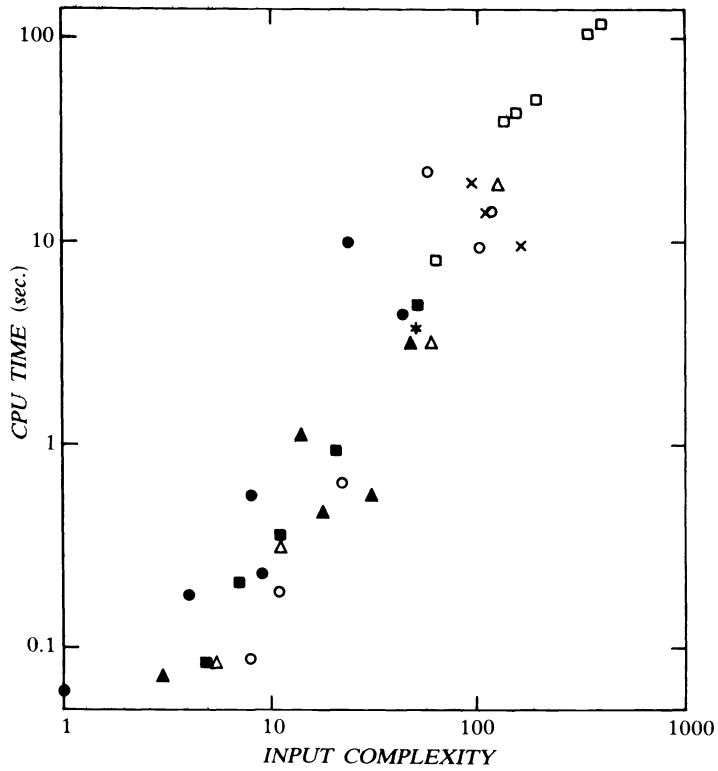


FIGURE 1

TABLE 1

Case	p	q	r	Input terms	Output terms	Input complexity	Output complexity	Time (sec)	Plot symbol
1	1	1	1	1	1	1	1	0.06	●
2	2	1	1	2	1	4	1	0.18	●
3	4	1	1	2	1	8	3	0.58	●
4	8	1	1	3	1	9	3	0.24	●
5	16	1	1	6	1	24	7	10.0	●
6	32	1	1	12	1	44	6	4.5	●
7	64	1	1	23	23	95	95	19.5	×
8	1	2	1	1	1	3	3	0.07	▲
9	2	2	1	2	1	14	9	1.10	▲
10	4	2	1	4	1	18	5	0.48	▲
11	8	2	1	6	1	31	9	0.57	▲
12	16	2	1	10	1	47	5	3.15	▲
13	32	2	1	19	—	106	—	>128	
14	1	2	2	1	1	5	5	0.08	■
15	2	2	2	2	2	7	7	0.22	■
16	4	2	2	3	2	11	8	0.36	■
17	8	2	2	4	3	21	19	0.96	■
18	16	2	2	12	2	51	12	4.8	■
19	32	2	2	21	21	111	111	14.4	×
20	1	3	2	1	1	8	8	0.08	○
21	2	3	2	2	1	11	7	0.19	○
22	4	3	2	4	3	22	19	0.66	○
23	8	3	2	8	2	58	16	22.0	○
24	16	3	2	16	4	102	35	9.3	○
25	32	3	2	29	—	210	—	>128	
26	1	4	2	1	1	5	5	0.08	▽
27	2	4	2	2	2	11	11	0.31	▽
28	4	4	2	4	—	49	—	36.8	*
29	8	4	2	8	4	60	35	3.1	▽
30	16	4	2	16	3	125	35	19.3	▽
31	32	4	2	29	—	243	—	>128	

Case	Sum of cases	Input terms	Output terms	Input complexity	Output complexity	Time (sec)	Plot symbol
32	1 thru 6	16	1	63	6	8.1	□
33	8 thru 12, 32	27	1	135	9	40.1	□
34	14 thru 16, 33	31	3	150	18	43.6	□
35	17, 34	33	33	164	164	9.6	×
36	18, 34	40	3	193	21	51.8	□
37	20 thru 24, 36	60	4	347	36	111	□
38	26, 27, 37	62	4	364	36	110	□
39	29, 38	66	5	398	43	123	□
40	30, 39	80	—	517	—	>128	

variable, and few analyses entail more than 2, so the testing was confined to these values.

Cases 7, 19 and 35 are examples where the limit function returned UND or an inconclusive result, or where cancellation occurred, thus forcing ASYMP to retain all terms.

Cases 13, 25, 31, and 40 are examples where computation was manually interrupted after exceeding an arbitrary limit of 128 seconds.

Despite its relatively modest total complexity, Case 28 was automatically terminated after 37 seconds because of insufficient space for large integers. The example was

$$M^{3\ \mathrm{LOG}^2(\mathrm{LOG}(\mathrm{LOG}(\mathrm{LOG}(M))))}N^{\mathrm{LOG}(6\ \mathrm{LOG}(N))+2}+4096M\ \mathrm{LOG}(M)\ N^{N^N+N^9}$$
$$+M^2\ N^7+8\ M\ N^2,$$

so it is not hard to understand why LIMIT had such a hard time. This example illustrates the fact that the performance is very sensitive to the difficulty of the most difficult factors. The performance is also sensitive to how soon difficult factors occur during testing, especially if they are relatively dominant. This sensitivity is also illustrated by the appreciable scattering in Fig. 1 despite the fact that it is a 3-decade log-log plot. Although the data suggests a slope of about 2, hence a roughly quadratic growth of computing time with input complexity for this class of expressions, it would be rash to fit any specific curve to data having this much variability.

Cases 32 through 40 were constructed by adding together the inputs for previous successful cases, in order to gather data for longer expressions than is likely to be successfully simplified for this random expression generator. For example the input and output for case 39 are respectively

$$N^2\ \mathrm{LOG}^7(\mathrm{LOG}(4)\ \mathrm{LOG}(\mathrm{LOG}(N)))+5\ M^4\ N\ \mathrm{LOG}(9\ \mathrm{LOG}(N))$$
$$+\mathrm{LOG}(\mathrm{LOG}(N))^{\mathrm{LOG}(7)}+\mathrm{LOG}(6)\ M^{24}\ N\ \mathrm{LOG}(\mathrm{LOG}(N))$$
$$+11\ \mathrm{LOG}(\mathrm{LOG}(N))+\mathrm{LOG}(N)^{N^2}+\mathrm{LOG}(6561)\ \mathrm{LOG}^N(N)$$
$$+\mathrm{LOG}(9)\ N\ \mathrm{LOG}^9(N)+\mathrm{LOG}^7(N)+\mathrm{LOG}(M)\ N^6\ \mathrm{LOG}^2(N)+N^3\ \mathrm{LOG}^2(N)$$
$$+25\ M\ \mathrm{LOG}^2(N)+\mathrm{LOG}^2(N)+3\ \mathrm{LOG}(4)\ N^{28}\ \mathrm{LOG}(N)+N^8\ \mathrm{LOG}(N)$$
$$+7\ N^5\ \mathrm{LOG}(N)+8\ N^2\ \mathrm{LOG}(N)+4096^M\ N\ \mathrm{LOG}(N)$$
$$+M^2\ N\ \mathrm{LOG}(N)+\mathrm{LOG}(7)\ N\ \mathrm{LOG}(N)+67\ N\ \mathrm{LOG}(N)$$
$$+\mathrm{LOG}(M)\ \mathrm{LOG}(N)+M^2\ \mathrm{LOG}(N)+5\ M\ \mathrm{LOG}(N)$$
$$+\mathrm{LOG}(\mathrm{LOG}(3))\ \mathrm{LOG}(N)+\mathrm{LOG}(3)\ \mathrm{LOG}(N)$$
$$+12\ \mathrm{LOG}(N)+N\ 6^N+N^{5N^2+1}+6\ N^N+N^{15}+N^9+M\ N^8+M\ N^7+4\ N^7$$
$$+56\ N^6+N^3+M^2\ N^2+\mathrm{LOG}(7)\ M\ N^2+4M\ N^2+7N^2$$
$$+M\ \mathrm{LOG}^2(M)\ \mathrm{LOG}(\mathrm{LOG}(\mathrm{LOG}(M)))\ N+M^5\ \mathrm{LOG}(M)\ N$$
$$+5\ \mathrm{LOG}(M)\ N+M^9\ N+4\ M^7\ N+M^2\ N+15\ M\ N+\mathrm{LOG}(8)\ N$$
$$+2\ \mathrm{LOG}(7)\ N+3\ \mathrm{LOG}^3(2)\ N+884518\ N+M^{2M^6+1}\mathrm{LOG}(\mathrm{LOG}(M))$$
$$+\mathrm{LOG}^2(M)+10\ M^{10}\ \mathrm{LOG}(M)+134217728\ M^M+9\ M^6+7\ \mathrm{LOG}(5)\ M^5$$
$$+7\ M^2+3^{\mathrm{LOG}(9)}\ M+\mathrm{LOG}^N(3)\ M+2\ \mathrm{LOG}^9(\mathrm{LOG}(4))\ M+48\ M$$
$$+10\ \mathrm{LOG}(8)+6\ \mathrm{LOG}(2)+16942$$

and

$$\mathrm{ASYMP}(4096^M\ N\ \mathrm{LOG}(N)+N^{5N^2+1}+M^2\ N^2+M^{2M^6+1}\ \mathrm{LOG}(\mathrm{LOG}(M))$$
$$+\mathrm{LOG}^N(3)M)$$

Case 34 is the only instance where the LIMIT function returned UND or an inconclusive result, or where cancellation occured, for a case constructed by adding together previously successful cases.

Expressions generated by this generator are rarely suitable for the TAYLOR function, so all testing was done with the default setting of MAXTAYLOR = 0.

5. Conclusions and suggestions for further research. The demonstration in § 3 and the test results in § 4 indicate that order algebra is a feasible and worthwhile supplementary package for a computer-algebra system. However, there are open questions about what type of simplifications most users would like in such a package and about how best to implement the package. Hopefully, experience gained from varied use of the prototype described here will lead to the development of a more efficient, flexible, and comprehensive package. Especially promising improvements are:

1. extension of the TAYLOR function to form expansions in terms of a set of arbitrary basis functions specified by the user or automatically deduced from the given expression;
2. provision of optional and more drastic simplification for the operators o and ω;
3. provision for asymptotic approximation of sums and integrals in terms of asymptotic approximations of their limits and their summands or integrands.

Acknowledgment. I thank D. Askey, R. W. Gosper, N. Pippenger, and R. Zippel for their helpful suggestions.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1975.

[2] J. COHEN AND J. KATCOFF, *Symbolic solution of finite-difference equations*, ACM Trans. Math. Software 3 (1977), pp. 261–271.

[3] R. W. GOSPER, JR., *Indefinite hyperbolic Sums in MACSYMA*, Proceedings of the 1977 MACSYMA Users' Conference, NASA CP-2012, pp. 237–252.

[4] ———, *Decision procedure for indefinite hypergeometric summation*, Proc. Natl. Acad. Sci. U.S.A., 75 (1978), pp. 40–42.

[5] J. H. HALTON, *Asymptotics for formula manipulation*, Proceedings of the 1968 Summer Institute in Symbolic Mathematical Computation, IBM, June 1969, pp. 149–194.

[6] G. H. HARDY, *Orders of Infinity*, Cambridge University Press, London, 1910.

[7] J. IVIE, *Some MACSYMA programs for solving difference equations*, ACM Trans. Math Software, 4 (1978), pp. 24–33.

[8] D. E. KNUTH, *Big Omicron and Big Omega, and Big Theta*, ACM SIGACT News, April-June, 1976, pp. 18–24.

[9] MATHLAB GROUP, *MACSYMA Reference Manual*, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1977.

[10] R. MOENCK, *On computing closed forms for summations*, Proceedings of the 1977 MACSYMA Users' Conference, NASA CP-2012, pp. 237–252.

[11] P. WANG, *Evaluation of Definite Integral by Symbolic Manipulation*, Ph.D. dissertation and Project MAC Technical Report TR-92, M.I.T., Cambridge, MA, October, 1971.

[12] R. ZIPPEL, *Univariate power series expansions in MACSYMA*, Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, R. D. Jenks, ed., pp. 198–208.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0002 $01.00/0

NEW ALGORITHMS FOR POLYNOMIAL SQUARE-FREE DECOMPOSITION OVER THE INTEGERS*

PAUL S. WANG† AND BARRY M. TRAGER‡

Abstract. Previously-known algorithms for polynomial square-free decomposition rely on greatest common divisor (gcd) computations over the same coefficient domain where the decomposition is to be performed. In particular, gcd of the given polynomial and its first derivative (with respect to some variable) is obtained to begin with. Application of modular homomorphism and p-adic construction (multivariate case) or the Chinese remainder algorithm (univariate case) results in new square-free decomposition algorithms which, generally speaking, take less time than a single gcd between the given polynomial and its first derivative. The key idea is to obtain one or several "correct" homomorphic images of the desired square-free decomposition first. This provides information as to how many different square-free factors there are, their multiplicities and their homomorphic images. Since the multiplicities are known, only the square-free factors need be constructed. Thus, these new algorithms are relatively insensitive to the multiplicities of the square-free factors.

Key words. square-free decomposition, lucky evaluation, polynomial factoring, Chinese remainder algorithm, p-adic construction

1. Introduction. The polynomial square-free decomposition process has many uses in symbolic algebraic computation. Its applications include polynomial factorization, partial fraction decomposition, and integration of rational functions. A recent paper by David Yun [7] describes several algorithms for square-free (SQFR) decomposition. Each of these methods depends on computing the greatest common divisor (GCD) of the polynomial to be decomposed and its first derivative (with respect to some variable). For these methods, this GCD computation dominates the cost of the SQFR decomposition. When computing over the integers, this GCD computation can be costly, especially if the polynomial is multivariate.

Several new algorithms for SQFR decomposition over the integers are presented. These algorithms avoid computing the above mentioned GCD over the original coefficient domain and are generally less costly than that single GCD computation when applied to non-SQFR polynomials. The key idea is to obtain one or several "correct" homomorphic images of the desired SQFR decomposition first. This provides information as to how many different SQFR parts (or factors) there are, their multiplicities, and their homomorphic images. If the homomorphic image is square-free, then the given polynomial is square-free. Thus SQFR polynomials are detected very early in these algorithms. Otherwise, the homomorphic images of the SQFR factors are used to construct these factors which leads to the complete SQFR decomposition. Since only the SQFR factors need to be reconstructed, these new algorithms are relatively insensitive to the multiplicities of the factors.

Two different SQFR algorithms are described. The modular SQFR algorithm described in § 3 uses the Chinese remainder algorithm for reconstruction and is

* Received by the editors May 23, 1978. This work was supported in part by the Laboratory for Computer Science (formerly Project MAC), an M.I.T. Interdepartmental Laboratory, sponsored by the United States Department of Energy under contract E(11-1)-3070, and by the National Aeronautics and Space Administration under Grant NSG 1323. This work was also supported in part by Kent State University and some of the materials incorporated in this work were developed with the financial support of National Science Foundation Grant NCS78-02234.

† Department of Mathematics, Kent State University, Kent, Ohio 44242.

‡ Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

especially suited to univariate polynomials. The p-adic square-free algorithm described in § 2 uses a Hensel type p-adic procedure for construction and is more efficient for multivariate polynomials.

Our algorithms depend on the use of correct or "lucky" homomorphic mappings. In § 4 we carefully discuss the concept of "luckiness" and related issues. Finally in § 5 we outline generalizations of our approach and its implications for symbolic manipulation systems [3]. Actual machine timings are included in the Appendix.

2. The p-adic SQFR algorithm. Let $F(x, x_2, \cdots, x_t) \in \mathbf{Z}[x, x_2, \cdots, x_t]$ be a polynomial primitive with respect to x. A SQFR decomposition of $F(x, x_2, \cdots, x_t)$ consists of finding polynomials $F_1(x, x_2, \cdots, x_t), \cdots, F_n(x, x_2, \cdots, x_t)$ over $\mathbf{Z}$ such that

$$
\begin{aligned}
&F = F_1^{e_1} F_2^{e_2} \cdots F_n^{e_n} \\
&0 < e_1 < e_2 \cdots < e_n \\
&\gcd(F_i, F_j) = 1 \quad \text{for } i \neq j
\end{aligned}
\tag{2.1}
$$

and each F_i has no repeated factors (is SQFR).

Let $\{a_2, \cdots, a_t\}$ be a set of integers (not necessarily distinct) and $F_0(x) = F(x, a_2, \cdots, a_t)$. If $\deg(F_0) = \deg(F(x, x_2, \cdots, x_t))$ in x then $\{a_2, \cdots, a_t\}$ is called a *valid evaluation* (or simply evaluation). Let $F_{i_0}(x) = F_i(x, a_2, \cdots, a_t)$ for $i = 1, \cdots, n$, $d = \deg(\gcd(F_0, F_0'))$ and $\delta = \deg(\gcd(F, \partial F/\partial x))$ in x, with $d \geqq \delta$ always true.

LEMMA 2.1. *If $F_0(x)$ is obtained by a valid evaluation of F, then*

$$
F_0(x) = F_{1_0}^{e_1} \cdots F_{n_0}^{e_n} \tag{2.2}
$$

is the square-free decomposition of $F_0(x)$ if and only if $d = \delta$.

Proof. The "only if" part is obvious. If (2.2) is not the square-free decomposition of $F_0(x)$ then at least one of the following is true: (a) $F_{i_0}(x)$ is not SQFR for some i, (b) $\gcd(F_{i_0}, F_{j_0}) \neq 1$ for some $i \neq j$. It can be deduced that (a) and/or (b) imply that $d > \delta$. □

In this algorithm a few different evaluations will be generated. From these different evaluations, a set, say $\{a_2, \cdots, a_t\}$, that gives a minimum value for d will be used. If d is zero for any evaluation, then F_0 is SQFR and so is F. Suppose $d \neq 0$, then F_0 is not SQFR. In this case, one applies the univariate SQFR algorithm of § 3 to $F_0(x)$ obtaining the square-free decomposition of $F_0(x)$ over $\mathbf{Z}$:

$$
F_0(x) = f_1^{r_1} f_2^{r_2} \cdots f_m^{r_m}, \qquad 0 < r_1 < r_2 \cdots < r_m. \tag{2.3}
$$

Note that $r_m \geqq e_n$ and if $r_m = e_n$ then $\deg(f_m) \geqq \deg(F_n)$ in x.

If the evaluation $\{a_2, \cdots, a_t\}$ used is selected from randomly generated evaluations, then $d = \delta$ with high probability (see § 4 for details). If $d = \delta$ then by Lemma 2.1 we have $m = n$, $r_i = e_i$ and $f_i = F_{i_0}$ for all i. In other words (2.3) is exactly the same as (2.2). The algorithm proceeds to construct F_n from f_m on the assumption that $r_m = e_n$ and $f_m = F_{n_0}$ which is a weaker condition than $d = \delta$. The evaluation $\{a_2, \cdots, a_t\}$ is "unlucky" if this assumption is incorrect. In the unlucky case a division test later in the algorithm will fail, causing the selection of a different evaluation that lowers the value of d.

If $m = 1$ and $r_m = k \neq 1$ then the algorithm proceeds by taking the kth root of $F(x, x_2, \cdots, x_t)$. If the kth root is not exact then $\{a_2, \cdots, a_t\}$ is unlucky. For $m > 1$, the

algorithm computes

$$D(x, \cdots, x_t) = \frac{1}{(r_m - 1)!}\left(\frac{\partial}{\partial x}\right)^{r_m - 1} F(x, \cdots, x_t),$$

$$D_0(x) = \frac{1}{(r_m - 1)!}\left(\frac{\partial}{\partial x}\right)^{r_m - 1} F_0(x).$$

THEOREM 2.1. *If G is the "most repeated part" of the square-free decomposition of F and e is its multiplicity, i.e., $F = HG^e$, G is SQFR, $\gcd(H, G) = 1$ and H has no factors with multiplicity greater than $e-1$, then $G \mid D = (\partial/\partial x)^{e-1} F$ and G and D/G are relatively prime.*

Proof. Consider

$$\left(\frac{\partial}{\partial x}\right)^{e-1} F = G^e\left(\frac{\partial}{\partial x}\right)^{e-1} H + (e-1)\frac{\partial G^e}{\partial x}\left(\frac{\partial}{\partial x}\right)^{e-2} H + \cdots$$
$$+ \binom{e-1}{i}\left(\frac{\partial}{\partial x}\right)^{i} G^e\left(\frac{\partial}{\partial x}\right)^{e-i-1} H + \cdots + H\left(\frac{\partial}{\partial x}\right)^{e-1} G^e.$$

Each term on the right hand side of the above equation is divisible by G^2 except the last term which is

$$H\left(\frac{\partial}{\partial x}\right)^{e-1} G^e = e!HG\left(\frac{\partial G}{\partial x}\right)^{e-1} + HG^2P$$

for some polynomial P. Therefore $G \mid D$ and $\gcd(G, D/G) = 1$ since $\gcd(G, \partial G/\partial x) = 1$. □.

It follows from Theorem 2.1 that $f_m(x) \mid D_0(x)$, $\gcd(f_m, D_0/f_m) = 1$ and

$$D(x, x_2, \cdots, x_t) \equiv f_m(x)(D_0(x)/f_m(x)) \mod \mathbf{S} \tag{2.4}$$

where $\mathbf{S}$ is the ideal $(x_2 - a_2, \cdots, x_t - a_t)$. Thus the congruence (2.4) can be lifted using a multivariate p-adic construction. One can use the well-known EZ algorithm [5] for this purpose. If one wishes to take advantage of the new EEZ p-adic construction devised recently which features correct leading coefficient distribution and variable-by-variable construction, more conditions have to be imposed on the evaluation integers a_i. Interested readers are referred to [6] for details of the EEZ algorithm. Let $f_m^{(1)}(x) = f_m(x)$ and $g_m^{(1)}(x) = D_0(x)/f_m(x)$. A sequence of polynomials $f_m^{(i)}(x, \cdots, x_t)$ and $g_m^{(i)}(x, \cdots, x_t)$, $i = 1, 2, \cdots$ will be constructed such that $f_m^{(i+1)} \equiv f_m^{(i)} \mod \mathbf{S}^i$, $g_m^{(i+1)} \equiv g_m^{(i)} \mod \mathbf{S}^i$ and $D(x, \cdots, x_t) \equiv f_m^{(i)} g_m^{(i)} \mod \mathbf{S}^i$. The construction terminates whenever $f_m^{(i)}$ stops changing and becomes a divisor of D over $\mathbf{Z}$ or when i exceeds h, the total degree of $D(x, \cdots, x_t)$ in $x_2, \cdots, x_t$. In any case we have $\hat{f}_m = f_m^{(h+1)}$.

LEMMA 2.2. *F is divisible by $\hat{f}_m^{r_m}$ if and only if $r_m = e_n$ and $\hat{f}_m = \pm F_n$.*

Proof. Obviously, $F_n^{e_n} \mid F$. Since $r_m \geqq e_n$, $f_m^{r_m} \mid F$ implies that $r_m = e_n$ and $\hat{f}_m \mid F_n$. Since F is primitive and $\deg(\hat{f}_m) \geqq \deg(F_n)$ in x, we have $\hat{f}_m = \pm F_n$. □

Now divide $\hat{f}_m^{r_m}$ into F. If the division is exact then, by Lemma 2.2, $F_n \leftarrow f_m$ and $e_n \leftarrow r_m$. Now set $F \leftarrow F/F_n^{e_n}$, $F_0 \leftarrow F_0/f_m^{r_m}$, $d \leftarrow d - (e_n - 1)\deg(f_m)$ and iterate the above process to find F_{n-1} and e_{n-1} etc. until the SQFR decomposition is complete.

If the division does not succeed then the evaluation $\{a_2, \cdots, a_t\}$ is "unlucky". In this case the algorithm goes back for a different evaluation that lowers the value of d.

3. Modular SQFR algorithm. Here we will consider the SQFR decomposition of a primitive polynomial $F(x) \in \mathbf{Z}[x]$. We will use as moduli a sequence of large primes, usually chosen to be just smaller than the word size on the machine used. The algorithm generalizes immediately to multivariate polynomials if we replace our moduli by linear forms $x_i - a_i$, where the a_i are elements of some finite field. This generalization is inefficient by comparison with the algorithm of § 2 since we believe that large multivariate computations are almost invariably sparse. Modular algorithms do not take advantage of the sparseness of their inputs, unlike algorithms based on Hensel's lemma. On the other hand modular algorithms are much less sensitive to unlucky primes or unlucky evaluations. A Hensel algorithm performs its lifting with only one homomorphic image and its unluckiness may only be determined after lifting past some predetermined bound. Modular algorithms construct the solution from many different homomorphic images and the occurrence of the first lucky image will cause any previous unlucky images to be discarded. In the univariate case sparsity is less of an issue and modular algorithms tend to perform quite well.

By induction suppose that we have computed the SQFR decomposition $F(x) \equiv \hat{f}_1^{e_1} \hat{f}_2^{e_2} \cdots \hat{f}_n^{e_n} \bmod q$, where q is a product of the different primes used. The algorithm proceeds by reducing $F(x)$ modulo the next prime modulus p which does not divide the leading coefficient of $F(x)$ *yielding* $f(x) \in \mathbf{Z}_p[x]$. The SQFR decomposition $f(x) = f_1^{r_1} \cdots f_m^{r_m}$ is then computed in the image domain, $\mathbf{Z}_p$. As long as the modulus is greater than the degree of $F(x)$, one can show that any of the SQFR algorithms presented by Yun [7] will work properly. However, his algorithm (c) is computationally somewhat better than the others and is recommended. Now we check whether the current SQFR decomposition is compatible with the ones obtained before. Being compatible means that each has the same number of factors ($n = m$), corresponding degrees, and multiplicities ($r_i = e_i$). As in Lemma 2.1 compatibility is completely determined by $d = \deg(\gcd(f(x), f'(x))) = \deg(f_1^{r_1 - 1} \cdots f_m^{r_m - 1})$ which is easily computed from the SQFR decomposition of $f(x)$. Potentially "lucky" primes are those associated with the minimal d so far. Any prime giving rise to a larger value of d is discarded, and the discovery of a smaller value of d than any seen so far will cause the currently accumulated results to be discarded. We now apply the Chinese remainder algorithm [1] in parallel to the ordered lists $(f_1, \cdots, f_m)$ and $(\hat{f}_1, \cdots, \hat{f}_m)$ yielding a new list $(\hat{f}_1, \cdots, \hat{f}_m)$ such that $F(x) \equiv \hat{f}_1^{e_1} \cdots \hat{f}_m^{e_m}$ modulo pq. We will show that there are only a finite number of unlucky primes. The appearance of the first lucky prime guarantees that we will only perform Chinese remainder interpolations with lucky primes from then on. The occurrence of unlucky primes is actually an extremely rare event, so we will almost never have any wasted computation.

As in most modular algorithms there is a problem with nonuniqueness during interpolation. We have to impose a leading coefficient in the $\hat{f}_i$ which is guaranteed to be a multiple of the correct leading coefficient. We do this by imposing the leading coefficient (lcf) of F on each $\hat{f}_i$. Then we check $\hat{f}_m$ after every Chinese remainder stage to see whether or not it changed. If it is unchanged then it is very likely that the primitive part of $\hat{f}_m$ is precisely F_m. We verify this by attempting to divide *principal part* $(\hat{f}_m)^{e_m}$ into F. If the division succeeds then we set F to the quotient and decrement m by 1. Then we investigate whether or not the new $\hat{f}_m$ was unchanged by the last interpolation. If the division fails or $\hat{f}_m$ did change, then we simply continue accumulating results with more primes.

4. Unlucky primes and substitution values. In this section moduli will refer to both integer primes and linear forms $x_i - a_i$. A prime is lucky if and only if the modular

homomorphism Φ commutes with SQFR decomposition, i.e. $\Phi(SQFR(F)) = SQFR(\Phi(F))$. It is easily seen that this is equivalent to F having compatible SQFR decompositions in the domain and image spaces. The commutative diagram formulation generalizes to any operation for which one might want to use a Hensel or modular algorithm. One should note that there are some algorithms like factoring univariate polynomials for which Hensel algorithms are applicable even though every prime may be unlucky. An example is $x^4 + 1$ which factors modulo every prime even though it is irreducible over the integers. Unlucky primes can yield useful information which may enable one to solve problems, albeit usually requiring more computation than lucky primes would.

When computing SQFR decompositions of polynomials, however, as with GCD computations, unlucky moduli do not provide much useful information and we need to be sure that "almost all" moduli are lucky. Assuming a SQFR decomposition of F as in (2.1), an unlucky prime implies that either some F_{i_0} is not SQFR, or gcd $(F_{i_0}, F_{j_0}) \neq 1$ for some $i \neq j$. If we let $G = F_1 F_2 \cdots F_n$ and $G_0 = \Phi(G)$, then G is SQFR and an unlucky modulus implies that G_0 is not SQFR. G is SQFR implies that gcd $(G, G) = 1$ which forces *discriminant* $(G) \neq 0$ (*discriminant* $(G) =$ *resultant* $(G, G')/lcf(G)$). An unlucky modulus implies *discriminant* $(G_0) = 0$. By Collins' theorem 4 [2] *discriminant* $(G_0) = 0$ implies $\Phi($*discriminant* $(G)) = 0$. Thus the polynomial *discriminant* $(G) \in \mathbf{Z}[x_2, \cdots, x_t]$ is nonzero and vanishes for all unlucky homomorphisms. For univariate F this shows that there are only a finite number of unlucky primes. For multivariate F the space of unlucky evaluations satisfies this polynomial relation and thus lies on an $n-2$ dimensional hypersurface of the $n-1$ dimensional space of evaluations. This implies that almost all points are lucky. (See also [1]).

5. Conclusions. The central idea behind these new SQFR algorithms is to take advantage of a compact representation of the answer in order to minimize the number of modular images or Hensel lifting stages required to reconstruct the answer. Previous SQFR algorithms started by computing gcd (F, F') as a multivariate polynomial over the integers. Since we are really only interested in the SQFR factor F_i and their multiplicities e_i, there is no need to actually compute the gcd in the original domain. Instead we perform complete SQFR decompositions in the image domain, and use the information generated to reconstruct only the F_i's in our original domain.

Modular and Hensel algorithms were first introduced to symbolic computation systems as a technique for combating intermediate expression swell. The greatest advantage can be obtained from these algorithms by maximizing the time spent in the image domains and minimizing the time spent reconstructing solutions in the original domain. Currently, independent algorithms have been published which take advantage of modular techniques, but complex computations tend to proceed by applying these algorithms sequentially to multivariate polynomials over the integers. More emphasis should be placed on performing entire computations in the image domain, and only reconstructing solutions over the integers for the final results. The central problems include insuring uniqueness in the image domain and finding tests which verify "good reduction". A general solution along these lines would be a big step toward eliminating intermediate expression swell during symbolic computations.

Appendix. The timing examples were done using the MACSYMA system running on a DEC KL10 [3]. NSQFR refers to the hybrid algorithm described in §§ 2 and 3 of this paper. OSQFR is Yun's algorithm (c) in conjunction with the EZGCD algorithm [4] directly applied to multivariate polynomials over **Z**. All times are in seconds.

Polynomial	NSQFR	OSQFR
$(3x^2+x+1)y^2+2xy+x^2+x$	.04	.03
$(y^4+x^3)(y^3+x^2)^2(y^2+x)^3$	1.08	2.04
$(z+y+x+1)(z-y+x+4)^2(z-2y+x+7)^3$	.35	1.3
$(y+x^2+5)(6y+2x^3+31)^3(xy^3+8y-x)^5$	5.05	46.47
$(z^2+xyz+x^2)^2(3z^2+(y^2+x)z-4)^3$ $(z^3+z^2+(x-1)y)^4$	30.36	70.1
$(3y^4+x+5)(6y^2+2x+31)^2$ $((x^2+x+1)y^3-y+x+8)^4(xy+8y+x)^{10}$	15.73	321.23
$(x^4351x^3+27x^2-31x+1)^2$ $(6x^5+251x^3-372x^2+15x-323)^6$	1.22	1.69
$(x^4+351x^3+27x^2-31x+1)^3$ $(6x^5+251x^3-372x^2+15x-323)^9$	2.3	4.53
$(x^3+75x^2+68x+1)^3(x^2+15x+35)^6$ $(7x^2+750x+137)^9(x+75)^{12}$	3.06	5.53
$(x^3+75x^2+68x+1)^5(x^2+15x+35)^{10}$ $(7x^2+750x+137)^{15}(x+75)^{20}$	7.74	18.99

REFERENCES

[1] W. S. BROWN, *On Euclid's algorithm and the computation of polynomial greatest common divisors*, J. Assoc. Comput. Mach., 18 (1971), pp. 478–504.

[2] G. E. COLLINS, *The calculation of multivariate polynomial resultants*, Ibid., 18 (1971), pp. 515–532.

[3] MATHLAB GROUP, *MACSYMA Reference Manual*, Version 10, Laboratory for Computer Science, M.I.T., Cambridge, MA, 1979.

[4] J. MOSES AND D. Y. Y. YUN, *The EZGCD algorithm*, Proc. of ACM Annual Conference, Aug. 1973 (Atlanta), pp. 159–166.

[5] P. S. WANG AND L. P. ROTHSCHILD, *Factoring multivariate polynomials over the integers*, Math. Comput., 29, (1975), pp. 935–950.

[6] P. S. WANG, *An improved multivariate polynomial factoring algorithm*, Ibid., 32 (1978), pp. 1215–1231.

[7] D. Y. Y. YUN, *On square-free decomposition algorithms*, Proceedings of 1976 ACM SYMSAC, Aug. 1976 (Yorktown Heights, NY), pp. 26–35.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0003 $01.00/0

SYMBOLIC VECTOR AND DYADIC ANALYSIS*

MICHAEL C. WIRTH†

Abstract. A computer program is described which performs symbolic algebra and calculus with vectors and dyadics. Implemented on the MACSYMA algebraic manipulation system, it is intended to be a reasonably complete analysis system for applications such as plasma physics and fluid dynamics. It includes manipulations of dot and cross products; gradient, divergence, curl and Laplacian operators; directional derivatives and outer products. Vector and dyadic equations can be automatically expanded into components for arbitrary orthogonal coordinate systems. Designed primarily for 2-dimensional and 3-dimensional use, the program also has some capability for higher dimensions. The internal structure of the code is discussed with regard to efficiency considerations. Example calculations and execution times are presented.

Key words. computer algebra, algebraic manipulation, vectors, dyadics, tensors, vector calculus, MACSYMA, fluid dynamics, magnetohydrodynamics

Introduction. The algebra and calculus of vectors and dyadics is indispensable to many areas of the engineering and physical sciences. For example, it is extensively used in computational fluid dynamics for applications such as aerodynamics and magnetohydrodynamics. In both of these areas, it allows the description and manipulation of the physical equations which govern fluid flow, independent of coordinate system and in a compact form. These vector/dyadic equations can then be converted to component form (with respect to a particular coordinate system), resulting in a set of scalar partial differential equations. These, in turn, can be solved numerically by techniques such as finite differencing or spectral methods.

The author has been using the algebraic manipulation system, MACSYMA[1], to build software tools which automate these steps, including the generation of FORTRAN code for finite difference calculations. An earlier MACSYMA program developed by David R. Stoutemyer [5] was used for the symbolic vector analysis portion of this work. It provided the capability to expand and simplify vector differential equations, and express them in component form for a wide variety of orthogonal coordinate systems. Scalar and vector potentials could also be calculated. Unfortunately, it lacked some desired capabilities: the vector simplifications were not complete enough; directional derivatives and outer products were not handled; and dyadics[2] could not be used.

This paper describes an extended and restructured MACSYMA program that corrects these deficiencies. Its capabilities are described, the internal structure of the code and related efficiency issues are discussed, and a list of possible improvements is presented. Example calculations and execution times are presented in the Appendix.

Capabilities. The system can manipulate scalars, vectors and dyadics. Expressions containing higher rank tensors composed of these objects (e.g., the outer product of 3

* Received by the editors May 23, 1978.

† AF Flight Dynamics Laboratory, Wright-Patterson Air Force Base, Ohio 45433.

[1] MACSYMA is a large computer programming system developed by the Mathlab Group of the MIT Laboratory for Computer Science [2]. Accessible over the ARPA network, it provides symbolic manipulation facilities for differentiating, integrating, taking limits, solving systems of linear or polynomial equations, factoring polynomials, expanding functions in Laurent or Taylor series, solving differential equations (by direct or transform methods), etc. It also provides a language similar to ALGOL-60 for writing user programs.

[2] The term "dyadic" is used here to denote an unindexed object composed of physical components, as distinguished from "second-rank tensor" which is generally used to denote an indexed object of contravariant-covariant components [3]. This is the same distinction that is usually drawn between "vector" and "first-rank tensor."

vectors) can be manipulated and simplified, but cannot be expressed in component form. Scalars, vectors and dyadics can take any of the forms shown in Table 1.

TABLE 1
Forms of scalars, vectors and dyadics.

Object	Form	Representation	Examples*
Scalar	Named object	A MACSYMA variable	F, G
Scalar	Single vector or dyadic component	An indexed vector name, or doubly-indexed dyadic name	$A[1]$, $T[1, 2]$
	Expression	Any MACSYMA scalar expression or an expression composed of scalar, vector and dyadic objects which evaluates to a scalar.	$F+G$, $A \cdot B$
Vector	Named object	A MACSYMA variable which is declared to be a vector (e.g., VECTOR (A, B, C, D, E);)	A, B, C, D, E
Vector	Collection of components	A MACSYMA list of explicit, scalar components	$[1, 2, 3]$, $[X, X+Y, 2*Z+3]$
Vector	Expression	An expression composed of scalar, vector and dyadic objects and the operations from Table 2 which evaluates to a vector	$A+B$, GRAD F, $A+[1, 2, 3]$
Dyadic	Named object	A MACSYMA variable which is declared to be a dyadic (e.g., DYADIC(T, U);)	T, U
Dyadic	Collection of components	A MACSYMA matrix of explicit, scalar components	$[1\ \ 2\ \ 3\ ]$ $[X\ \ Y\ \ Z\ ]$ $[P\ \ Q\ \ X+Y]$
Dyadic	Expression	An expression composed of scalar, vector and dyadic objects and the operations from Table 2 which evaluates to a dyadic	$T+U$, $T \cdot U$

* The variables A, B, C, D, and E will be consistently used as vectors in this paper, T and U as dyadics.

TABLE 2
Operators.

Operation	Syntax	Type of operator
Addition	$A+B$	Infix, binary operator
Subtraction	$A-B$	Infix, binary operator
Component by component multiplication	$A*B$	Infix, binary operator
Component by component division	A/B	Infix, binary operator
Dot (inner) product	$A \cdot B$	Infix, binary operator
Cross product	$A \tilde{} B$	Infix, binary operator
Gradient	GRAD F	Prefix, unary operator
Divergence	DIV A	Prefix, unary operator
Curl	CURL A	Prefix, unary operator
Laplacian	LAPLACIAN A	Prefix, unary operator
Directional derivative	A DOTDEL B	Infix, binary operator
Outer product	$\{A, B, C\}$	Operator with arbitrary number of arguments

All variables are assumed to be scalars unless declared otherwise (as noted in Table 1). These objects can be composed into expressions using the algebraic and differential operators shown in Table 2. Note that the component by component operations of scalar multiplication, "*", and scalar division, "/", are a by-product of the normal MACSYMA facilities for element by element operations with lists and matrices.

The program offers three types of capabilities for manipulating these expressions: expression simplification, specification of coordinate system, and conversion to component form.

Simplification. The simplification of vector and dyadic expressions involves the application of standard vector theorem transformations under the control of user-set flags. For example, executing VECTORSIMP($A\tilde{}(B\tilde{}C)$); with the flag EXPAND-CROSS set to TRUE, results in the value $(A \cdot C) * B - (A \cdot B) * C$. The set of transformations which was implemented was chosen to be a reasonably complete set and was taken directly from a review article commonly used by plasma physicists [1]. The Appendix gives examples of these transformations. The capabilities of the simplifier are discussed in more detail in the following section on the structure of the code.

Coordinate system specification. Arbitrary curvilinear orthogonal coordinate systems can be used by specifying an analytic transformation from the coordinate system to Cartesian coordinates. For example, spherical coordinates are specified by:

SCALEFACTORS([[R * SIN(THETA) * COS(PHI), R * SIN(THETA) * SIN(PHI), R * COS(THETA)], R, THETA, PHI]);

The last three elements of the outer list are the new coordinate variables, and the three elements of the inner list specify the transformation from the new coordinates to Cartesian coordinates. The number of elements in these lists denotes the dimension.

Orthogonal coordinate systems of dimension higher than 3 can be specified and handled by the system, with the exception of expressions involving the operations of cross product and curl. The tensor rank of the results of these operations depends on the dimension of the space used. For example, the cross product of two vectors (or the curl of one vector) in 2-dimensions and 3-dimensions is a scalar and a vector, respectively, but in 4-dimensions is a dyadic. The current version of the system is not capable of expanding a dyadic formed by a 4-dimensional vector cross product (or curl) into component form. With the exception of cross product and curl operations, the system is usuable for arbitrary dimension. Note that the dimension-dependent behavior of these operations also implies that the dimension of the space must be chosen (SCALE-FACTORS must be executed) before simplifying expressions involving these operations, not just before expanding into component form.

Because the program is designed for curvilinear orthogonal coordinate systems, all of the differential operations (i.e., gradient, divergence, etc.) can be defined in terms of scale factors [1, 3], SF_i. In 3-dimensions, these are related to the differential arc length dL by

$$dL^2 = dX^2 + dY^2 + dZ^2 = SF_1^2 * dE_1^2 + SF_2^2 * dE_2^2 + SF_3^2 * dE_3^2$$

where the E_i designate the new curvilinear coordinates, and X, Y, and Z are Cartesian coordinates. The SCALEFACTORS function calculates these scale factors from the Jacobian of the transformation specified in its argument and stores them in an array named SF. For example, in spherical coordinates,

$$dL^2 = dX^2 + dY^2 + dZ^2 = dR^2 + R^2 * d\text{THETA}^2 + R^2 * \text{SIN(THETA)}^2 * d\text{PHI}^2$$

and the scalefactors are

$$SF_1 = 1, \qquad SF_2 = R, \qquad SF_3 = R * \text{SIN(THETA)}$$

Some of the differential operations such as the gradient of a vector and the divergence of a dyadic involve collections of derivatives of the scale factors which make it convenient to introduce the Christoffel symbols defined by

$$\text{CHRISTOFFEL}_{I,J,K} = \frac{\text{DELTA}(I,J)\left(\frac{d}{dE_K} SF_J\right) - \text{DELTA}(I,K)\left(\frac{d}{dE_J} SF_K\right)}{SF_J SF_K}$$

where DELTA(I, J) equals 1 if $I = J$ and 0 otherwise [1]. The SCALEFACTORS function also calculates the Christoffel symbols and stores them in an array named CHRISTOFFEL.

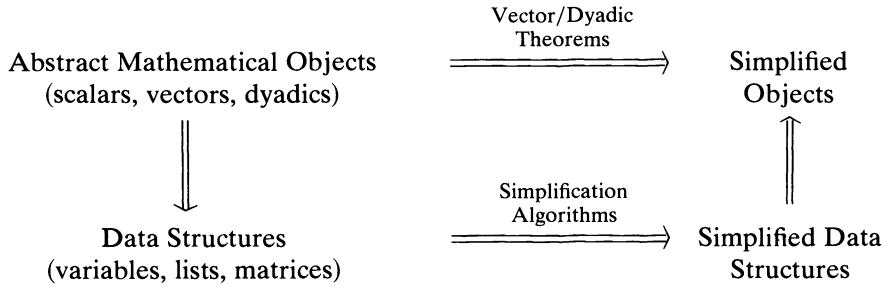
Conversion to component form. The EXPRESS function converts a scalar, vector, or dyadic expression into a set of scalar component expressions with respect to the currently specified coordinate system. Scalars remain scalars, vectors become lists of components, and dyadics become matrices of components. The various differential operators (GRAD, DIV, etc.) become differential expressions in the components which involve the scalefactors and, in some cases, Christoffel symbols. For example, for general curvilinear coordinates $[E_1, E_2, E_3]$, the gradient of a vector A is a dyadic T with components:

$$T_{I,J} = \left(\sum_{K=1}^{3} \text{CHRISTOFFEL}_{I,J,K} * A_K \right) + \frac{\frac{d}{dE_I} A_J}{SF_I}$$

These differential expressions in the scalar components can then be manipulated by the standard MACSYMA facilities.

Structure of the code.

Modeling. Scientific programming can often be viewed as a modeling process consisting of two parts: a set of data structures which represent the objects modeled, and algorithms (programs) which simulate the transformations the objects undergo in the modeled physical or mathematical process. The figure below shows the correspondence between abstract vector simplification and the computational model that is used in the expression simplifier portion of the program. The abstract mathematical objects are represented by data structures, the transformations which these objects undergo when vector/dyadic theorems are applied are modeled by the simplification algorithms and the resulting data structures represent the simplified abstract objects.



Data structures. As the figure illustrates, data structures and algorithms play an interdependent role in algebraic manipulation. The ease with which the transformation algorithms can be constructed is critically dependent on the structures chosen to represent the abstract objects. If the set of transformations to be modeled is too broad, multiple representations may be needed for the same abstract object, each one appropriate to a particular class of transformations [4]. For example, MACSYMA offers a large set of mathematical capabilities and uses multiple representations [2]. In this program, the following representations are used (also see Table 1): A symbolic vector (dyadic) is represented by a named variable which is declared to MACSYMA to be "NONSCALAR" and which carries a "TENSOR" property with a value of 1 (2). The component form of a vector (dyadic) consists of a list (matrix) of scalar components. The simplifier treats the named object and its component form identically; the EXPRESS function produces the second from the first. Expressions composed of these structures are represented in MACSYMA's general recursive form. These structures were chosen to take advantage of the existing MACSYMA facilities for handling nonscalar variables and for performing algebra with lists and matrices.

Expansion control. When the user simplifies a vector expression by hand, a certain goal is usually in mind. A sequence of transformations are applied which drive the expression toward a particular form. The choice of the goal, the "simplest" such form, is a subjective judgment and is often not easy to automate [4]. Our simplifier function leaves this judgment up to the user; by setting flag variables the user can control which vector/dyadic transformations are applied by the simplifier.

But this is not a complete solution; the user could have chosen an inconsistent set of flag settings. For example, the system contains an expansion for the Laplacian of a vector:

$$\text{LAPLACIAN } A \Rightarrow \text{GRAD DIV } A - \text{CURL CURL } A$$

and it also contains an expansion for curl of curl:

$$\text{CURL CURL } A \Rightarrow \text{GRAD DIV } A - \text{LAPLACIAN } A.$$

If both of these transformations were enabled at the same time, an infinite circular substitution would result.

This problem is solved for the case of an operator applied to a compound argument by always applying the transformations in the direction that implies expansion. Thus, the simplifier will expand $\text{GRAD}(F * G)$ to $F * \text{GRAD}(G) + G * \text{GRAD}(F)$, but will not do the reverse transformation. In those cases where the transformation is not an argument expansion and where circular substitutions are possible, as in the case of LAPLACIAN A above, special checks are included in the simplifier algorithm.

The set of expansion control flags which are used by the simplifier is shown in Table 3. These flags are arranged in a hierarchy. EXPANDALL when set to TRUE enables all expansions (except for the special case, LAPLACIAN vector). EXPANDPLUS enables all expansions of linear operators

$$\text{OP}(A + B) \Rightarrow \text{OP}(A) + \text{OP}(B).$$

EXPANDGRAD enables all expansions of GRAD. EXPANDGRADDOT is more specific and controls just expansions of $\text{GRAD}(A \cdot B)$ forms.

TABLE 3
*Expansion control flags.**

Flag variable	Expansions controlled
EXPANDALL	All forms below, except LAPLACIAN vector
EXPANDPLUS	Operator(term + term)
EXPANDPROD	Operator(factor * factor)
EXPANDDOT	All "·" forms.
EXPANDDOTPLUS	(term + term) · (term + term)
EXPANDDOTPROD	(factor * factor) · (factor * factor)
EXPANDCROSS	all "~" forms.
EXPANDCROSSPLUS	(term + term)~(term + term)
EXPANDCROSSPROD	(factor * factor)~(factor * factor)
EXPANDCROSSCROSS	arg~(arg~arg) or (arg~arg)~arg
EXPANDGRAD	All "GRAD" forms.
EXPANDGRADPLUS	GRAD(term + term)
EXPANDGRADPROD	GRAD(factor * factor)
EXPANDGRADDOT	GRAD(arg · arg)
EXPANDDIV	All "DIV" forms.
EXPANDDIVPLUS	DIV(term + term)
EXPANDDIVPROD	DIV(factor * factor)
EXPANDDIVCROSS	DIV(arg~arg)
EXPANDCURL	All "CURL" forms.
EXPANDCURLPLUS	CURL(term + term)
EXPANDCURLPROD	CURL(factor * factor)
EXPANDCURLCROSS	CURL(arg~arg)
EXPANDCURLCURL	CURL CURL arg, Null effect if EXPANDLAPLVECT is set.
EXPANDLAPL	All "LAPLACIAN" forms, except LAPLACIAN vector
EXPANDLAPLPLUS	LAPLACIAN(term + term)
EXPANDLAPLPROD	LAPLACIAN(factor * factor)
EXPANDLAPLSCLR	LAPLACIAN(scalar)
EXPANDLAPLVECT	LAPLACIAN(vector), Note comment on EXPANDCURLCURL.
EXPANDOUTP	All "{arg, · · · , arg}" forms.
EXPANDDOTDEL	All "arg DOTDEL arg" forms.

* 1. Arg, factor, and term are arbitrary arguments, factors and terms, respectively.
2. The variable, EXPANDFLAGS, has as value a list of the names of all the expansion flags.
3. The default value for all flags is FALSE.

Canonical forms. Besides expansion, the simplifier also transforms expressions into canonical form. For example, $B \cdot A$, where A and B are vectors, is transformed into $A \cdot B$, based on the internal MACSYMA lexical ordering of expressions. This allows reductions to take place by cancellation of like terms, e.g., $A \cdot B - B \cdot A \Rightarrow A \cdot B - A \cdot B \Rightarrow 0$. The combination of expansion and transformation to canonical form allows expressions like $A \tilde{} B \tilde{} C + B \tilde{} C \tilde{} A + C \tilde{} A \tilde{} B$ to be reduced to zero.

Transformation mechanisms. MACSYMA offers a number of facilities for implementing these simplifier transformations, all of which were used by Stoutemyer's vector-expression simplifier: the MACSYMA internal simplifier, pattern matching and replacement facilities, and the ability to write a function which the user can explicitly call to accomplish simplification.

The MACSYMA internal simplifier transforms an expression based on its lead operator and recursively simplifies subexpressions. It handles all of the normal arithmetic operations, and accommodates both scalars and nonscalars. The operator "·" is normally treated as noncommutative multiplication. Various flags control simplification. Thus by setting DOTDISTRIB true, the original vector-only program directs the MACSYMA simplifier to expand $A \cdot (B + C)$ to $A \cdot B + A \cdot C$. New operators and functions can be declared to have special properties. For example, if the flag, EXPANDGRAD, is set to true, then the vector package when invoked temporarily declares GRAD to be linear and the internal MACSYMA simplifier will transform GRAD$(3 * F)$ into $3 * (\text{GRAD}\, F)$. Similarly, the operator "·" is declared to be commutative by the original program so that MACSYMA will transform dot products into canonical order.

MACSYMA allows pattern matching rules to be defined which may be applied locally or globally. The original program specified a number of global rules for canonical transformations such as $A \tilde{} B \cdot C \Rightarrow A \cdot B \tilde{} C$. The anti-commutative property for cross products was implemented by a global rule which performed the replacement $B \tilde{} A \Rightarrow -A \tilde{} B$ using a pattern of the form

ANY~BEFORE.

ANY is declared to be a pattern variable that matches any expression. BEFORE is declared to be a pattern variable that matches any expression satisfying ORDER(BEFORE,ANY), a predicate which tests to see if the right hand side of the cross product is lexicographically before the left hand side. Thus the above pattern matches only if a cross product is out of canonical order. The matched expression is then replaced by −BEFORE~ANY.

Finally, the user can write a MACSYMA function which explicitly performs any required transformation. The original program contains such a function, VECTORSIMP. Given a vector expression, it interprets the user-set expansion flags and declares the appropriate properties to the internal simplifier. For example, if any or all of EXPANDDOTPLUS, EXPANDPLUS, EXPANDDOT, or EXPANDALL are set, then DOTDISTRIB is set so that MACSYMA will expand dot products of sums. Then VECTORSIMP invokes VSIMP which examines the lead operator of expressions, sequentially checking them for equivalence with one of "~", GRAD, DIV, CURL, or LAPLACIAN. If one of these operators is found and the appropriate expansion flag is set, an ad hoc replacement is made. Subexpressions are recursively simplified.

Type checking. The extension of the original vector-only system required more than simply adding a data structure for dyadics and code for the additional manipulations. Some of the original vector simplifications were no longer applicable. For example, cross product is anti-commutative for vectors, but not for dyadics. Thus the global pattern replacement rule $B \tilde{} A \Rightarrow -A \tilde{} B$ needed to check not only the lexicographic ordering of A and B but also whether they were vectors or dyadics.

Therefore, a type checking function, VTYPE, was introduced which determines the tensor rank of an expression (scalar $\Rightarrow$ 0, vector $\Rightarrow$ 1, dyadic $\Rightarrow$ 2, etc.) and conditions the application of vector/dyadic transformations. Unfortunately, VTYPE is expensive to execute; it needs to recursively examine each subexpression and variable of an expression to determine its rank. This makes the matching of pattern ANY~BEFORE time consuming. To compound the problem, this global match is repeatedly attempted by MACSYMA whenever an expression involving a cross product is evaluated, even when there is little probability of a successful match. A

particularly blatant example of this problem is the reduction of the expression $A \tilde{} B \tilde{} C + B \tilde{} C \tilde{} A + C \tilde{} A \tilde{} B$ to zero. In the original system, which did no type checking, this reduction took approximately 3.4 seconds. In a preliminary version of the extended system after VTYPE was included, this time increased to over 25 seconds. The global pattern match for ANY~BEFORE was attempted almost 1,000 times and VTYPE was executed over 2,100 times! The use of global pattern matching rules was convenient for programming, but in this case, proved to be very inefficient. In the current version of the system, almost all uses of the pattern matching facilities have been replaced with explicit code in the VECTORSIMP function. The execution time for the reduction of $A \tilde{} B \tilde{} C + B \tilde{} C \tilde{} A + C \tilde{} A \tilde{} B$ to zero is now 1.3 seconds.

Structure of the simplifier. The increased generality of the simplifier, the necessity for type checking, and efficiency considerations led to the following structure for vector and dyadic simplification in the current system:

1. The internal MACSYMA features for manipulating nonscalar variables are not used because the required simplifications differ between vectors and dyadics.
2. The use of pattern matching facilities has been reduced to a few special cases because of the efficiency problems mentioned above.
3. The function VECTORSIMP now does the bulk of the simplification work. It is table-driven and structured much like the MACSYMA internal simplifier. Called with a single argument, a vector/dyadic expression, it recursively examines subexpressions, invokes operator-associated simplification functions (e.g., the GRAD operator has an associated function GRADSIMP), marks simplified subexpressions to prevent their unnecessary rescanning, and returns the simplified result.
4. The function VTYPE marks expressions internally with their tensor rank so that subsequent applications of VTYPE to them execute quickly.

The net result of these changes is that the new vector/dyadic simplifier is more powerful and often executes faster than the original vector-only version. Simplification examples and run times which compare the two versions are shown in the Appendix.

Structure of SCALEFACTORS and EXPRESS. The coordinate system facilities embodied in the SCALEFACTORS function, and the component form facilities in EXPRESS are straightforward revisions of the original versions. SCALEFACTORS now computes the Christoffel symbols for the new orthogonal coordinate system as well as the scalefactors. EXPRESS has been extended to handle directional derivatives, outer products and dyadics. Like VECTORSIMP, it is table-driven; each vector operator has an associated function (e.g., DOTEXPRESS for dot product) which is invoked when an expression with that lead operator is encountered. This means that new vector operators can be added to the system without changing its structure by defining functions for the simplification and conversion to component form.

Possible improvements. The current MACSYMA program is a reasonably complete system for vector and dyadic algebra and differential calculus. Future improvements might include:

1. Incorporation of VECTORSIMP into the internal MACSYMA simplifier so that it need not be called explicitly.
2. Elimination of the EXPAND flags for controlling simplification since they are too numerous (30 now) and too cumbersome, to be replaced by a subexpression selection facility (like MACSYMA's BOX function) which allows the user to specify subexpressions based on position rather than operators contained.

3. Addition of line and wavy line underscores for typeout of vector and dyadic variables.

4. Extension of the system to integral calculus for vectors and dyadics (e.g., Stokes' theorem and Gauss' theorem).

Appendix. Examples and execution times. The following examples have been extracted from actual output files and have been edited only to save space and to add the columns which compare execution times for the current system with those for the original program by Stoutemyer. These times are for interpreted MACSYMA code.

Expression simplification. The following MACSYMA statements were executed before running the simplification examples shown below:

BATCH(VECT, ">",DSK,SHARE);	Load latest version of vector/dyadic package from SHARE directory.
DECLARE([$K1, K2$], CONSTANT);	Declare $K1$ and $K2$ to be constants. Note: F and G are scalars by default.
VECTOR(A, B, C, D, E);	Declare A, B, C, D, and E to be vectors.
DYADIC(T, U);	Declare T and U to be dyadics.
FOR I IN EXPANDFLAGS DO I::TRUE;	Set all of the expansion flags to true.

VECTORSIMP(expression) ⇒ simplified expression	Execution time (msec.) Current system	Original system
$A \cdot B\tilde{}C \Rightarrow A \cdot B\tilde{}C$, $A\tilde{}B \cdot C \Rightarrow A \cdot B\tilde{}C$		
$B \cdot C\tilde{}A \Rightarrow A \cdot B\tilde{}C$, $B\tilde{}C \cdot A \Rightarrow A \cdot B\tilde{}C$		
$C \cdot A\tilde{}B \Rightarrow A \cdot B\tilde{}C$, $C\tilde{}A \cdot B \Rightarrow A \cdot B\tilde{}C$	420 avg	520 avg
$A \cdot B \Rightarrow A \cdot B$, $B \cdot A \Rightarrow A \cdot B$, $A \cdot T \Rightarrow A \cdot T$, $T \cdot A \Rightarrow T \cdot A$	75	Not handled
$A\tilde{}B \Rightarrow A\tilde{}B$, $B\tilde{}A \Rightarrow -A\tilde{}B$, $A\tilde{}T \Rightarrow A\tilde{}T$, $T\tilde{}A \Rightarrow T\tilde{}A$	70	Not handled
$0\tilde{}A \Rightarrow 0$, $A\tilde{}0 \Rightarrow 0$, $A\tilde{}A \Rightarrow 0$	50	35
DIV CURL A ⇒ 0, CURL GRAD $F \Rightarrow 0$	8	8
GRAD $K1 \Rightarrow 0$, LAPLACIAN $K1 \Rightarrow 0$	5	Not handled
GRAD $(K1 * F) \Rightarrow K1$ GRAD F	6	5
DIV $(K1 * A) \Rightarrow K1$ DIV A	6	6
CURL $(K1 * A) \Rightarrow K1$ CURL A	6	6
LAPLACIAN $(K1 * F) \Rightarrow K1$ LAPLACIAN F	6	5
$A \cdot (B+C) \Rightarrow A \cdot C + A \cdot B$	158	69
$A \cdot \{B, C\} \Rightarrow A \cdot BC$	232	Not handled
$\{A, B\} \cdot C \Rightarrow AB \cdot C$	222	Not handled
$(K1 * A) \cdot (K2 * B) \Rightarrow K1K2A \cdot B$	93	75
$A\tilde{}(B+C) \Rightarrow A\tilde{}C + A\tilde{}B$	248	1227
$(A+B)\tilde{}C \Rightarrow B\tilde{}C + A\tilde{}C$	246	1229
$(F * A)\tilde{}(G * B) \Rightarrow A\tilde{}BFG$	309	580
$A\tilde{}(B\tilde{}C) \Rightarrow A \cdot CB - A \cdot BC$	306	995
$(C\tilde{}B)\tilde{}A \Rightarrow A \cdot CB - A \cdot BC$	552	831
$A\tilde{}\{B, C\} \Rightarrow \{A\tilde{}B, C\}$	305	Not handled
$\{A, B\}\tilde{}C \Rightarrow \{A, B\tilde{}C\}$	296	Not handled

VECTORSIMP(expression) ⇒ simplified expression	Execution time (msec.) Current system	Original system
$A\tilde{}B\tilde{}C+B\tilde{}C\tilde{}A+C\tilde{}A\tilde{}B \Rightarrow 0$	1264	3392
$(A\tilde{}B)\cdot(C\tilde{}D) \Rightarrow A\cdot CB\cdot D-A\cdot DB\cdot C$	975	Not fully expanded
$(A\tilde{}B)\tilde{}(C\tilde{}D) \Rightarrow A\cdot C\tilde{}DB-AB\cdot C\tilde{}D$	1670	2250
GRAD$(F+G) \Rightarrow$ GRAD $G+$GRAD F	108	59
GRAD$(F*G) \Rightarrow F$ GRAD $G+$GRAD F G	186	79
GRAD$(A\cdot B) \Rightarrow A$ DOTDEL $B-$CURL $A\tilde{}B$ $+A\tilde{}$ CURL $B+B$ DOTDEL A	468	Not handled
DIV$(A+B) \Rightarrow$ DIV $B+$DIV A	109	63
DIV$(F*A) \Rightarrow$ DIV A $F+A\cdot$ GRAD F	336	90
DIV$(F*T) \Rightarrow F$ DIV $T+$GRAD $F\cdot T$	233	90
DIV$(A\tilde{}B) \Rightarrow$ CURL $A\cdot B-A\cdot$ CURL B	363	Not handled
DIV$\{A, B\} \Rightarrow A$ DOTDEL $B+$DIV A B	150	Not handled
CURL$(A+B) \Rightarrow$ CURL $B+$CURL A	109	65
CURL$(F*A) \Rightarrow$ CURL A $F-A\tilde{}$GRAD F	270	Not fully expanded
CURL$(A\tilde{}B) \Rightarrow -A$ DOTDEL $B+A$ DIV B $-$DIV A $B+B$ DOTDEL A	220	Not fully expanded
CURL CURL $A \Rightarrow$ GRAD DIV $A-$LAPLACIAN A	165	75
Note. With EXPANDLAPLVECT set to FALSE.		
LAPLACIAN$(A+B) \Rightarrow$ GRAD DIV B $-$CURL CURL $B+$GRAD DIV $A-$CURL CURL A	350	Not fully expanded
LAPLACIAN$(F*G) \Rightarrow F$ DIV GRAD G $+$DIV GRAD F $G+2$ GRAD $F\cdot$ GRAD G	490	Not fully expanded
LAPLACIAN$(F*A) \Rightarrow A$ DIV GRAD F $+$GRAD DIV A $F-$CURL CURL A F $+2$ (GRAD F) DOTDEL A	403	Not fully expanded
LAPLACIAN $F \Rightarrow$ DIV GRAD F	107	46
LAPLACIAN $A \Rightarrow$ GRAD DIV $A-$CURL CURL A	165	Not fully expanded
$\{A\} \Rightarrow A$	40	Not handled
$\{A, B+C, D+E\} \Rightarrow \{A, C, E\}+\{A, C, D\}+\{A, B, E\}$ $+\{A, B, D\}$	360	Not handled
$\{A, F*B, G*C\} \Rightarrow \{A, B, C\}FG$	190	Not handled
C DOTDEL$(A+B) \Rightarrow C$ DOTDEL A $+$C DOTDEL B	113	Not handled
B DOTDEL$(F*A) \Rightarrow (B$ DOTDEL $A)F$ $+A$ $B\cdot$ GRAD F	240	Not handled
$(B+C)$ DOTDEL $A \Rightarrow C$ DOTDEL $A+B$ DOTDEL A	94	Not handled
$(F*B)$ DOTDEL $A \Rightarrow (B$ DOTDEL $A)F$	110	Not handled

Conversion to component form. The following examples are taken directly from MACSYMA output and demonstrate conversion to component form first for 3-dimensional Cartesian coordinates (the default dimension and coordinate system) and then for spherical coordinates. Note that integer subscripts denote components and that

coordinate variable subscripts denote differentiation. The same MACSYMA statements for initialization are used here as in the simplification examples above. The "gctimes" shown below refer to garbage collections for reclaiming list storage space, which occur periodically.

(*C*10) EXPRESS(*F*);
time = 31 msec.
(*D*10) F

(*C*11) EXPRESS($A+B$);
time = 86 msec.
(*D*11) $[B_1+A_1, B_2+A_2, B_3+A_3]$

(*C*12) EXPRESS(*T*);
time = 67 msec.

(*D*12)
$$\begin{bmatrix} T_{1,1} & T_{1,2} & T_{1,3} \\ T_{2,1} & T_{2,2} & T_{2,3} \\ T_{3,1} & T_{3,2} & T_{3,3} \end{bmatrix}$$

(*C*13) EXPRESS($A \cdot B$);
time = 85 msec.
(*D*13) $A_3 * B_3 + A_2 * B_2 + A_1 * B_1$

(*C*14) EXPRESS($A \cdot T$);
time = 119 msec.
(*D*14) $[A_3 * T_{3,1} + A_2 * T_{2,1} + A_1 * T_{1,1}, A_3 * T_{3,2} + A_2 * T_{2,2} + A_1 * T_{1,2},$
$A_3 * T_{3,3} + A_2 * T_{2,3} + A_1 * T_{1,3}]$

(*C*15) EXPRESS($A \tilde{} B$);
time = 167 msec.
(*D*15) $[A_2 * B_3 - B_2 * A_3, B_1 * A_3 - A_1 * B_3, A_1 * B_2 - B_1 * A_2]$

(*C*16) EXPRESS(GRAD *F*);
totaltime = 643 msec. gctime = 557 msec.
(*D*16) $[F_X, F_Y, F_Z]$

(*C*17) EXPRESS(GRAD *A*);
time = 402 msec.

(*D*17)
$$\begin{bmatrix} A_1 & A_2 & A_3 \\ X & X & X \\ A_1 & A_2 & A_3 \\ Y & Y & Y \\ A_1 & A_2 & A_3 \\ Z & Z & Z \end{bmatrix}$$

(*C*18) EXPRESS(DIV *A*);
time = 122 msec.
(*D*18)
$$A_{3_Z} + A_{2_Y} + A_{1_X}$$

(*C*19) EXPRESS(CURL *A*);
time = 137 msec.
(*D*19)
$$[A_{3_Y} - A_{2_Z}, A_{1_Z} - A_{3_X}, A_{2_X} - A_{1_Y}]$$

(*C*20) EXPRESS(LAPLACIAN *F*);
time = 92 msec.
(*D*20)
$$F_{ZZ} + F_{YY} + F_{XX}$$

(*C*21) EXPRESS(*A* DOTDEL *B*);
totaltime = 873 msec. gctime = 494 msec.
(*D*21)
$$[A_3 * B_{1_Z} + A_2 * B_{1_Y} + A_1 * B_{1_X}, A_3 * B_{2_Z} + A_2 * B_{2_Y} + A_1 * B_{2_X},$$
$$A_3 * B_{3_Z} + A_2 * B_{3_Y} + A_1 * B_{3_X}]$$

(*C*22) EXPRESS({*A*, *B*});
time = 167 msec.
(*D*22)
$$\begin{bmatrix} A_1 * B_1 & A_1 * B_2 & A_1 * B_3 \\ B_1 * A_2 & A_2 * B_2 & A_2 * B_3 \\ B_1 * A_3 & B_2 * A_3 & A_3 * B_3 \end{bmatrix}$$

(*C*23) SPHERICAL: [[*R* * SIN(THETA) * COS(PHI), *R* * SIN(THETA) * SIN(PHI),
R * COS(THETA)], *R*, THETA, PHI]$
time = 14 msec.

(*C*24) SCALEFACTORS(SPHERICAL)$

$$SF_1 = 1$$

$$SF_2 = R$$

$$SF_3 = R * \text{SIN(THETA)}$$

$$\text{CHRISTOFFEL}_{2,1,2} = -\frac{1}{R}$$

$$\text{CHRISTOFFEL}_{2,2,1} = \frac{1}{R}$$

$$\text{CHRISTOFFEL}_{3,1,3} = -\frac{1}{R}$$

$$\text{CHRISTOFFEL}_{3,2,3} = -\frac{\text{COS(THETA)}}{R * \text{SIN(THETA)}}$$

$$\text{CHRISTOFFEL}_{3,1,1} = \frac{1}{R}$$

$$\text{CHRISTOFFEL}_{3,3,2} = \frac{\text{COS(THETA)}}{R * \text{SIN(THETA)}}$$

totaltime = 11,572 msec. gctime = 3,594 msec.

(*C*25) EXPRESS(DIV *A*);
time = 128 msec.

$$(D25)\quad \frac{(A_1 * R^2 * \text{SIN(THETA)})_R + (A_2 * R * \text{SIN(THETA)})_{\text{THETA}} + (A_3 * R)_{\text{PHI}}}{R^2 * \text{SIN(THETA)}}$$

(*C*26) EXPRESS(GRAD *F*);
time = 86 msec.

$$(D26)\quad \left[F_R, \frac{F_{\text{THETA}}}{R}, \frac{F_{\text{PHI}}}{R * \text{SIN(THETA)}}\right]$$

(*C*27) EXPRESS(CURL *A*);
time = 145 msec.

$$(D27)\quad \left[\frac{(A_3 * R * \text{SIN(THETA)})_{\text{THETA}} - (A_2 * R)_{\text{PHI}}}{R^2 * \text{SIN(THETA)}}, \frac{A_{1_{\text{PHI}}} - (A_3 * R * \text{SIN(THETA)})_R}{R * \text{SIN(THETA)}}, \frac{(A_2 * R)_R - A_{1_{\text{THETA}}}}{R}\right]$$

(*C*28) EXPRESS(LAPLACIAN *F*);
time = 102 msec.

(*D*28)

$$\frac{(F_R * R^2 * \text{SIN(THETA)})_R + (F_{\text{THETA}} * \text{SIN(THETA)})_{\text{THETA}} + \left(\frac{F_{\text{PHI}}}{\text{SIN(THETA)}}\right)_{\text{PHI}}}{R^2 * \text{SIN(THETA)}}$$

Acknowledgment. I would like to acknowledge the following people and organizations for their assistance:

David R. Stoutemyer for providing the original vector manipulation system.

The MACSYMA Consortium for providing MACSYMA, which is supported, in part, by the Department of Energy under contract Number E(11-1)-3070 and by the National Aeronautics and Space Administration under Grant NSG 1323.

Brendan McNamara, from Lawrence Livermore Laboratory, for his guidance in making this program useful to real users.

The reviewers for their suggestions which improved the structure of this paper.

REFERENCES

[1] D. L. BROOK, *Revised and enlarged collection of plasma physics formulas and data*, NRL Memorandum Report 3332, Naval Research Laboratory, May 1977.

[2] *MACSYMA Reference Manual*, Version Nine, Laboratory for Computer Science, MIT, Cambridge, 1977.

[3] P. M. MORSE AND H. FESHBACH, *Methods of Theoretical Physics*, Vol. I, McGraw-Hill, New York, 1953.

[4] J. MOSES, *Algebraic Simplification: A Guide for the Perplexed*, Comm. ACM, 14 (1971), pp. 527–537.

[5] D. R. STOUTEMYER, *Symbolic computer vector analysis*, Comput. Math. Appl., to appear.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0004 $01.00/0

A NATURAL STRUCTURE THEOREM FOR COMPLEX FIELDS*

H. I. EPSTEIN†

Abstract. This paper presents a theorem which describes the structure of algebraic relationships which must hold when a certain set of transcendental functions are algebraically dependent. The functions in the set may be logarithmic, exponential, trigonometric, hyperbolic, or indefinite integrals. This structure theorem has important applications to symbolic mathematical computation. A procedure for finding regular representations for classes of transcendental expressions based upon the structure theorem is discussed. By use of this representation procedure, it is possible to solve the equivalence problem for expressions in the class being considered.

Key words. symbolic mathematical computation, simplification, regular representation, structure theorem, transcendental function arithmetic, exponential function, logarithmic function, trigonometric function, hyperbolic function, algebraic independence, differential field, differential algebra

1. Introduction. In this paper we show how a theorem, which tells whether a set of transcendental functions are algebraically independent, can be used to develop a procedure for representing functions in transcendental function fields. The function fields we deal with, called eplath fields, are those obtained by starting with the rational functions and the algebraic operations and then taking indefinite integrals, or applying the exponential function, a trigonometric function, or a hyperbolic function, to a function previously obtained.

In § 2 of this paper, the basic notions necessary to define eplath field precisely are presented. In § 3, eplath field is defined and the natural structure theorem for eplath fields is presented. This theorem is a generalization of the structure theorem in [7] and [8]. That theorem, first proven by Rothstein, is a structure theorem dealing with what Caviness and Rothstein call a generalized log-explicit field. (See [7] or [8] for precise definitions.) For our purposes, it is sufficient to know that an eplath field can always be embedded in a generalized log-explicit field. Thus, any function field to which our natural structure theorem applies, can be embedded in a field to which Caviness and Rothstein's structure theorem applies. Indeed, this is the essence of the proof of the natural structure theorem. For this reason the proofs of the structure theorem and its corollary will not be presented in this paper.

Embedding an eplath field in a generalized log-explicit field, however, requires representing trigonometric and hyperbolic functions in terms of exponentials. Thus, a system of procedures for doing symbolic computation based on the Caviness–Rothstein structure theorem, would require converting all trigonometric and hyperbolic functions into exponential equivalents in the representation step, before doing any computation. The natural structure theorem, on the other hand, offers the possibility of carrying out the computation with trigonometric and hyperbolic functions in their more natural form.

In § 4, the outline of a procedure for representing functions in an eplath field is presented. The basic idea of this procedure is analogous to the basic ideas involved in the simplification algorithms in [1] and [2] which are based on Risch's structure theorem (see [4]). They are also similar to the simplification procedures described in [7] and [8].

* Received by the editors May 23, 1978.
† Raytheon Corporation, Sudbury, Massachusetts.

2. Definitions and notation. Let F be a field and $D_1, \cdots, D_n$ mappings from F into F satisfying the three rules

$$D_j(a+b) = D_j(a) + D_j(b), \tag{1}$$

$$D_j(ab) = aD_j(b) + bD_j(a), \tag{2}$$

$$D_j(D_k(a)) = D_k(D_j(a)) \tag{3}$$

for any a, b in F and $1 \leqq j, k \leqq n$. F is called a *differential field* with *derivation operators* $D_1, \cdots, D_n$. Let F^* be a subfield of the differential field F. Then each derivation operator, D, on F has a restriction to F^* called a *derivative*. If, for any derivative, D, and any a in F^*, Da is in F^*, then F^* is itself a differential field. In this case F^* is said to be a *differential subfield* of F and F is said to be a *differential extension* of F^*.

Henceforth, it is assumed that F is a differential field with n derivation operators $D_1, \cdots, D_n$ and F^* is a subfield of F. The set $C(F) = \cap_{j=1}^{n} \{a \in F : D_j a = 0\}$ is called the *constant field of* F. Let K be a subfield of the complex numbers. In [3] Kolchin proved that to each F there corresponds a differential extension field U with the property that any finitely generated differential extension of F is isomorphic to a subfield of U via an isomorphism which commutes with the derivation operators (i.e. preserves derivatives). U will always be assumed to be such an extension of F and a differential extension of F will be identified with its isomorphic image in U.

Let a be in F^* and θ in U. If $D\theta$ is in F^* for each derivative D, then θ is said to be *primitive over* F^*. If, for each derivative D, $aD\theta = Da$ then θ is said to be a *logarithm over* F^* and is written $\theta = \log a$. If θ is primitive over F^*, θ is said to be *simple-logarithmic over* F^* if there are a_j in F^* and c in $C(U)$ such that $\theta + c$ is in $F^*(\log a_1, \cdots, \log a_k)$. Otherwise, θ is said to be *nonsimple*.

If $D\theta = \theta Da$ for each derivative in F^*, θ is said to be *exponential over* F^* and is written $\theta = \exp a$. If ψ is a hyperbolic function of a, written $\psi = \operatorname{hyp} a$, ψ is in the field $F^*(\exp a)$ which is a subfield of the differential field $F(\exp a)$. Furthermore, $\exp a$ is algebraic over $F^*(\operatorname{hyp} a)$. Similarly, if $D\theta = \theta D(ia)$ for each derivative in F^*, $\theta = \exp(ia)$. Then, if ψ is a trigonometric function of a, written $\psi = \operatorname{trig} a$, ψ is in $F^*(i, \exp(ia))$ which is a subfield of the differential field $F(i, \exp ia)$. Moreover, $\exp(ia)$ is algebraic over $F^*(\operatorname{trig} a)$.

An element, θ, in U which is transcendental over F^* and such that $C(F^*) = C(F^*(\theta))$ is said to be a *transcendental variable* over F^*.

3. The structure theorem for elementary fields. Let K be a subfield of the complex numbers. Let $F_0 = K(z_1, \cdots, z_n)$ be the differential field of rational functions in n indeterminates $z_1, \cdots, z_n$ with coefficients in K having n derivation operators $D_1, \cdots, D_n$ such that $D_j z_k = 0$ for $j \neq k$, $D_j z_j = 1$ and $D_j c = 0$ for c in K. For $j = 1, 2, \cdots, m$ let $F_j = F_0(\theta_1, \cdots, \theta_j)$. Suppose $C(F_0) = C(F_m)$ and one of the following conditions hold for each θ_j:

a. $\theta_j = \exp a_j$, $a_j \in F_{j-1}$,
b. θ_j is primitive and nonsimple over F_{j-1},
c. $\theta_j = \log a_j$, $a_j \in F_{j-1}$,
d. θ_j is algebraic over F_{j-1},
e. $\theta_j = \operatorname{trig} a_j$, $a_j \in F_{j-1}$,
f. $\theta_j = \operatorname{hyp} a_j$, $a_j \in F_{j-1}$.

Then F_m is said to be an *eplath field.* Define the index sets

$$E = \{j\colon \theta_j = \exp a_j \text{ is a transcendental variable over } F_{j-1}\};$$

$$L = \{j\colon \theta_j = \log a_j \text{ is a transcendental variable over } F_{j-1}\};$$

$$T = \{j\colon \theta_j = \text{trig } a_j \text{ is a transcendental variable over } F_{j-1}\}; \text{ and}$$

$$H = \{j\colon \theta_j = \text{hyp } a_j \text{ is a transcendental variable over } F_{j-1}\}.$$

When F_m is an eplath field and T and H are empty, F_m is what Caviness and Rothstein call a generalized log-explicit field. If i is in F_m, the field G_m obtained by replacing $\theta = \text{trig } (a)$ with $\theta = \exp (ia)$ and replacing $\theta = \text{hyp } (a)$ with $\theta = \exp (a)$ will be a generalized log-explicit field containing F_m.

An eplath field F_m may not be a differential field. For example, let $F_1 = F_0(\sin z_1)$. Then, since $D_1(\sin z_1) = \cos z_1$, is not in F_1, F_1 is not a differential field. However, there is always an eplath field G which is an algebraic extension of F_m and a differential extension of F_0. One such G is $G = F_m(\psi_1, \cdots, \psi_m)$ where $\psi_j = \exp a_j$ for $j \in H$; $\psi_j = \sin a_j$ if $\theta_j = \cos a_j$ or $\theta_j = \sec a_j$ and $\psi_j = \cos a_j$ if $j \in T$ and neither $\theta_j = \cos a_j$ nor $\theta_j = \sec a_j$; $\psi_j = 1$ otherwise. If i is in F_0, it is possible to find a $G = F_0(\psi_1, \cdots, \psi_m)$ where each $F_0(\psi_1, \cdots, \psi_j)$ is a differential field.

Before stating the structure theorem for eplath fields it should be pointed out that eplath fields may contain elements which are not elementary functions. For example, $\operatorname{erf} z = \int \exp (z^2)\, dz$ is in the eplath field $Q(z, \exp (z^2), \operatorname{erf} z)$ but it is not an elementary function. However, an elementary field can always be considered an eplath field if the constant field K is suitably chosen.

THEOREM (Structure theorem for eplath fields). *Let $F_m = F_0(\theta_1, \theta_2, \cdots, \theta_m)$ be an eplath field. Suppose $i \in F_0$ and $a \in F_m$. Then*

(a) log *a is not a transcendental variable over F_m if and only if there are rational integers n_j, not all zero, and k in $C(F_m)$ such that*

$$a^{n_0} \prod_{j\in L} a_j^{n_j} \prod_{j\in E} \theta_j^{n_j} \prod_{j\in T} (\exp ia_j)^{n_j} \prod_{j\in H} (\exp a_j)^{n_j} = k; \tag{4}$$

(b) exp *a or* hyp *a is not a trascendental variable over F_m if and only if there are rational integers n_j, not all zero, and k in $C(F_m)$ such that*

$$n_0 a + \sum_{j\in E\cup H} n_j a_j + \sum_{j\in L} n_j \theta_j + \sum_{j\in T} n_j i a_j = k; \tag{5}$$

(c) trig *a is not a transcendental variable over F_m if and only if there are rational integers n_j, not all zero, and k in $C(F_m)$ such that*

$$n_0 ia + \sum_{j\in E\cup H} n_j a_j + \sum_{j\in L} n_j \theta_j + \sum_{j\in T} n_j i a_j = k. \tag{6}$$

Because the condition in part (a) of the theorem involves both the θ_j's (in which the a_j's are expressed) and the ψ_j's the condition stated in the following corollary may be preferable in applications.

COROLLARY 1. *Suppose $i \in F_0$ and $a \in F_m = F_0(\theta_1, \cdots, \theta_m)$ where F_m is an eplath field. Then* log *a is not a transcendental variable over F_m iff there are rational integers n_j, not all zero, such that for each differential operator D,*

$$0 = n_0 Da/a + \sum_{j\in L} n_j Da_j/a_j + \sum_{j\in E\cup H} n_j Da_j + \sum_{j\in T} n_j i Da_j. \tag{7}$$

Observe that the structure theorem and its corollary both require that i be an element of $C(F_m)$. This is to insure that ia_j be in $F_0(\psi_1, \cdots, \psi_{j-1})$ whenever $\theta_j = \text{trig } a_j$.

Thus, if no θ_j are both trigonometric and transcendental, the requirement that i be in F_m can be dropped provided θ is not trigonometric.

Although it appears that one could circumvent the requirement in the structure theorem that i be an element of F_0, by adjoining i to F_0, this has certain pitfalls. For example, suppose i is not in K, $F_0 = K(z)$ and $\theta_1 = \int dz/(z^2+1)$. Then $F_1 = F_0(\theta_1)$ is an eplath field since θ_1 is nonsimple over F_0. But, if i is adjoined to F_0, θ_1 is no longer nonsimple since, now, $2\theta_1 = i \log(z+i) - i \log(z-i) + c$ for some c in U. Thus, when a new constant is adjoined to F_m, F_m may cease to be eplath. However, this can happen only if there is some nonsimple, primitive, transcendental variable among the θ_j.

Thus, it has been shown that the structure theorem holds even when i is not in F_m provided that either (i) there is no nonsimple, primitive, and transcendental θ_j or (ii) there is no trigonometric θ_j and $\theta = \text{trig } a$ is not considered.

4. Representation and equivalence. The structure theorem can be used as a basis for finding regular representations for elements in an eplath field. This is done by first finding an eplath field containing a given expression and then determining its canonical representation relative to the θ's in that eplath field. Because an expression which is in an eplath field is in several eplath fields, the representation obtained is not entirely unique. It is unique relative to the choice of θ's in the eplath field used, however. Still, in order to determine whether two expressions are equivalent, it is sufficient to find an eplath field which contains both and see whether they have the same representation relative to this field.

The representation procedure is essentially inductive. Since canonical forms for rational functions are well known, suppose that $F_m = F_0(\theta_1, \cdots, \theta_m)$ is an eplath field containing i to whose elements the representation procedure can be applied. Consider the problem of finding regular representations for elements of $F_m(\theta)$. The following method is a generalization of algorithms in [1], which implement part of the Risch structure theorem and is an extension of the procedure in [7] for implementing the structure theorem in [8].

If θ is algebraic over F_m with $p(x)$ as its minimum polynomial, then elements of $F_m(\theta)$ are represented as elements of $F_m[\theta]$ modulo $p(\theta)$.

If $\theta = \exp a$, $\theta = \text{hyp } a$ or $\theta = \text{trig } a$ with $a \in F_m$, apply the structure theorem to determine whether θ is a transcendental variable over F_m. If it is, simply consider θ as another indeterminate for representation purposes. Otherwise, the structure theorem implies that θ is algebraic over $F_m(e^c)$ for some c in $C(F_m)$. Here serious difficulties can arise. For, it may be a difficult unsolved problem to find a unique representation for elements of $F_m(e^c)$ even for relatively simple values of c. For example, if $c = 1$ and $\pi \in F_m$, finding a unique representation for $e + \pi$ is complicated by the fact that it is not known whether $e + \pi$ is a rational number. If it is assumed that elements of $F_m(e^c)$ can be represented uniquely, since θ is algebraic over $F_m(e^c)$ the algebraic case now applies. Note that when e^c is algebraic over F_m, e^c need not be adjoined to F_m before returning to the algebraic case.

Suppose $\theta = \log a$, with $a \in F_m$. If θ is a transcendental variable over F_m, simply consider θ a new indeterminate. Either the theorem or the corollary can be used. Both will require solving systems of linear equations and may require representing elements in G_m (a generalized log-explicit field containing F_m) which are not in F_m. When applying the structure theorem, itself, a $\theta_j = \text{trig } a_j$ introduces $\exp(ia_j)$ which may not be in F_m. In applying the corollary, new elements can arise because F_m need not be closed under differentiation. In either event, $G_m = F_m(t)$ for some t algebraic over F_m since G_m is algebraic and finitely generated over F_m. So, elements in G_m can be

represented by applying the algebraic case; then, the structure theorem is applied to discover whether or not θ is a transcendental variable over F_m.

If θ is not a transcendental variable, the structure theorem implies that there are $c \in C(F_m)$ and rational number r such that $k = \log c + i\pi r$ and $\theta \in G_m(k)$. Here, there is a two-fold problem with constants. First, the value of k depends on the branches of the logarithm function chosen for each logarithmic θ_j and θ. Thus, the user of the procedure should probably be questioned before going further. Secondly, representing elements in $G_m(k)$ may involve difficult unsolved problems such as whether $\log 2 \cdot \log 3$ is rational. However, if the problems with constants can be resolved, θ is in $G_m(k)$.

Finally, suppose θ is primitive over F_m. If θ is a transcendental variable over F_m, θ can, again, be treated as another indeterminate for representation purposes. It follows from Liouville's theorem [4] that θ will be a transcendental variable over $G_m = F_m(t)$, and, therefore, also over F_m, if it is nonsimple over G_m. This can be determined by using either a modification of the Risch integration algorithm [5] and [6] or Rothstein's integration algorithm [7]. In either case, it will be necessary to do computation in G_m, so as in the logarithmic case, it may be necessary to adjoin an algebraic t to F_m in order to determine that θ is a transcendental variable. If θ is simple-logarithmic, either integration algorithm will find $\theta = \Sigma\, c_j \log v_j + w + k$, with $v_j,\ w \in G_m,\ c_j \in C(G_m)$ and $k \in C(U)$. If adjoining these logarithms to G_m as in the logarithmic case can be accomplished, and k can be adjoined to the resulting field, elements of $F_m(\theta)$ can now be represented.

Since each possibility for θ such that $F_m(\theta)$ is an eplath field has been considered, the induction step in the representation procedure is complete. Notice that if $F_m(\theta)$ is, indeed, an eplath field, it will have the same constant field as F_m. This is important because of the inductive nature of the representation procedure. For, once a new constant is adjoined to some F_m, the resulting field may fail to be an eplath field and the induction can no longer continue. For example, let $F_m = Q(z, \theta_1)$, where Q is the field of rational numbers, and $\theta_1 = \int dz/(z^2+2)$. If θ causes $\sqrt{2}$ to be adjoined the F_m, θ_1 is no longer nonsimple. Thus, if θ_1 is not replaced by logarithms, the structure theorem could no longer be used to determine whether there are algebraic relationships between θ_1, θ, and any further θ's.

5. Conclusions. The structure theorem presented in § 3 is a potentially powerful tool for doing algebraic simplification. The representation procedure based on it allows trigonometric and hyperbolic functions to be handled directly without an intermediate step in which these functions are first converted into equivalent expressions involving exponentials. This representation procedure is an important step in doing transcendental function arithmetic with trigonometric and hyperbolic functions. Furthermore, it offers the hope of developing algorithms for integration in finite terms of expressions involving trigonometric and hyperbolic functions without first expressing them in exponential terms.

However, at present the full power of this structure theorem cannot be exploited. For, doing so would require an algorithm for doing integration in finite terms for elements of an eplath field. While Rothstein [7] points out that such an algorithm exists, he claims it is impractical. He does, however, present an algorithm which can be applied to eplath fields in which there are no algebraic θ's. Risch [5], [6] on the other hand, presented an integration algorithm which can be applied to eplath fields in which there is no θ which is simultaneously primitive, nonsimple, and transcendental. Although it seems possible to generalize these algorithms to work in an eplath field, they would still have the drawback of requiring trigonometric and hyperbolic functions be converted

into exponential equivalents. Thus, this new structure theorem implies both the need for and the possibility of developing an improved integration algorithm.

REFERENCES

[1] H. I. EPSTEIN, *Algorithms for elementary transcendental function arithmetic*, Ph.D. Thesis, Univ. of Wisconsin, Madison, 1975.

[2] H. I. EPSTEIN AND B. F. CAVINESS, *A structure theorem for the elementary functions and its applications to the identity problem*, Internat. J. Comput. Information Sci., to appear.

[3] E. B. KOLCHIN, *Differential Algebra and Algebraic Groups*, Academic Press, New York, 1973.

[4] R. H. RISCH, *Further results on elementary functions*, RC2402, IBM Thomas J. Watson Res. Ctr., Yorktown Hts., NY, Mar. 1969.

[5] ———, *The problem of integration in finite terms*, Trans. Amer. Math. Soc., 139 (1969), pp. 167–189.

[6] ———, *The solution of the problem of integration in finite terms*, Bull. Amer. Math. Soc., 76 (1970), pp. 605–608.

[7] M. ROTHSTEIN, *Aspects of symbolic integration and simplification of exponential and primitive functions*, Ph.D. Thesis, Univ. of Wisconsin, Madison, 1976.

[8] M. ROTHSTEIN AND B. F. CAVINESS, *A structure theorem for exponential and primitive functions*, this Journal, to appear.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0005 $01.00/0

NEW ALGORITHMS FOR POLYNOMIAL MULTIPLICATION*

DOROTHEA A. KLIP†

Abstract. Exploiting the structure of the 2-dimensional sorting problem associated with the polynomial product has been the strategy in the design of certain algorithms which are faster for a large class of problems than those found in the literature. First a parallel is drawn between GEN–MULT and Horowitz's SORT–MULT algorithm [*A sorting algorithm for polynomial multiplication*, J. Assoc. Comput. Mach., 22 (1975), pp. 450–462]. The former owes its better performance to the global incorporation of the structure. A geometric picture of the relative location of the product exponents enabled us to present a new approach, implemented by the SLICE algorithms, in which the product exponents are sorted in separate parallel slices. For a class of problems of moderate size and sparsity, which was investigated by Johnson [*Sparse polynomial arithmetic*, Proc. Eurosam Conf., SIGSAM Bull., 8 (1974), pp. 63–71] with the ALTRAN system, the simple SLICE–S algorithm gave a 1:4 improvement, timewise and relative to sorting effort, as compared with ALTRAN's LI algorithm. The optimized version, for large, sparse problems, is based on theoretical arguments. It performs in linear time as an average with respect to its major operations.

Key words. polynomial multiplication, systems for symbolic manipulation, time complexity of $X + Y$ sorting

1. Introduction. The importance of efficient algorithms in the field of symbolic mathematics arises from the large storage requirements for mathematical expressions in general combined with the expense of manipulating them. In the case of the polynomial product one has to generate an expression whose size is the product of the size of each of the operands. It is the general consensus that symbolic entities can only be dealt with in an efficient way when adhering to a predefined canonical ordering for the indeterminates. However, since the algorithms presented in the literature [1], [6] require at best $O(mn \log n)$ for the sorting effort for operands containing m and n terms respectively, Gustavson and Yun [3] proposed the generation of a nonsorted product, which could be done in $O(mn)$ operations. We shall only consider algorithms for the generation of a sorted product.

Since order of magnitude is not the sole criterion for efficiency, attempts have been made to construct more efficient algorithms for certain problems which may frequently occur in practice. We shall present results of our comparative study of these algorithms.

When starting with ordered operands, the product matrix has a certain structure, which is a tableau in reverse order, as Horowitz [5] pointed out. His algorithm SORT–MULT, which requires $O(n^2 \log n)$ operations, will be compared with our algorithm GEN–MULT, which is also based on the tableau properties. GEN–MULT's better behavior for the test cases presented by Horowitz is due to the incorporation of the tableau properties in a global way.

However, ALTRAN's original algorithm and the improved HEAP version were in fact also based on the tableau properties. Difficulty in handling equal exponents led to the design of the List–Insertion or LI algorithm. Its better performance for a large class of problems is also based on the structure of the product matrix, as will be explained by us in § 5.

Being aware of the importance of exploiting the information contained in both operands, while at the same time realizing that GEN–MULT only exploits the structure of a *general* tableau, we looked for a scheme based on the minimal information, which

* Received by the editors May 23, 1978. This work was supported by PHS Program Project Grant No. HE11310 and a Medical Center Faculty Research Grant.

† Department of Physiology and Biophysics and Department of Computer and Information Sciences, University of Alabama, Birmingham, Alabama 35294.

would be a guideline for a more efficient approach. A geometric picture of the relative location of the product exponents was obtained, from which it was seen that the product terms can be sorted in separate parallel slices. The SLICE algorithms are based on this premise. For a large class of problems of moderate size, which were presented by Johnson [6], the simple SLICE–S algorithm gave a ratio 1:4 in performance as an average relative to the LI algorithm.

For large, sparse problems an analysis of the sparse random case was made, which resulted in an optimized version, projected in algorithm SLICE–O. The average complexity for the sorting procedure is $O(mn)$ for its major operations. The factor of proportionality is dependent on the number of equal exponents in the product and on environmental conditions.

From a theoretical point of view, SLICE–O could be modified to the extent that it works exactly in linear time, which means that the execution time is proportional with nm, at the expense of auxiliary storage of the same size as the $m \times n$ product. This approach has been recently implemented as the bucket-sort algorithm by Teer [14] in the FORMULAPASCAL system. The complexity of "$X + Y$ sorting", which is the term for the mathematical abstraction of the multiplication problem was studied by Harper, Payne, Savage and Straus [4]. Their main results were that any algorithm for binary sorting (taking for convenience $n = m$) cannot perform better than $n^2 \log_2 n$, when making only use of the tableau properties. This lower bound is sharp. When taking into account the special properties of the product matrix and calculating an upper bound for the number of order types in the product, assuming that X and Y are already sorted, the lower bound for the complexity of sorting $X + Y$ is $\lesssim 8n \log_2 n$. On the basis of this result Fredman [2] challenges us by means of a nonconstructive proof with the assertion that any $X + Y$ problem can be sorted in $O(n^2)$.

The test data presented were obtained on an IBM 370/158 computer with MVS operating system from FORTRAN coded programs (G1 compiler), with the aid of our VARLIST list processing system [10], [11]. Only univariate polynomials were considered with single word length integer coefficients, represented in distributive form [8], [9].

2. Notation.

Polynomial operands. In the univariate case the input polynomials are represented by

$$A(x) = \sum_{1 \leq j \leq n} a_j x^{\alpha_j} \quad \text{and} \quad B(x) = \sum_{1 \leq j \leq m} b_j x^{\beta_j}.$$

α_j and β_j are positive integers which are decreasing with increasing j; a_j and b_j are numerical coefficients.

pp-matrix. The $n \times m$ matrix of the exponent set $\{e_{ij}\}$ of the product terms $p_{ij} = a_i b_j x^{\alpha_i + \beta_j}$ $(i = 1, \cdots, n; j = 1, \cdots, m)$, $e_{ij} = \alpha_i + \beta_j$,

$$\begin{vmatrix} e_{11} & e_{12} & \cdots & e_{1m} \\ e_{21} & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ e_{n1} & \cdots & \cdots & e_{nm} \end{vmatrix}$$

FIG. 1. *The pp-matrix.*

is a special case of a tableau (see § 3.1). It is called, following Horowitz, the *pp*-matrix.

Covering set. A set of product terms p_{ij} which has the property that product terms not calculated yet have lower ordering than the largest of this set is called a "covering set" for the *pp*-matrix.

Structure number S. In case the exponent differences of the operands are random, the largest of these differences is called, following Johnson [6], the structure number for this particular random case.

PPL. This is the abbreviation for partial product list, which is an ordered subset of the product terms.

FINL. This is the abbreviation for final product list; the sorted terms which are no longer needed for the sorting of the remaining terms are deleted to FINL.

3. The algorithms GEN–MULT and SORT–MULT.

3.1. GEN–MULT. In early years of symbolic mathematics at our Institution, when computer memory was small, an algorithm was developed which was based on the properties of the matrix of the product exponents (Fig. 1). Horowitz [5] was the first to observe that this matrix is a tableau in reverse order; the integers in each row are decreasing with increasing row number, and the same property holds relative to the columns. His referral to this matrix as the *pp*-matrix will be adhered to in the sequel.

The algorithm GEN–MULT generates the terms from certain diagonal sections of the *pp*-matrix (see example in Fig. 2). One should bear in mind that this matrix is only a frame of reference as to the order in which product terms should be produced from the

1	2	3	5	6	8	9
120	116	97	90	75	63	56
2	4	5	7	8	9	10
112	108	89	82	67	60	48
4	6	7	9	9	11	11
95	91	72	65	50	43	31
6	6	9	9	11	11	13
90	86	67	60	45	38	26
6	7	9	10	11	12	14
77	73	54	47	32	25	13
7	8	10	11	12	14	16
74	70	51	44	29	22	10
8	10	10	12	13	15	17
64	60	41	34	19	12	0

FIG. 2. *pp-matrix used as a reference for the algorithm GEN–MULT. The terms calculated in a particular step are indicated by the corresponding step number in the upper right corner. The underlined elements are the reference terms for the next step.*

operands $A(x)$ and $B(x)$. At a certain stage in the process certain terms have been calculated and put in order. Realizing that the exponents form a tableau, the left upper corner represents those terms. Part of the calculated terms, no longer needed for the sorting process of the remaining terms, has been deleted to the final product list. Those which are needed for the proper continuation of the process form the so-called "covering set". They are the sorted terms of the linear partial product list PPL. In the next step PPL is updated relative to the term with lowest ordering, which is called the reference term, p_{ref}, for this step; the *pp*-matrix is scanned in increasing order of the rows, inserting terms on a particular row until the current term has lower or equal ordering than p_{ref}, or until the column number of its right neighbor equals that of a term, earlier processed in this step, which had lower or equal ordering relative to p_{ref}. As soon as updating is achieved, the top part of PPL up to the reference term is deleted to FINL.

The formal outline of the algorithm GEN–MULT, presented in Fig. 4, will be clarified by the example of the polynomials.

$$A(x) = x^{56} + x^{48} + x^{31} + x^{26} + x^{13} + x^{10} + 1,$$

$$B(x) = x^{64} + x^{60} + x^{41} + x^{34} + x^{19} + x^{12} + 1.$$

The *pp*-matrix associated with it, is pictured in Fig. 2.

Step	Start	End	Comparisons
1	120		0
2		116	
		112	1
3	112	112	
		97	1
4	97	108	1
		97	
		95	1
5	95	95	
		90	1
		89	2
6	90	91	2
	89	90	2
		89	
		86	2
		77	3
		75	1
7	86	86	
	77	82	3
	75	77	
		75	
		74	2
		73	2
		72	1
8	74	74	
	73	73	
	72	72	
		70	2
		68	1
		67	2
		64	4
9	70	70	
	68	68	
	67	67	3
	64	65	2
		64	
		60	2+2
		56	1
		54	4
		50	3
10	60	60	4
	56	56	
	54	54	
	50	51	2
		50	
		48	1
		47	2
		41	3
11	48	48	
	47	47	
	41	45	3
		44	3
		43	2
		41	
		38	2
		32	3
		31	1
12	38	38	
	32	34	3
	31	32	
		31	
		29	2
		25	1
13	29	29	
	25	26	2
		25	
		19	1
14	19	22	1
		19	
		13	1
15	13	13	
		12	1
16	12	12	1
		10	
17		0	0
Total comparisons:			90

FIG. 3. *Listing of steps for example in Fig. 2.*

The total number of exponent comparisons for sorting the total product was 90, as can be verified from the steplisting in Fig. 3. The algorithm was mentioned in [8] and was documented with respect to its major aspects in [7].

```
Procedure GEN-MULT:                                   init;
[m ≧ n > 2]                                           COL(i) ← 1; [i = 2, · · · , n]
[COL(i) is column number of term on row i to          delete (p11);
 be processed next]                                   enter (p12);
[calculate p11 and delete to FINL]                    CMIN ← 2;
[calculate p12 and put on PPL]                        COL(1) ← 3;
[CMIN is the reference column number;
 terms with j = CMIN do not have to be
 processed in current step]                           ITOP ← 1; e_ref ← e12;
[ITOP is the first row which has not been fully       i ← 2;
 processed]                                           while (terms are left) do;
                                                      while (i ≦ n and COL(i) < CMIN) do;
[main loop]                                           j ← COL(i); insert (p_ij);
[p_ij is calculated and inserted in PPL               if COL(i) = m then ITOP ← ITOP + 1;
                                                                          COL(i) ← COL(i) + 1;
                                                      trm ← e_ij;
                                                      if (trm ≧ e_ref or COL(i) = CMIN) then do;
[next row]                                              i ← i + 1;
                                                      if trm ≧ e_ref then CMIN ← j;          end;
[end main loop]                                       if (CMIN = 1 or i = n) then do;
[delete top part of PPL; generate produces i_ref      delete (TOP); generate (e_ref);
 and j_ref, row resp. column number of bottom         CMIN ← m + 1; i ← ITOP;
 term p_ref]                                          if i_ref = ITOP then do; CMIN ← j_ref;
                                                      i ← i + 1;          end;
                                                      else i ← i + 1;              end   end
                                                                        end
                                                      end GEN-MULT
```

FIG. 4. *Outline of GEN-MULT. Procedure* **insert** *inserts the term in PPL, starting at the reference term, going either upward or downward until its proper position is found.*

3.2. SORT-MULT, Horowitz's best algorithm for polynomial sorting. In this algorithm [5, p. 458] a set of product terms is generated (and maintained as a binary tree structure) with the property that at most one term from each row of the *pp*-matrix is represented; if the column number of a possible candidate in row i is represented on the list (by a term with row $i' < i$) then this term will not be inserted in this step. Consequently this set is called a minimal covering set relative to the term with highest ordering. When updating is completed, the term with highest ordering is deleted and appended to the final product. Insertion of a candidate into the tree occurs from a point which is randomly selected.

In Table 1 (columns 2 and 4) the exponent comparisons are listed for the most relevant "random case," which is defined as follows: the exponents α_j and β_j of the polynomials $A(x)$ and $B(x)$ are generated according to the rule

$$\alpha_n = 0; \qquad \alpha_{n-j} = \alpha_{n-j+1} + \text{RAND}(20) \qquad (j = 1, \cdots, n-1)$$

$$\beta_m = 0; \qquad \beta_{m-j} = \beta_{m-j+1} + \text{RAND}(20) \qquad (j = 1, \cdots, m-1)$$

where RAND(20) is an integer randomly selected from the set $\{1, 2, \cdots, 20\}$. The structure number S is 20 in this case.

TABLE 1

Comparison of Horowitz's SORT-MULT *with* GEN-MULT *and* SLICE-S, *for the random case with structure number* 20, *for polynomial operands with number of terms* $n = m$. *The values in the second column are taken from* [5]. *Although* SLICE-S *is more efficient,* GEN-MULT *will find application in case of multivariate operands with a large number of indeterminates, because* SLICE-S *requires isomorphic conversion of the exponent sets to their univariate images.*

	SORT-MULT	GEN-MULT		SLICE-S
n	Exponent comparisons	Steps	Exponent comparisons	Exponent comparisons
3	11	4	5	0
4	33	7	15	1
5	53	10	31	3
6	107	12	50	7
7	157	14	79	16
8	221	16	114	28
9	264	18	159	42
10	387	21	208	58
11	554	23	273	82
12	668	25	343	110
13	757	26	413	141
14	1053	27	508	179
15	1118	29	600	213
16	1409	31	720	251
17	1596	32	855	299
18	1643	34	951	364
19	2289	36	1103	394
20	2031	38	1268	489

3.3. Comparison of GEN-MULT and SORT-MULT for Horowitz's input polynomials. The test data which Horowitz presented [5, p. 461] were classified as "dense," "sparse", "intermediate" and "random". However, the distinction "structured" and "unstructured" would have been more meaningful as a basis for comparison of performance. "Structured" could be defined as a regular pattern for the exponent sequences of either operand. Any algorithm that takes advantage of the structure of the problem is bound to perform better than algorithms which are not designed on this basis. This fact becomes clear when inspecting Horowitz's test data for his SORT-MULT algorithm against conventional methods, which have the same upper bound $O(mn \log n)$ for the sorting effort.

GEN-MULT is an improvement over SORT-MULT due to the global incorporation of the structure. For the "random case" with $S = 20$ an average ratio 0.52 was found in favor of GEN-MULT as can be verified by inspecting the data in columns 2 and 4 of Table 1. Although the linear insertion gives an a priori $O(mn^2)$ bound for the sorting effort of GEN-MULT, as opposed to SORT-MULT's bound $O(nm \log n)$ due to the binary search strategy, a consistent increase in ratio with increasing n was not observed for the low values of n in the test data. With the availability of the SLICE algorithms, the application of GEN-MULT will be restricted to special multivariate problems in which n and m are of moderate size. In § 6 a review is given.

The characteristics and differences of the algorithms are listed below.

GEN–MULT	SORT–MULT
I. CHARACTERISTICS	
1. A partial product list, maintaining a linear list structure, is updated by inserting terms from the *pp*-matrix, relative to the term which is *at the bottom* of the list after completion of the previous step.	1. A partial product list, in the form of a binary tree, is updated *relative to the top term* until each row of the *pp*-matrix either has a representative, or if it is known that the current candidate has lower ordering than the top term.
2. A step terminates by deleting the top part of the partial product list up to the reference term.	2. A step terminates by deleting the top term from the list.
II. DIFFERENCES	
1. The algorithm incorporates dynamically *the global structure* of the *pp*-matrix, since it may occur that several closely spaced terms on a particular row are inserted in one step.	1. The algorithm looks for the largest exponent of a minimal set, which means that only *the local structure* of the *pp*-matrix is taken into account.
2. The number of steps is $O(n+m)$.	2. The number of steps ranges from $O(n+m)$ for the dense case to $O(nm)$ for the sparse case.
3. Although theoretically the same upperbound $O(mn \log_2 n)$ holds for the number of exponent comparisons when the partial product list would be maintained as a tree structure, the cheaper linear list structure proved sufficiently adequate to bring out GEN–MULT's superior performance for the test cases presented by Horowitz.	3. The number of exponent comparisons was proven to be $O(mn \log_2 n)$.
4. The partial product list contains $O(2n)$ terms.	4. The partial product list contains a number of terms which is $\leqq n$.

4. The algorithms SLICE–S and SLICE–O.

4.1. The 2-dimensional picture of the relative location of the product exponents. We have seen in § 2 that GEN–MULT exploits the properties of a general tableau. However, the *pp*-matrix has additional properties, namely the difference of two elements in the same row only depends on their column numbers; a similar statement holds for 2 elements in the same column. In formula:

$$e_{ij} - e_{ik} = \alpha_i + \beta_j - \alpha_i - \beta_k = \beta_j - \beta_k,$$

$$e_{ij} - e_{kj} = \alpha_i + \beta_j - \alpha_k - \beta_j = \alpha_i - \alpha_k.$$

This means that we do not need all the information contained in the *pp*-matrix; the order of the product terms is completely defined by the two linear arrays v_{1j} and v_{j1} of the exponent differences of each of the operands. If one defines:

$$v_{11} = 0; \qquad v_{1j} = e_{11} - e_{1j} \quad (j = 2, \cdots, m); \qquad v_{i1} = e_{11} - e_{i1} \quad (i = 2, \cdots, n)$$

it follows $v_{1i} - v_{1j} = \beta_j - \beta_i$; $v_{i1} - v_{j1} = \alpha_j - \alpha_i$.

The exponent differences v_{i1} and v_{1j} are plotted on the axes ξ and η of a Cartesian coordinate system, such that $\xi_{i-1} \equiv v_{i1}$ $(i = 2, \cdots, n)$, $\eta_{j-1} \equiv v_{1j}$ $(j = 2, \cdots, m)$. When assigning to the points v_{ij} (with coordinates ξ_{i-1}, η_{j-1}) the norm $|v_{ij}| = \xi_{i-1} + \eta_{j-1}$, then these points can be considered the images of the product exponents in the sense that the following relationships are valid:

1. $e_{ij} - e_{kl} = -|v_{ij}| + |v_{kl}|$;
2. $e_{ij} = e_{kl}$ iff $\xi_{i-1} + \eta_{j-1} = \xi_{k-1} + \eta_{l-1}$, i.e. if the corresponding points v_{ij}, v_{kl} are located on the straight line $\xi + \eta = c$, such that $c = \xi_{i-1} + \eta_{j-1}$.
3. $e_{ij} > e_{kl}$ iff $\xi_{i-1} + \eta_{j-1} < \xi_{k-1} + \eta_{l-1}$.

These formal results can be stated in the following simple manner:

When moving a rod from the origin to the opposite end of the rectangle under equal angles with the coordinate axes one encounters the images of the product exponents in their correct ordering.

4.2. Algorithm SLICE–S. In a first intuitive approach, presented by algorithm SLICE–S, the rectangle is divided in (at most) $2(n+m)-6$ slices by choosing the set of parallel lines

$$\xi + \eta = v_{j1} \quad (j = 2, \cdots, n); \qquad \xi + \eta = v_{1j} \quad (j = 2, \cdots, m);$$

$$\xi + \eta = v_{n1} + v_{1j} \quad (j = 2, \cdots, m-1);$$

$$\xi + \eta = v_{1m} + v_{j1} \quad (j = 2, \cdots, n-1)$$

as shown in Fig. 5. The marked reduction in number of symbolic comparisons is obtained at the expense of a number of numerical comparisons for each element whether or not it is located in the current slice. This implies that the linear arrays v_{1j} and v_{j1} actually have to be constructed as the initializing step.

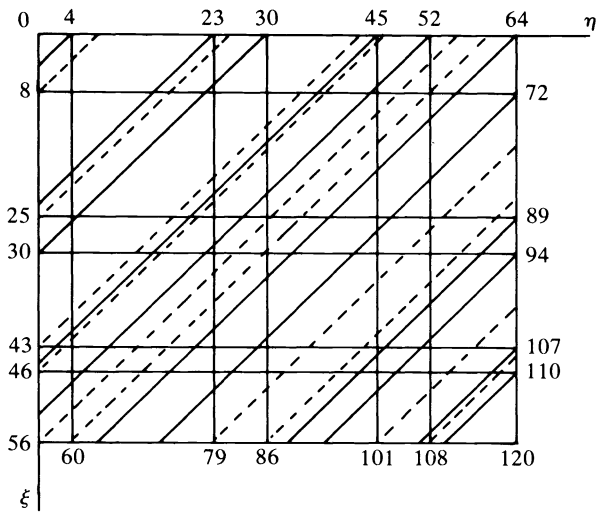


FIG. 5. *The pp-matrix of Fig. 2, pictured relative to the value of the exponents. The points on the coordinate axes represent the partial sums of the exponent differences of the polynomials $B(x)$ and $A(x)$ from the example. Only the internal points of the rectangle which are located in one "slice" (section between adjacent parallel lines) have to be put in order on PPL. Following procedure PUTLIST one can verify that* 14 *symbolic exponent comparisons are required, when comparison starts from the bottom of PPL. It was earlier found that the same example requires* 90 *comparisons for GEN–MULT.*

In Table 1 the symbolic comparisons for SLICE–S for Horowitz's input polynomials for the random case are listed in the 4th column.

4.3. Optimization for the sparse, random case implemented in algorithm SLICE–O.

4.3.1. Division of the rectangle. In an attempt to find the optimal approach, we present the following lemmas.

LEMMA 1. *The division of the rectangle in slices of equal area will ensure minimization of the symbolic sorting effort, when averages are taken over a large number of tests.*

Proof. The points of the rectangular network are distributed with uniform density, when averages are taken over a large number of tests, in which n, m and S are kept constant. Equal partitioning of the rectangle as a means for optimization is a direct consequence of a simple calculation from the calculus of variations; since the sorting for the jth section, containing x_j terms, requires $O(x_j^2)$ (in case of a linear list structure), the sum $S = \sum_{1 \leqq j \leqq T} x_j^2$ has to be minimized under the constraint $\sum_{1 \leqq j \leqq T} x_j = \text{constant}$ $(= nm)$. It is readily found that $x_1 = x_2 = \cdots = x_T$, in which T is the total number of sections.

4.3.2. The rigorous approach. Once the constant of proportionality c in the relationship $T = cmn$ for a particular computing environment is obtained (see § 4.3.6), the optimal value for T can be calculated. The division of the rectangle in slices of equal area then involves the following calculations. Referring to the notation of the outline in Fig. 6, we define

$$a = VA(n) \quad (\text{degree of } A(x)); \qquad b = VB(m) \quad (\text{degree of } B(x))$$

and assume that $b \geqq a$. Dividing the rectangle into two congruent triangles and the parallelogram which they enclose, we introduce

```
Procedure SLICE–S:
[choose m ≧ n > 2]
[partial sums of exp. diff. of A(x)]      VA(1) ← 0; VA(i) ← v_i1; [i = 2, ···, n]
[partial sums of exp. diff. of B(x)]      VB(1) ← 0; VB(j) ← v_1j; [j = 2, ···, m]
[column number of next term on row        COL(i) ← 2; [i = 2, ···, n − 1]
 i]
[main loop in PUT–LIST starts with        I ← 2;
 row I]
 [p_11 has highest ordering]              delete (p_11); i ← 2; j ← 2;
                                          init: begin;
                                          while VA(2) > VB(j) do; delete(p_1j); j ← j + 1;  end;
[since e_22 < e_12 and e_22 < e_21,       while VA(i) < VB(2) do; delete(p_i1); i ← i + 1;  end;
 initialization comprises deletion of     if VA(i) < VB(j) then do; delete(p_i1); i ← i + 1;  end;
 terms p_1j and p_i1 up to                else do;
 min (p_12, p_21)]                        if VA(i) > VB(j) then do; delete(p_1j); j ← j + 1;  end;
                                          if VA(i) = VB(j) then do; TERM = sum(p_i1, p_1j);
                                                   delete(TERM); i ← i + 1; j ← j + 1;
                                              end;                end;
                                                end;
[TERM1 and TERM2 are terms at             V1 ← VA(i); TERM1 ← p_i1; V2 ← VB(j); TERM2 ← p_1j;
 the boundary; side1 = true if V1         side1 ← true; side2 ← true; DIAG ← 0;
 moves along left vertical boundary;      main loop:
 side2 = true if V2 moves along           while
 upper horizontal boundary]                 DIAG ≧ min(VA(n) + VB(m − 1), VB(m) + VA(n − 1))   do
[DIAG = min(V1,V2). At start V1           if V1 < V2 then do; DIAG ← V1; TERM ← TERM1; end;
 moves along left vertical boundary,              else do; DIAG ← V2; TERM ← TERM2;  end;
 V2 along upper horizontal                if V1 = V2 then TERM ← sum(TERM1, TERM2);
 boundary]                                put-list(DIAG); append(TERM);
                                          delete (PPL);
```

```
[terms above the line ξ×η = DIAG                if(V1 ≦ V2 and side1 = true then do;
 are inserted in PPL; after the terms           if i < n then do; i ← i+1; V1 ← VA(i); TERM1 ← p_i1;
 on DIAG have been contracted and                        end; else do; side 1 ← false; i ← 2;
 appended to PPL, PPL is appended                              end;          end;
 to FINL]                                       if(V1 ≧ V2 and side2 = true) then do;
[next value for V1]                             if j < m then do; j ← j+1; V2 ← VB(j); TERM2 ← p_1j;
[initialize moving along lower                                 end; else do; side2 ← false; j ← 2;
 horizontal boundary]                                          end;          end;
[next value for V2]                             if(V1 ≦ V2 and side1 = false) then do;
[initialize moving along right vertical         V1 ← VA(n) + VB(i); TERM1 ← p_ni; i ← i+1;
 boundary]                                                     end;
                                                if(V1 ≧ V2 and side2 = false) then do;
                                                V2 ← VB(m) + VA(j); TERM2 ← p_jm; j ← j+1;
                                                              end;
                                                delete(remaining terms);
                                                end SLICE-S;                                        end;

Procedure PUT-LIST:

                                                init;
[THRU is true if all terms above                THRU ← false; THR(i) ← false; [i = I, ···, n−1]
 DIAG have been processed;                      while THRU = false do; THRU ← true; i ← I; CMIN ← m;
THR(i) is true if next term on row              main loop:
 i is below DIAG]                               while(i ≦ n−1 or CMIN > 2) do;
                                                if THR(i) = false then    do; j ← COL(i);
[in each cycle only one term per row            if v_ij < DIAG then do; insert(p_ij);
 is inserted, because diagonally                  if COL(i) = m−1 then I ← I+1;
 located terms tend to be closer                  else do; COL(i) ← COL(i)+1;
 spaced than terms on the same row]                       v_ij ← VA(i) + VB(j); THRU ← false;
                                                      end;        end;
                                                if v_ij = DIAG then TERM ← sum(TERM, p_ij);
                                                if v_ij ≧ DIAG then do; CMIN ← j; THR(i) ← true;
                                                                  end; end; i ← i+1;
                                                                          end
                                                            end;
                                                end PUT-LIST;
```

FIG. 6. *Outline of procedure SLICE–S with subprocedure PUT–LIST.*

S_{tr} = the number of slices of each triangle,

S_p = the number of slices of the parallelogram.

S_{tr} and S_p can be solved from $T = 2S_{tr} + S_p$ (total number of slices) and $0.5a^2/S_{tr} = (b-a)a/S_p$ (equal area for trapezoidal and parallelogrammic slices). It follows $S_{tr} = 0.5Ta/b$; $S_p = T(b-a)/b$. The increments of DIAG (see outline in Fig. 6) in the triangular sections are dependent on the slice number and are denoted by s_j ($j = 1, \cdots, S_{tr}$). With the notation $U \equiv a/S_{tr}^{0.5}$ it is found after a simple calculation

$$s_j = U(j^{0.5} - (j-1)^{0.5}) = U/(j^{0.5} + (j-1)^{0.5}) \qquad (j = 1, \cdots, S_{tr}). \tag{1}$$

Provided that $S_p \geqq 1$, the equal increments s_p of the parallelogrammic area are b/T.

4.3.3. Relaxing the condition (1). For each individual random case the network vertices are nonuniformly distributed over the rectangular area. The implementation of (1) would not be practical, due to the great amount of square root calculations. By means of the following considerations we virtually eliminate the cumbersome computations, but are globally upholding the requirement of equal area for the separate slices. It is first noticed in Lemma 2.

```
Procedure SLICE-O;

                                   a ← min(VA(n), VB(m));
                                   b ← max(VA(n), VB(m));
[S_tr is number of slices for triangle]
                                   T ← total number of slices;
[I_m is number of main intervals of equal length
 M_inc, the main increment, which is the length
 of the interval between the j²th and (j+1)²-
 th diagonal]                      S_tr ← 0.5 * T * a/b;
                                   I_m ← s_tr^0.5; M_inc ← a/I_m;
                                   first triangle:
                                   DIAG ← M_inc; put-list(DIAG);
                                   DIAG ← DIAG + 0.5 * M_inc; put-list(DIAG);
[each main division point i (=j²) is the center
 of 2i subintervals of length M_inc/(2j); these
 are the subincrements S_inc]      STEPS ← I_m − 1;
                                   main loop:
                                   for j ← 2 to STEPS do;
                                   j' ← 2j; S_inc ← M_inc/j';
                                        for k ← 1 to j'   do; DIAG ← DIAG + S_inc;
                                        put-list(DIAG); end;
                                                  end;
                                   remaining slices:
[last increment L_inc equals M_inc divided by
 2*I_m. These increments are added to DIAG,
 starting at the center of the I_mth interval until
 a total length a]                 L_inc ← M_inc/(2 * I_m);
                                   STEPS ← I_m + (a − I_m * M_inc)/L_inc;
                                   for k ← 1 to STEPS do;
                                   DIAG ← DIAG + L_inc; put-list(DIAG):
                                                  end;
                                   parallelogram:
                                   if a = b then goto last-triangle;
[S_p is number of slices in parallelogram]
[s_p are equal increments]         S_p ← T(b − a)/b; s_p ← b/T;
                                   if (S_p ≦ 1 or s_p = 0) then go to next-vertex;
                                   for k = 1 to S_p do;
                                   DIAG ← DIAG + s_p; put-list(DIAG);
                                                  end;
                                   next vertex: DIAG ← b; put-list(DIAG):
[algorithm for second triangle equals that of
 first triangle in reverse order]  last-triangle:

                                   end SLICE-O;
```

FIG. 7. *Outline of the main aspects of algorithm SLICE–O. For greater clarity we have omitted the obvious actions at each step, i.e. initializing TERM (=terms located on DIAG; see PUT–LIST, Fig. 6) as an empty list and, after insertion is completed, appending TERM to PPL. Then PPL is deleted to FINL.*

LEMMA 2. *The sum of the increments between the j^2-th and the $(j+1)^2$-th slice is independent of j and thus equals $U \equiv a/S_{tr}^{0.5}$.*

Proof. It follows from (1) that

$$\sum_{j^2+1 \leqq i \leqq (j+1)^2} U(i^{0.5} - (i-1)^{0.5}) = U(-j + (j+1)) = U.$$

Consequently, after dividing the triangle side in $S_{tr}^{0.5}$ equal main intervals U, we now take $U/(2j)$ for the slice width around the jth main interval. Since $j^2 = i$, where i numbers the subintervals, the exact slice width $U/(i^{0.5} + (i-1)^{0.5})$ at the jth main division point is approximated by $U/(2i^{0.5})$. Thus, instead of dividing the trapezoidal sections between the j^2 and $(j+1)^2$ diagonals into $2j+1$ sections of *equal area*, this section is divided into j sections of *equal slice length* $U/(2j)$ and $j+1$ sections of *equal slice length* $U/(2j+2)$.

4.3.4. Outline of algorithm SLICE–O. The simplified version of the rigorous approach, discussed above, is outlined in Fig. 7.

4.3.5. Order of magnitude for the sorting effort.
The essential operations.

LEMMA 3. *Asymptotic optimal performance of SLICE–O for the sparse random case, for its major operations, occurs when the total number of slices T is chosen proportional with mn; T = cmn, where c depends on the degree of sparseness and on environmental conditions. This implies that the algorithm performs in linear time and requires O(mn) for the symbolic sorting effort.*

Proof. The major operations of the algorithm are

1. The numeric operations for calculating DIAG, which are $O(T)$.
2. The symbolic sorting effort per term for the terms of each slice, which, in our case of linear insertion, requires $O(nm/T)$. Accordingly, the total effort is $O(n^2m^2/T)$.
3. Concatenation of the sorted PPL's which requires $O(T)$ operations.

It follows that if and only if T is chosen proportional with nm, the asymptotic behavior of the algorithm is $O(mn)$.

Consequently, the total symbolic sorting effort is $O(mn)$ and the total computing time is proportional with mn.

The minor operations. An operation not included in the proof above is the assignment of the correct slice to each product term. It is considered a minor operation in the area of symbol manipulation due to its low weight factor. In the outline of algorithm PUT–LIST (Fig. 6) the value v_{ij} (see § 4.1), corresponding with a candidate product term p_{ij}, has to be compared with the slice boundary value DIAG. Since $T = O(mn)$, while only m terms in each row are placed in a slice, the number of vain comparisons per term is $O(n)$. This accounts for the nonlinearity of the total procedure. Other minor operations in PUT–LIST have the same bound.

Theoretical improvement of the term insertion. The term insertion could be reduced to linear behavior when generating the T PPL's simultaneously and calculating for each term its appropriate slice. The linearity achieved in this manner would be at the cost of auxiliary memory space, which in our system, for $n = m = 300$, would be approximately 180K bytes.

An intermediate approach, which requires approximately $\frac{1}{10}$ of the auxiliary space mentioned above for the same example, would be to generate the $T^{0.5}$ major PPL's (with equal slice width (see § 4.3.3)) sequentially, but generating the $2j$ minor PPL's within each major slice simultaneously. With 1 term per row as an average located in a major slice, the insertion of this term in the correct minor slice would be bounded by $\log_2 n$, when applying binary search.

4.3.6. Experimental results.

Tests with algorithm SLICE–O. Tests with SLICE–O were done for $n = m$, with values 50, 70, 90, 150, 200, 300 and S-values 64 and 1000. The definition of the structure number S was given in § 2 and § 3.2. 64 was the largest value for S employed in Johnson's experiments. From observation it was found that for $S = 64$ approximately $\frac{2}{3}$ of the nm product terms are distinct. $S = 1000$ corresponded with 98% sparsity. Therefore this value for S was used to generate operands which were good representations of the completely sparse random case. In Table 2 are displayed the symbolic comparison effort per term, the value of c (see § 4.3.5) which gives the ratio of the optimal number of slices T versus n^2. The observed minimal time divided by n^2 is listed in the last column. The data listed for optimal T were derived from a great number of tests over a certain range of T values, such that a significant increase in time was observed at both ends of the spectrum.

With respect to the data, the following comments are made:

1. The uncertainty in symbolic sorting effort (which is closely related to that in

TABLE 2

Data obtained with algorithm SLICE–O for S-*values* 64; 1,000; 10,000. *In the first column the value of* $n(=m)$ *is displayed. The number of symbolic sorting operations per term is shown in the* 2*nd column. From the factor of proportionality c the optimal value of T can be derived. The data in the* 4*th column show that the algorithm exhibits near linear behavior.*

S = 64				S = 1,000			
n	Symb. sort/n^2 ±10%	$c = T/n^2$ ±10%	Time/n^2 (millisec) ±2%	n	Symb. sort/n^2 ±10%	$c = T/n^2$ ±10%	Time/n^2 (millisec) ±2%
50	2.2	0.12	2.5	50	2.3	0.16	3.1
70	2.7	0.08	2.5	70	2.6	0.12	3.1
90	2.8	0.06	2.4	90	3.0	0.10	3.1
150	2.8	0.05	2.5	150	3.2	0.08	3.3
200	2.9	0.05	2.4	200	3.3	0.08	3.4
300	3.3	0.05	2.6	300	3.5	0.07	3.5

S = 10,000			
n	Symb. sort/n^2 ±10%	$c = T/n^2$ ±10%	Time/n^2 (millisec) ±2%
150	3.1	0.10	3.3

T/n^2) is much larger than the uncertainty in computing time. This is due to the low weight factors of the operations which are linearly dependent on T as specified in § 4.3.5. In mathematical terms this can be seen as follows. When defining $y :=$ total effort$/n^2$, $x := T/n^2$, we find $y = 1/x + (c_2 + c_3 n)x/c_1 + \cdots$ (terms of lower order in x) where c_1 and c_2 are the weight factors for the operation 2 and $1+3$ respectively c_3 is the weight factor for term comparison operations.

The significant part of this equation represents the branch of a nonorthogonal hyperbola in the first quadrant with asymptotes $x = 0$ and $c_1 y = (c_2 + c_3 n)x$. Due to the small ratio $(c_2 + c_3 n)/c_1$, the slope of this line is small. From this it follows that the x-value, corresponding with minimal y, is not sharply defined.

The uncertainty in time is mainly due to fluctuations in observed CPU time for identical runs, which is inherent with the MVS operating system (for the IBM 370/158).

2. The effect of the nonlinear term on time is not noticeable in the case $S = 64$. In the completely sparse case there is an increase of approximately 10% in time. Since it is most likely that the range in problem size of our test runs covers all cases occurring in the field of symbolic computation, we did not feel the need to implement one of the alternative approaches mentioned in § 4.3.5.

3. The ratio 2.5/3.3 in time of the cases characterized by $S = 64$ and $S = 1000$ is in close agreement with the observed ratio 2/3 for the number of distinct terms versus the size of the total product; the slightly larger ratio 2.5/3.3 is due to the fact that equal exponents of terms on an unsorted PPL will in general not occur sequentially.

4. In order to verify that the sorting effort is only dependent on the degree of sparsity, we listed at the bottom of Table 2 the results of a few tests with $S = 10^4$, for $n = 150$.

5. Due to the low average number of terms per slice, which is $1/c$ (see Table 2), we do not expect great advantage in time when employing a different list structure.

Comparative data for SLICE–S and SLICE–O. More extensive data for SLICE–S will be presented in § 5.4, after discussion of the LI algorithm. The disadvantage of SLICE–O is that an a priori guess of the optimal number of slices has to be made, which means that the system constant c has to be determined. The simple SLICE–S algorithm will cover a large class of problems without substantial sacrifice in computing time, as can be seen from the data in Table 3.

TABLE 3

Comparison of algorithms SLICE–S and SLICE–O. SLICE–O becomes significantly more profitable for the completely sparse case and large values for the number of terms of the polynomial operands.

	SLICE–S		SLICE–O	
	$S = 64$			
n	Symb. sort/n^2	Time/n^2 (millisec)	Symb. sort/n^2	Time/n^2 (millisec)
50	3.5	2.7	2.2	2.5
70	4.1	2.7	2.7	2.5
90	4.6	2.8	2.8	2.4
	$S = 1{,}000$			
50	4.8	3.9	2.3	3.1
70	6.4	4.3	2.6	3.1
90	7.8	4.6	3.0	3.1

4.3.7. The multivariate case. Construction of the linear arrays of the exponent differences in the SLICE procedures requires that in the multivariate case ($\nu \geqq 2$) the exponent sets

$$\{\varepsilon_{1j}^{(1)}, \varepsilon_{2j}^{(1)}, \cdots, \varepsilon_{\nu j}^{(1)} | j = 1, \cdots, n\} \text{ of polynomial } A(x),$$

$$\{\varepsilon_{1j}^{(2)}, \varepsilon_{2j}^{(2)}, \cdots, \varepsilon_{\nu j}^{(2)} | j = 1, \cdots, m\} \text{ of polynomial } B(x)$$

have to be mapped into the ring of the integers such that the order of the product terms produced from the univariate images uniquely defines the order of the multivariate product.

It is emphasized however that the SLICE algorithms only require knowledge of the exponent sets

$$\{\alpha_j^{(1)} | j = 1, \cdots, n\} \quad \text{and} \quad \{\alpha_j^{(2)} | j = 1, \cdots, m\}$$

of these images, since sorting in each slice is performed on the given multivariate operands. (The alternative approach, which requires inverse mapping of the univariate product, has been presented in a different context [13].)

Order of the sets $\{t^{(i)}\}$ of the terms of either operand is defined in the usual way by assigning a certain canonical ordering to the variables, say, $x_1 > x_2 > \cdots > x_\nu$. If

$$t_1^{(i)} := c_1^{(i)} x_1^{\varepsilon_{11}} x_2^{\varepsilon_{21}} \cdots x_\nu^{\varepsilon_{\nu 1}}, \qquad t_2^{(i)} := c_2^{(i)} x_1^{\varepsilon_{12}} x_2^{\varepsilon_{22}} \cdots x_\nu^{\varepsilon_{\nu 2}},$$

then by definition $t_1^{(i)} > t_2^{(i)}$ if and only if one of the following condition holds

(1) $\varepsilon_{11} > \varepsilon_{12}$,

(2) $\varepsilon_{i1} = \varepsilon_{i2} \quad (i = 1, \cdots, j; j < \nu)$ and $\varepsilon_{i+1,1} > \varepsilon_{i+1,2}$.

(If a variable does not explicitly appear on a term, then its exponent is interpreted as 0.)

The isomorphic mapping is achieved as follows.

First the operators $\{\rho_j | j = 2, \cdots, \nu\}$ for the mapping operation are defined:

$$\rho_j := \sum_{1 \leqq i \leqq 2} \left(\max_{1 \leqq k \leqq n'} (\varepsilon_{jk}^{(i)}) - \min_{1 \leqq k \leqq n'} (\varepsilon_{jk}^{(i)}) \right) + 1 \qquad \left(n' = \begin{cases} n & \text{if } i = 1 \\ m & \text{if } i = 2 \end{cases} \right)$$

where the maxima (and minima in case of the occurrence of negative exponents (see e.g. [8, p. 3])) for the jth variable are computed for the terms of $A(x)$ and $B(x)$ respectively.

Then the recursive generation of the integers $\beta_{kj}^{(i)}$ according to the scheme

$$\beta_{1j}^{(i)} = \varepsilon_{1j}^{(i)}; \qquad \beta_{kj}^{(i)} = \varepsilon_{kj}^{(i)} + \rho_k \beta_{k-1,j}^{(i)} \qquad (k = 2, \cdots, \nu)$$

for each term $t_j^{(i)}$ will yield the sets $\{a_j^{(i)}\}$, with $\alpha_j^{(i)} := \beta_{\nu j}^{(i)}$, and will guarantee the required isomorphism.

5. ALTRAN's List-Insertion or LI algorithm. The advantage of the LI algorithm for problems of moderate size over alternative methods in the ALTRAN system is due to the exploitation of the structure of the partial product list, although this fact has not been clearly recognized.

5.1. Basic idea. A certain linear ordered partial product list is maintained such that all terms not processed yet have lower ordering than the current top term. This list of terms can be considered a "covering set". This property is preserved, if, after deletion of the top term, say p_{ij}, the "neighbors" $p_{i,j+1}$ and $p_{i+1,j}$ are processed in case the following conditions are fulfilled:

1. these terms have not been processed earlier;
2. the terms $p_{i-1,j+1}$ and $p_{i+1,j-1}$ have earlier been processed.

Although the LI algorithm with its linear list structure has an a priori $O(n^2 m)$ behavior as compared with $O(nm \log_2 n)$ for the HEAP algorithm [1], [6], [12] with respect to the number of exponent comparisons, LI performed considerably better than HEAP for virtually all cases which are likely to occur in applications.

5.2. Explanation of LI's superior performance. We were able to show that the effect of two lemmas are responsible for LI's better performance.

LEMMA 4. *If $e_{ij} > e_{kl}$ then the probability that $e_{i,j+1} > e_{k,l+1}$ is greater than $\frac{1}{2}$.*

Proof. $e_{i,j+1} - e_{k,l+1} = \alpha_i - \alpha_k + \beta_{j+1} - \beta_{l+1} = (\alpha_i + \beta_j) - (\alpha_k + \beta_l) + (\beta_{j+1} - \beta_j) - (\beta_{l+1} - \beta_l) = (e_{ij} - e_{kl}) + (\beta_{j+1} - \beta_j) - (\beta_{l+1} - \beta_l)$. Since $e_{ij} - e_{kl} > 0$, whereas, in the random case, the remaining expression has equal probability to be >0 or <0, we conclude that $e_{i,j+1} - e_{k,l+1} > 0$ in the majority of cases.

One can express Lemma 4 in the following way: if one generates the set of right "neighbors" of the terms of a covering set, then their ordering will resemble the ordering of the former set.

LEMMA 5. *If p_{ij} and p_{kl} are terms of a covering set, such that $p_{ij} > p_{kl}$ then, if $p_{i,j+1}$ is inserted prior to $p_{k,l+1}$, the number of unsuccessful exponent comparisons is minimal if insertion starts from the bottom of PPL.*

Proof. Since, according to Lemma 4, the probability that $e_{i,j+1} > e_{k,l+1}$ is greater than $\frac{1}{2}$, no unsuccessful comparison of the corresponding terms $p_{i,j+1}$, $p_{k,l+1}$ is done in the majority of cases, when processing $p_{i,j+1}$ prior to $p_{k,l+1}$, while starting the exponent comparisons from the bottom of PPL. It is further noticed that the terms of a "covering set" are in general closer spaced than a term and its neighbor on the same row (column); the average distance of two neighbors is $N/(n+m-2)$, where N is the degree of the product. The average distance between two terms of the final product is $N/(nm)$. Since a

covering set (which we supposed to include p_{ij} and p_{kl}) can be considered a not fully updated part of the final product list, it is expected that $e_{kl} > e_{i,j+1}$ in the majority of cases, so that insertion from the bottom of PPL also minimizes the amount of unsuccessful comparisons relative to this pair.

5.3. The Quasi-LI or Q-LI algorithm. In order to compare the LI algorithm with GEN–MULT and SLICE, also relative to time, we implemented a version of LI which is conceptually simpler, the so-called Quasi-LI algorithm. Starting with a covering set, only the right neighbor(s) of the top term is (are) processed. In this way one avoids the complicated checks mentioned under 1 and 2 in § 5.1. However, due to the asymmetric approach the number of exponent comparisons is $\lesssim 10\%$ larger, as can be seen from Table 4.

TABLE 4

SLICE–S and GEN–MULT as compared with ALTRAN's LI algorithm and with a simpler version, Q-LI, which was implemented in order to obtain time comparisons. In the first column the number of terms $n(=m)$ is displayed, in the second the structure number S. The data in the third column are taken from [6]. *The advantage in time of GEN–MULT over Q-LI is mainly due to the simpler administrative procedure of the former algorithm. SLICE–S works in linear time (for fixed S) for these input polynomials and therefore its relative efficiency increases with increasing n.*

		LI	Q-LI		GEN–MULT		SLICE–S	
n	S	Exponent comp./n^2	Exponent comp./n^2	Time/n^2 (millisec)	Exponent comp./n^2	Time/n^2 (millisec)	Exponent comp./n^2	Time/n^2 (millisec)
10	4	1.39	1.47	4.5	1.48	3.0	0.18	1
10	16	1.97	2.17	4.5	2.00	3.5	0.53	2.5
10	64	2.20	2.49	4.5	2.36	4.0	0.78	3.0
30	4	1.93	2.01	7.7	1.83	2.2	0.36	1
30	16	3.80	4.25	8.1	3.44	3.2	1.34	2.0
30	64	5.18	5.98	5.2	5.04	4.3	2.40	2.8
50	4	2.19	2.21	4.7	1.90	2.1	0.37	0.8
50	16	4.83	5.26	6.4	4.01	3.2	1.61	1.7
50	64	7.46	8.91	7.4	7.20	5.2	3.48	2.7
70	4	2.29	2.29	5.0	1.94	2.1	0.37	0.7
70	16	5.47	6.01	7.6	4.39	3.4	1.75	1.6
70	64	9.64	11.28	9.3	8.78	5.9	4.14	2.7
90	4	2.38	2.33	5.2	1.96	2.1	0.40	0.7
90	16	5.94	6.47	8.1	4.65	3.7	1.86	1.6
90	64	11.93	13.28	10.8	10.06	6.9	4.63	2.8

5.4. Comparison of Q-LI with GEN–MULT and SLICE–S. In Table 4 the relative behavior of the various algorithms is displayed for the test data presented by Johnson. Comparing first the symbolic sorting effort of LI (and Q-LI) with GEN–MULT it is seen that GEN–MULT is slightly more profitable. LI recruits new product terms on the basis of the structure of the previous sorted PPL.

The advantage in time of GEN–MULT is considerably greater. This can be understood when realizing that with LI, for each product term processed, one has to keep track of the corresponding row and column number(s). This even leads to the paradoxical situation (in our system) that it takes longer to process 30×30 terms with low value for S (so that many could be contracted) than the sparse case.

SLICE–S greatly profits from structured cases (low value for S). It performs better than GEN–MULT and LI with increasing value of n (keeping S fixed), since the latter

algorithms do not perform in linear time. Since SLICE–S becomes significantly worse than SLICE–O only for very large problems, SLICE–S is expected to perform in linear time (keeping S fixed) for the problems in Table 4. This expectation is confirmed by the test data.

6. Review of the presented algorithms. We have shown that GEN–MULT performed better than Horowitz's SORT–MULT and ALTRAN's LI-algorithm for the test cases presented in the respective papers. In its present form GEN–MULT's theoretical bound is $O(mn^2)$, but transfer to a different data structure could lower this bound to SORT–MULT's $O(mn \log n)$ behavior. GEN–MULT will find useful application only for multivariate polynomials with many indeterminates for which the mapping of the exponent sets required for the SLICE approach, as explained in § 4.3.7, could be cumbersome. Those polynomials, when containing a great number of terms, would be most efficiently handled in recursive representation, as reported by us in [8]. In this case multiplication is likewise done recursively, with each intermediate univariate case, which then would be of moderate size, handled by GEN–MULT.

The SLICE algorithms will be appropriate for the remaining cases, which are the univariate polynomials and the multivariate polynomials with a small (say less than 5) number of indeterminates.

SLICE–S is efficient for moderate size (say up to 100 terms) of the operands and requires $O(n)$ auxiliary space.

SLICE–O requires advance knowledge of the system constant c, although a rough estimate is sufficient for an efficient performance. In its present form it works in near linear time (time/n^2 = constant) as an average for the random case and requires very little auxiliary storage. Although $1/c \approx 20$ terms is the average value for the work space, in exceptional cases this number could be appreciably higher, due to situations where clusters of product terms would appear.

7. Connection with theoretical results. The $O(mn)$ performance of the SLICE algorithms and of Teer's bucket-sort algorithm [14] as an average for the random case can be considered a step forward in the design of efficient algorithms for $X + Y$ sorting. Research should be continued in order to realize the possibility of $O(n^2)$ sorting for any $X + Y$ problem. Although there is an infinite variety in exponent sequence for each of the operands, for fixed n, one should take into account that the number of order types for the product is finite (Harper, Payne, Savage and Straus [4]). The algorithm we are looking for, should be able to dynamically recognize the order type with which the problem corresponds. Furthermore, Fredman's proof [2] is based on queries for each individual product term relative to a suitably chosen integer. This implies that the approach of older algorithms, which merely utilize binary comparisons, may have to be abandoned. The SLICE algorithms could be a first step towards realizing the theoretically found $O(n^2)$ performance.

Acknowledgment. The author wants to thank the reviewers and also Frank Teer for their constructive remarks and for their referral to the literature concerning the mathematical issues of $X + Y$ sorting.

REFERENCES

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

[2] M. L. FREDMAN, *Two applications of a probabilistic search technique: Sorting $X + Y$ and building balanced search trees*, Proc. 7th Annual ACM Symp. on Theory of Computing (Albuquerque, NM), May, 1975, pp. 240–244.

[3] F. GUSTAVSON AND D. Y. Y. YUN, *Arithmetic complexity of unordered sparse polynomials*, Proc. ACM Symp. on Symbolic and Algebraic Computation (Yorktown Heights, NY), Aug., 1976, pp. 149–153.

[4] L. H. HARPER, T. H. PAYNE, J. E. SAVAGE AND E. STRAUS, *Sorting $X + Y$*, Comm. ACM, 18 (1975), pp. 347–349.

[5] E. HOROWITZ, *A sorting algorithm for polynomial multiplication*, J. Assoc. Comput. Mach., 22 (1975), pp. 450–462.

[6] S. C. JOHNSON, *Sparse polynomial arithmetic*, Proc. Eurosam Conf., SIGSAM Bull., 8 (1974), pp. 63–71.

[7] D. A. KLIP, *Algebra by computer*, Technical Report for the Dept. of Computer and Information Sciences, University of Alabama, Birmingham, AL, September, 1973.

[8] ———, *Some aspects of a portable algebra system*, Proc. ACM South-East Regional Conf. (Nashville, TN), April, 1974, pp. 1–22.

[9] ———, *Different polynomial representations and their interaction in the portable algebra system PORT-ALG*, Proc. Eurosam Conf., SIGSAM Bull., 8 (1974), pp. 72–73.

[10] ———, *The variable cell length listprocessor VARLIST*, Proc. ACM National Conf. (San Diego, CA), Nov. 1974, pp. 128–132.

[11] ———, *The VARLIST listprocessing system*, Ref. Manual and Doc. Technical Report for the Dept. of Computer and Information Sciences, University of Alabama, Birmingham, AL, September, 1975.

[12] D. E. KNUTH, *The Art of Computer Programming*, vol. 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1971.

[13] R. MOENCK, *Practical fast polynomial multiplication*, Proc. ACM SYMSAC (Yorktown Heights, NY), August, 1976, pp. 136–148.

[14] F. TEER, *Formula manipulation and PASCAL*, D.Sc. Thesis, Vrije Universiteit, Amsterdam, May, 1978.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0006 $01.00/0

SOME REMARKS ON COMPUTING GALOIS GROUPS*

J. McKAY†

Abstract. Computational techniques based on Chebotarev's density theorem and tests for multiple transitivity are of use in finding the Galois group of a given polynomial. A computer search for trinomials having certain Galois groups over the rationals has produced examples with $PSL_3(2)$ as Galois group but none with the Mathieu group M_{11}.

Key words. Galois groups, polynomials, permutation groups

Although Galois theory is over a century old, very little has been published on the subject of feasible constructive techniques, with the exceptions of Stauduhar [7], Zassenhaus [8], Lefton [6], and Girstmair and Oberst [9]. The main problems of Galois theory over the rationals are:

(1) To determine Gal (f), given the polynomial $f(x)$.

(2) To find a polynomial $f(x)$ having a given permutation group G as Galois group.

Of these two problems, the second is the more difficult, since the existence of such an f is known only for relatively few groups. There appears to be no published construction producing $f(x)$ from an arbitrary solvable group G even though it is known that such an f exists for these G.

Throughout this paper, we take the base field to be the rationals and, in discussing the first problem, we shall assume that f is irreducible. We note in passing that the case of reducible f is more difficult, since it requires consideration of subfields common to the splitting fields of pairs of factors.

For our purposes, a "good" prime relative to some polynomial f is one which does not divide the discriminant disc (f) of f. If a prime is not good, then it is "bad": there are only finitely many bad primes relative to any given f.

We define the *shape* of a permutation R of degree n to be the partition of n induced by the lengths of the disjoint cycles of R. The factorization of a polynomial f modulo any prime p also induces a partition, namely the partition of deg (f) formed by the degrees of the factors.

The following results follow from the density theorem of Chebotarev (see [5]).

LEMMA 1. *For any good prime p relative to a polynomial f, the degree partition of the factorization of f mod p is the shape of some permutation in* Gal (f).

LEMMA 2. *As $s \to \infty$, the proportion of occurrences of a partition π as degree partition of the factorization of f* mod p_i $(i = 1, 2, \cdots, s)$ *tends to the proportion of permutations in* Gal (f) *whose shape is π.*

Lemma 1 yields lower bounds on Gal (f) based on the degree partitions of the factorizations of f modulo various primes, while Lemma 2 can provide estimates of the proportions of permutations of each shape in Gal (f). These estimates provide a statistically significant guess as to Gal (f) which may later be confirmed by deterministic methods such as the use of invariants.

Effective bounds on these estimates of proportions have been calculated by Lagarios and Odlyzko [5] using assumptions based on the generalized Riemann hypothesis, enabling Gal (f) to be determined uniquely in many cases.

We give tables for all transitive permutation groups of degree $\leqq 7$, supplementing Stauduhar's tables [7] and those of Zassenhaus [8], and include the distribution of

* Received by the editors May 23, 1978.

† Computer Science Department, Concordia University, Montreal, Quebec H3G 1M8.

permutation shapes and permutation generators. The notation G/H indicates that G is represented on the cosets of H. Groups marked "+" are groups of even permutations.

TABLE 1

Distribution of shapes for all transitive groups of degrees 3 to 7.

Degree 3

	1^3	$2\ 1$	3	$\lvert G\rvert$
$+A_3$	1	·	2	3
S_3	1	3	2	6

$a = (1\ 2\ 3)$, $b = (1\ 2)$
$A_3 = \langle a \rangle$, $S_3 = \langle a, b \rangle$

Degree 4

	1^4	$2\ 1^2$	2^2	3 1	4	$\lvert G\rvert$	imp 2×2
Z_4	1	·	1	·	2	4	*
$+V_4$	1	·	3	·	·	4	*
D_8	1	2	3	·	2	8	*
$+A_4$	1	·	3	8	·	12	
S_4	1	6	3	8	6	24	

$a = (1\ 3\ 4)$, $b = (1\ 3)$, $c = (2\ 4)$, $d = (1\ 2)(3\ 4)$
$Z_4 = \langle ac \rangle$, $V_4 = \langle bc, d \rangle$, $D_8 = \langle ac, bc \rangle$,
$A_4 = \langle a, d \rangle$, $S_4 = \langle ac, b \rangle$

Degree 5

	1^5	$2\ 1^3$	$2^2\ 1$	3 2	$3\ 1^2$	4 1	5	$\lvert G\rvert$
$+Z_5$	1	·	·	·	·	·	4	5
$+D_{10}$	1	·	5	·	·	·	4	10
F_{20}	1	·	5	·	·	10	4	20
$+A_5$	1	·	15	·	20	·	24	60
S_5	1	10	15	20	20	30	24	120

$a = (1\ 2\ 3\ 4\ 5)$, $b = (1\ 2)$, $c = (2\ 3\ 5\ 4)$
$Z_4 = \langle a \rangle$, $D_{10} = \langle a, c^2 \rangle$, $F_{20} = \langle a, c \rangle$,
$A_5 = \langle a, bab \rangle$, $S_5 = \langle a, b \rangle$

Degree 6

	1^6	$2\ 1^4$	$2^2\ 1^2$	2^3	$3\ 1^3$	$3\ 2\ 1$	3^2	$4\ 1^2$	4 2	5 1	6	$\lvert G\rvert$	imp 2×3	imp 3×2
Z_6	1	·	·	1	·	·	2	·	·	·	2	6	*	*
S_3	1	·	·	3	·	·	2	·	·	·	·	6	*	*
D_{12}	1	·	3	4	·	·	2	·	·	·	2	12	*	*
$+A_4$	1	·	3	·	·	·	8	·	·	·	·	12		*
G_{18}	1	·	·	3	4	·	4	·	·	·	6	18	*	
G_{24}	1	3	3	1	·	·	8	·	·	·	8	24		*
$+S_4/V_4$	1	·	9	·	·	·	8	·	6	·	·	24		*
S_4/Z_4	1	·	3	6	·	·	8	6	·	·	·	24		*
G_{36}^1	1	·	9	6	4	·	4	·	·	·	12	36	*	
$+G_{36}^2$	1	·	9	·	4	·	4	·	18	·	·	36	*	
G_{48}	1	3	9	7	·	·	8	6	6	·	8	48		*
$+PSL_2(5)$	1	·	15	·	·	·	20	·	·	24	·	60		
G_{72}	1	6	9	6	4	12	4	·	18	·	12	72	*	
$PGL_2(5)$	1	·	15	10	·	·	20	30	·	24	20	120		
$+A_6$	1	·	45	·	40	·	40	·	90	144	·	360		
S_6	1	15	45	15	40	120	40	90	90	144	120	720		

$a = (1\ 2\ 3)$, $b = (1\ 4)(2\ 5)(3\ 6)$, $c = (1\ 5\ 2\ 4)(3\ 6)$, $d = ab$, $e = bc^2$,
$r = (1\ 2)$, $s = (1\ 3\ 5)(2\ 4\ 6)$, $t = rsrs^2$, $v = (1\ 3)(2\ 4)$,
$w = (1\ 6)(2\ 5)(3\ 4)$, $x = (1\ 2\ 3\ 4\ 5)$, $y = (1\ 6)(2\ 5)$, $z = (2\ 3\ 5\ 4)$

$Z_6 = \langle d \rangle$, $S_3 = \langle e, w \rangle$, $D_{12} = \langle d, e \rangle$, $G_{18} = \langle a, b \rangle$, $G_{36}^1 = \langle a, b, e \rangle$,
$G_{36}^2 = \langle a, c \rangle$, $G_{72} = \langle a, b, c \rangle$, $A_4 = \langle s, t \rangle$, $G_{24} = \langle r, s \rangle$, $S_4/V_4 = \langle s, t, v \rangle$,
$S_4/Z_4 = \langle s, t, w \rangle$, $G_{48} = \langle r, s, v \rangle$, $PSL_2(5) = \langle x, y \rangle$, $PGL_2(5) = \langle x, y, z \rangle$,
$A_6 = \langle c, x \rangle$, $S_6 = \langle d, x \rangle$

Degree 7

	1^7	$2\ 1^5$	$2^2\ 1^3$	$2^3\ 1$	$3\ 1^4$	$3\ 2\ 1^2$	$3\ 2^2$	$3^2\ 1$	$4\ 1^3$	$4\ 2\ 1$	$4\ 3$	$5\ 1^2$	$5\ 2$	$6\ 1$	7	$\lvert G\rvert$
$+Z_7$	1	·	·	·	·	·	·	·	·	·	·	·	·	·	6	7
D_{14}	1	·	·	7	·	·	·	·	·	·	·	·	·	·	6	14
$+F_{21}$	1	·	·	·	·	·	·	14	·	·	·	·	·	·	6	21
F_{42}	1	·	·	7	·	·	·	14	·	·	·	·	·	14	6	42
$+PSL_3(2)$	1	·	21	·	·	·	·	56	·	42	·	·	·	·	48	168
$+A_7$	1	·	105	·	70	·	210	280	·	630	·	504	·	·	720	2520
S_7	1	21	105	105	70	420	210	280	210	630	420	504	504	840	720	5040

$a = (1\ 2\ 3\ 4\ 5\ 6\ 7)$, $b = (2\ 4\ 3\ 7\ 5\ 6)$, $c = (2\ 3)(4\ 7)$, $d = (1\ 2\ 3)$
$Z_7 = \langle a \rangle$, $D_{14} = \langle a, b^3 \rangle$, $F_{21} = \langle a, b^2 \rangle$, $F_{42} = \langle a, b \rangle$, $PSL_3(2) = \langle a, c \rangle$,
$A_7 = \langle a, d \rangle$, $S_7 = \langle b, d \rangle$

One may obtain upper bounds for Gal (f) using a method based on the following lemma [4]:

LEMMA 3. *Let K_a be the field generated over the prime field F by $A = \{a_i\}$, $|A| = n$, and similarly K_b generated over F by $B = \{b_k\}$, the partial sums $r < n$ at a time of the $\{a_i\}$, $|B| = m = {}^nC_r$: then,*

(i) $K_a = K_b$ *if* char $F \nmid r$,

and further, if $s = \sum a_i \in F$, then

(ii) $K_a = K_b$ *if* char $F \nmid n$.

Proof. (i) We may assume that $r > 1$. Now $a_i - a_j \in K_b$ since $a_i - a_j$ is the difference of two of the b's differing in one place. Suppose that $b_k = \sum_{j \in I} a_j$, $|I| = r$, $i \in I$; then

$$b_k + \sum_{j \in I} (a_i - a_j) = r a_i \in K_b.$$

(ii) With the further assumption that $s \in F$ we can repeat the above argument replacing the b_k by the complementary sums $s - b_k$ which yields the result for the case char $F \nmid n - r$, and so, using (i), we obtain (ii).

This result is used to show that the Galois group over the rationals of P_a with zeros $\{a_i\}$ is isomorphic to that of P_b with zeros $\{b_k\}$.

PROPOSITION. *$P_b(x)$ (with distinct zeros) is reducible if and only if* Gal (P_a) *is not r-transitive on its zeros.*

Here we take r-transitive to mean r-fold set-transitive, a concept which coincides with ordinary multiple transitivity in most cases (see Cameron [3] for exceptions). We note that by a simple counting argument $P_b(x)$ has at least $r(n-r)+1$ distinct zeros.

If we can prove by the Chebotarev lemmas above that $G \leqq \text{Gal}\,(f) \leqq A_n$, where G is a maximal subgroup of A_n, then this transitivity test will usually determine Gal (f). In practice it seems possible to determine a candidate for a factor of $P_b(x)$ derived from $f = P_a(x)$ by approximation before attempting to factorize it. Two typical examples are:

(1) $PSL_3(2)$: order 168, deg $(P_a) = 7$, $r = 3$. Polynomial P_b has degree ${}^7C_3 = 35$ and has an irreducible factor of degree 7 corresponding to the Steiner system $S(2, 3, 7)$.

(2) The Mathieu group M_{11}: order 7920, deg $(P_a) = 11$, $r = 5$. Polynomial P_b has degree ${}^{11}C_5 = 462$ and has an irreducible factor of degree 66 corresponding to the Steiner system $S(4, 5, 11)$.

The Steiner systems arise from the partial sums. The subscripts of the zeros of P_a which occur will form a Steiner system.

With regard to our second problem, that of finding a polynomial with a given Galois group, we mention first the effective methods of Atkin and Swinnerton-Dyer [1].

In our own search for polynomials with given Galois groups, we have concentrated on trinomials $x^n + ax^k + b$, using the following computer search technique (implemented by E. Regener and R. Rohlicek):

(1) For a small set of primes $T = \{p_i\}$ store those polynomials f_j with coefficients mod p_i such that $p_i | \text{disc}\,(f_j)$ or whose factorizations are "good". By a good factorization we mean one whose degree partition modulo a good prime is the shape of a permutation in the group under investigation.

(2) Let S be a subset of T consisting of the s primes having the smallest proportion of good factorizations (usually we take $s = 2$ or 3). For each s-tuple of trinomials $(f_1, \cdots, f_s)$ in the tables, lift the coefficients from residues mod p_i to residues mod $P = \sum_{i \in S} p_i$ by the Chinese remainder theorem. Now test all trinomials $f(x) = x^n + Ax^k + B$ with integer coefficients within a predetermined range, choosing all A and B within the range whose residues mod P are those given.

When Gal (f) consists only of even permutations, one can restrict the coefficients A and B by the condition that disc (f) is positive, using Swan's expression for the discriminant of a trinomial [2, p. 163]. In this case, of course, disc (f) must be square. Often the number of real roots of f, as determined by the Sturm remainder sequence, is a further restriction on the possible values of the coefficients, since the number of real roots is the number of fixed points of the involution induced by complex conjugation if this operation is nontrivial in Gal (f).

Our search has succeeded in finding several trinomials f for which Gal $(f) = PSL_3(2)$ on 7 letters. Each of these polynomials is of the form $x^7 + 7Ax^k + B$. No trinomials of the form $x^{11} + 11Ax^k + B$ with Galois group M_{11} have been found for $1 \leqq k \leqq 10$ and for coefficients in the ranges $|11A| \leqq 100{,}000$ and $|B| \leqq 50{,}000$. It would be interesting to know whether all trinomials $x^7 + Ax^k + B$ whose Galois group is $PSL_3(2)$ must have A divisible by 7.

Acknowledgment. I thank Dr. David Ford for his help in the preparation of the tables.

REFERENCES

[1] A. O. L. ATKIN AND H. P. F. SWINNERTON-DYER, *Modular forms on noncongruence subgroups*, Proc. Amer. Math. Soc. Symp. Pure Math., 19 (1971), Combinatorics, pp. 1–25.

[2] E. BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[3] P. CAMERON, *Transitivity of permutation groups on unordered sets*, Math. Z., 148 (1976), pp. 127–139.

[4] D. ERBACH, J. FISCHER AND J. MCKAY, *Polynomials with $PSL_3(2)$ as Galois Group*, J. Number Theory, to appear.

[5] J. C. LAGARIOS AND A. M. ODLYZKO, *Effective versions of the Chebotarev density theorem*, Algebraic Number Fields (L-functions and Galois Theory), A. Frölich, ed., Academic Press, 1977, pp. 409–464.

[6] P. LEFTON, *Galois resolvents of permutation groups*, Amer. Math. Monthly, 84 (1977), pp. 642–644.

[7] R. P. STAUDUHAR, *The determination of Galois groups*, Math. Comput., 27 (1973), pp. 981–996.

[8] H. ZASSENHAUS, *On the group of an equation*, Computers in Algebra and Number Theory, G. Birkhoff and M. Hall, eds., SIAM and AMS Proc., 1971, pp. 69–88.

[9] K. GIRSTMAIR AND U. OBERST, *Ein Verfahren konstruktiven Bestimmung von Galoisgruppen*, Jahrbuch Überblick, Birkhauser-Verlag, 1976, pp. 33-44.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0007$01.00/0

UNIFORM BOUNDS FOR A CLASS OF ALGEBRAIC MAPPINGS*

DAVID Y. Y. YUN†

Abstract. The computation of residues with respect to a set of given moduli and the Chinese remainder algorithm can be considered a pair of general invertible algebraic mappings. This class of algebraic mappings include the more familiar mappings of evaluation and interpolation as well as forward and inverse fast Fourier transform (FFT). The utility and significance of these mappings are fully recognized in such fields as symbolic and algebraic computation and signal processing. The importance of the more general pair of mappings as algebraic techniques is just beginning to be appreciated. All these mapping techniques are presented in this paper from a unifying perspective. Then, a uniform upper bound for these pairs of invertible mappings in terms of computational cost or time is established. Hopefully, this effort alleviates concerns of applicability of these mapping techniques and encourages their use in numerous other potential application areas.

Key words. residue computation, Chinese remainder algorithm, evaluation, interpolation, fast Fourier transform (FFT), symbolic and algebraic computation, extended Euclidean algorithm, asymptotic computational complexity, time or cost upper bound

1. Introduction. Algebraic mappings such as polynomial interpolation and inverse fast Fourier transform are well known to be special cases of the more general mapping of the Chinese remainder theorem (CRT) in arbitrary Euclidean domains. Each of these mappings, denoted here generally as φ^{-1}, is the inverse of another mapping, denoted by φ, and when taken together they form invertible pairs of mappings satisfying the relation:

$$\varphi^{-1} \circ \varphi = \varphi \circ \varphi^{-1} = 1,$$

where $\circ$ denotes composition and 1 denotes the identity mapping. The inverse mapping corresponding to interpolation is polynomial evaluation whereas the forward fast Fourier transform and the inverse FFT (denoted forthwith by FFT and IFFT respectively) form another pair of invertible mappings. Both of these pairs of mappings are derivable from the more general pair of algebraic mappings—residue(s) computation (also known as computing the modular representation) and the Chinese remainder algorithm (CRA). It is well known that FFT and IFFT both have the same computational cost (asymptotic upper bound) of $O(n \log n)$ while evaluation and interpolation have a slightly higher but still equal cost of $O(n \log^2 n)$. A natural question to ask is whether the more general mappings of residue computation and CRA also have equal computational cost. It can easily be shown that residue computation in the general case has the same computational cost as evaluation of polynomials (i.e., $O(n \log^2 n)$). So the remaining issue is whether computing with CRA can be shown essentially equivalent to a general interpolation process, hence equal in cost with the inverse mapping. Two of the most respected reference books in this field, [1] and [2], give an affirmative indication and a partial treatment of this issue. One offers a "preconditioned" CRA while the other gives a "single precision" version (exactly corresponding to interpolation). It is the intent of this paper to prove that, indeed, this general pair of invertible mappings also has the equal cost of $O(n \log^2 n)$ in a natural way. This general pair of invertible mappings is a common technique in algebraic computation. Specifically, it is often useful in symbolic systems (such as MACSYMA

* Received by the editors May 23, 1978. Presented at the SIAM-SIGSAM Symposium on Computer Algebra, SIAM 1978 Spring Meeting, May 24–26, Madison, Wisconsin.

† Mathematical Sciences Department, IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

and SCRATCHPAD) and in algebraic coding theoretic computations (such as Reed–Solomon or BCH codes) for reducing the problem sizes and recomposing the results. Motivation for clearly demonstrating this uniform computational bound came from two additional avenues. Renewed interest in pairs of invertible mappings can be seen in the area of public-key cryptography [3], [4]. The general CRA turns out to be particularly relevant in the recent work by S. Winograd [5] on the design of (multiplicatively) optimal discrete Fourier transforms and digital filters.

2. Preliminaries and definitions. Although the results and relations on these pairs of invertible mappings hold in arbitrary Euclidean domains, it is most convenient and clear first to restrict our attention to polynomial Euclidean domains, written as $F[x]$ where F is the coefficient domain which is a field. Indeed, it is in the polynomial domains that our main result seems to be new. Furthermore, we will find it necessary to establish notations and definitions which are most clear in the polynomial domain.

Throughout the paper we will use lower case italic letters to denote coefficient domain elements and use uppercase italic letters to denote polynomials in variable x. Thus a general form of a polynomial in $F[x]$ is

$$P(x) = \sum_{i=0}^{\deg(P)} p_i x^i,$$

where the p_i's are in the coefficient field F. If $\deg(P) = n$ then computing $P(x)$ means finding the $n+1$ coefficients p_i. This will involve $O(n)$ coefficient arithmetic operations. Let $M(n, m)$ be used to denote the upper bound on the number of coefficient arithmetic operations to multiply two polynomials of degree n and m respectively. It is often abbreviated by $M(n)$ in the case of multiplying two polynomials of equal degree n. The following assumption on $M(n, m)$ is a slight generalization of that made on $M(n)$ in [1]:

$$\textit{Assumption M.}\quad M(n_1, n_2) + \cdots + M(n_{k-1}, n_k) \leqq M(n) \text{ if } n = \sum_{i=1}^{k} n_i.$$

Note the similarity of this convexity assumption with its special case $aM(n) \leqq M(an)$ for all $a \geqq 1$.

Given A and B in $F[x]$ which is a Euclidean domain, there always exist Q and R in the same domain that satisfy the *division (with remainder)* relation:

$$A = BQ + R \qquad \deg(R) < \deg(B).$$

Let $D(n)$ denote the cost of dividing a degree $2n$ polynomial by a degree n polynomial. Then $D(n) = O(M(n)) = O(n \log n)$ [2]. With the concept of division, we have the concept of a *divisor*, the name given to B when the remainder of division with A is 0. For any two polynomials A and B, there exists a *greatest common divisor* (GCD), G, which divides both A and B while any common divisor of A and B also divides G.

The algorithm for computing GCD's in any Euclidean domain is the well known *Euclid's algorithm* (EA). It computes $\mathrm{GCD}(A_0, A_1)$ by an iterative division process yielding a *remainder sequence* $A_0, A_1, \cdots, A_k$, where A_i for $2 \leqq i \leqq k$ is the nonzero remainder from the division of A_{i-2} by A_{i-1}. The fact that EA computes $\mathrm{GCD}(A_0, A_1)$ is easily seen from the division relation of the A_i's and that $\mathrm{GCD}(A_0, A_1) = \mathrm{GCD}(A_1, A_2) = \cdots = \mathrm{GCD}(A_{k-1}, A_k) = A_k$. An algorithm which computes such a GCD with a cost of $O(M(n) \log n)$ is given in [1] which is Moenck's modification (to Euclidean domains) of a divide-and-conquer concept applied to calculation of integer GCD's due initially to Schönhage.

3. Extended Euclidean algorithm. GCD computations often serve the ultimate purpose of reducing rational numbers (fractions) or rational functions (ratios of polynomials) to lowest terms. Thus, it is not the GCD that is of interest but often its corresponding divisors of the numerator and denominator in the original fraction. Another frequent use of the GCD computation is to find solutions X and Y to the equation

$$AX + BY = \mathrm{GCD}(A, B), \quad \text{given } A \text{ and } B.$$

When dealing with finite field, algebraic number, and/or algebraic function arithmetic where $\mathrm{GCD}(A, B) = 1$, this equation provides the often indispensible operation of computing the inverse of A with respect to the modulus B or vice versa, i.e.

$$X = A^{-1} (\mathrm{mod}\ B) \text{ or } Y = B^{-1} (\mathrm{mod}\ A).$$

The algorithm for computing such X and Y given A and B is known as the *extended Euclidean algorithm* (EEA) and is a rather straightforward modification of the iteration in EA:

With $A_0 = A$ and $A_1 = B$,
Let $X_0 \leftarrow 1$; $X_1 \leftarrow 0$; $Y_0 \leftarrow 0$; $Y_1 \leftarrow 1$;
Iterate for $1 \leqq i \leqq k$ as in EA,
$A_{i+1} \leftarrow A_{i-1} - Q_i A_i$ by division, together with the iterations
$X_{i+1} \leftarrow X_{i-1} - Q_i X_i$
$Y_{i+1} \leftarrow Y_{i-1} - Q_i Y_i$
Then $X = X_k$ and $Y = Y_k$ satisfy the desired equation

$$AX + BY = \mathrm{GCD}(A, B).$$

In fact, a similar relation holds for each i:

$$A_0 X_i + A_1 Y_i = A_i (= AX_i + BY_i).$$

This relation forms the basis for the $O(M(n) \log n)$ GCD algorithm given in [1]. We show, simply, how to use procedure HGCD and modify the GCD algorithm in [1] to compute X and Y, with $\mathrm{GCD}(A, B)$ as a by-product. For completeness we will state some definitions and essential results. Most of the omitted proofs can be found in our two principal reference books, [1] and [2].

DEFINITION. Let A_0 and A_1 result in the remainder sequence $A_0, A_1, \cdots, A_k$, and Q_i be the quotient of dividing A_{i-1} by A_i. For $0 \leqq i < j \leqq k$, we define

$$R_{ii} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

$$R_{ij} = \begin{bmatrix} 0 & 1 \\ 1 & -Q_j \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & -Q_{j-1} \end{bmatrix} * \cdots * \begin{bmatrix} 0 & 1 \\ 1 & -Q_{i+1} \end{bmatrix}.$$

LEMMA 3.1.

$$\begin{bmatrix} A_j \\ A_{j+1} \end{bmatrix} = R_{ij} * \begin{bmatrix} A_i \\ A_{i+1} \end{bmatrix} \quad \text{for } 0 \leqq i < j \leqq k, \qquad R_{0j} = \begin{bmatrix} X_j & Y_j \\ X_{j+1} & Y_{j+1} \end{bmatrix} \quad \text{for } 0 \leqq j < k.$$

The principal technique to be applied here is divide-and-conquer. An algorithm based on it aims to find the middle element of the remainder sequence with respect to A_0 and A_1, i.e., the element of the remainder sequence which has the smallest degree exceeding half that of A_0. An essential concept for discussing such a midpoint is the following:

DEFINITION. $l(i)$ is the *unique integer* such that $\deg(A_{l(i)}) > i$ and $\deg(A_{l(i)+1}) \leqq i$.

LEMMA 3.2. *Given polynomials* A_0 *and* A_1 *in* F[x] *with* $\deg(A_0) = n > \deg(A_1)$, $\mathrm{HGCD}(A_0, A_1) = R_{0,l(n/2)}$, *i.e., the first row of this* 2×2 *matrix, say* (X_i, Y_i), *satisfies* $A_0 X_i + A_1 Y_i = A_i$ *with* $\deg(A_i) > n/2$ *and the second row gives the next* X_{i+1} *and* Y_{i+1} *which yields* A_{i+1} *with degree* $\leqq n/2$. *The computing cost of* HGCD *is* $O(M(n) \log n)$.

Based on these results, we can now state the extended Euclidean GCD algorithm which is a modification of Procedure GCD from [1, p. 308].

Input: Polynomials A_0 and A_1 in F[x] where $n = \deg(A_0) > \deg(A_1)$.

Output: A row vector (X_k, Y_k) such that

$$(X_k, Y_k) * \begin{bmatrix} A_0 \\ A_1 \end{bmatrix} = X_k A_0 + Y_k A_1 = \mathrm{GCD}(A_0, A_1).$$

procedure EEGCD(A_0, A_1);
if A_1 divides A_0 **then return** (0, 1);
else begin
$R \leftarrow \mathrm{HGCD}(A_0, A_1)$;

$$\begin{bmatrix} B_0 \\ B_1 \end{bmatrix} \leftarrow R * \begin{bmatrix} A_0 \\ A_1 \end{bmatrix};$$

if $B_1 = 0$ **then return** first row of R;
if B_1 divides B_0 **then return** second row of R;
Q and $C \leftarrow$ respectively, quotient and remainder of dividing B_0 by B_1;

$$\textbf{return } \mathrm{EEGCD}(B_1, C) * \begin{bmatrix} 0 & 1 \\ 1 & -Q \end{bmatrix} * R$$

end
end EEGCD;

It is interesting to compare this algorithm with Procedure GCD of [1] in terms of their similarities and differences. Their similarity indicates the relatively minor additional effort for finding not only the GCD but also solving the equation $A_0 X_k + A_1 Y_k = \mathrm{GCD}(A_0, A_1)$. The difference, mainly the statement with a test for $B_1 = 0$, actually points out a subtle error in Procedure GCD which is also present in HGCD and causes them to give unexpected answers contrary to those predicted by theory. This error was discovered by F. Gustavson when he implemented and tested an earlier version of EEGCD in a draft of this paper.

The next theorem shows the correctness of Procedure EEGCD and establishes its computing cost.

THEOREM EEGCD. *Given* A *and* B *in a Euclidean polynomial domain* F[x] *with* $n = \deg(A) \geqq \deg(B)$, *the problem of finding* $G = \mathrm{GCD}(A, B)$ *together with unique* X *and* Y *such that* $AX + BY = G$, *where* $\deg(Y) < \deg(A) - \deg(G)$ *and* $\deg(X) < \deg(B) - \deg(G)$, *can be computed in time* $O(M(n) \log n)$.

Proof. It is sufficient to show that algorithm EEGCD successfully computes X $(= X_k)$ and Y $(= Y_k)$, since, then, 2 multiplications and an addition will yield the GCD, G, by Lemma 1. But Lemma 2 implies that the call to HGCD yields $R = R_{0,l(n/2)}$. Then the division of B_0 by B_1 insures that the recursive calls to EEGCD have arguments (i.e. B_1 and C) with degrees less than $n/2$. Thus the algorithm terminates. Furthermore, Lemma 1 and the definition of R_{ij} (specifically, $i = 0$ and $j = k$) guarantee that the final 2×2 matrix must contain X_k and Y_k as the elements of the first row.

The uniqueness and degree constraints on X and Y come directly from EEA. First, EEA and this algorithm guarantee the existence of X and Y satisfying $AX + BY = G$.

In fact, since the computation of X and Y uses the quotient sequence associated with A and B, the degree constraints must hold. Assuming the existence of X' and Y' also satisfying the degree constraints and the equation $AX'+BY'=G$, then subtracting the two equations we get $A(X-X')=B(Y'-Y)$. Now, A/G and B/G are relatively prime, so B/G must divide $(X-X')$ which has degree less than $\deg(B)-\deg(G)$. Hence $(X-X')=0$, i.e. $X=X'$. Similarly $Y=Y'$. Therefore, the X and Y satisfy the degree constraints and $AX+BY=G$ must be unique.

The computing cost is expressed by the inequality

$$T(n)\leqq T(n/2)+c_1M(n)+c_2M(n)\log n.$$

The first term on the right-hand-side accounts for the recursive call to EEGCD. The second term accounts for all the multiplications and divisions in the algorithm. The final term, which is the dominant term of this recurrence relation, reflects the cost of HGCD which in fact does most of the work. The cost of algorithm EEGCD is, therefore, easily seen to be $O(M(n)\log n)$. □

4. Residue computation and Chinese remainder algorithm. The modular representation of an element A of a Euclidean domain with respect to a given set of k *moduli*, $P_1, P_2, \cdots, P_k$, is a set of *residues* or remainders, $R_1, R_2, \cdots, R_k$, which result from dividing A by P_i for $i=1,2,\cdots,k$. Generalizing from integers to polynomial Euclidean domains, if A and P are two polynomials satisfying $\deg(A)<\deg(P)$ where $P=P_1P_2\cdots P_k$ and P_i's are pairwise relatively prime, then A is uniquely represented by the set of residues R_i w.r.t. P_i. Specifically, given R_i and P_i, the unique A with degree less than $\deg(P)$ can be computed by the generalized *Lagrange interpolation formula*:

$$\sum_{i=1}^{k} R_iC_iD_i \text{ modulo } P, \quad \text{where } C_i=P/P_i \text{ and } D_i=C_i^{-1} \text{ modulo } P_i.$$

This is the well-known concept of modular representation and Chinese remainder algorithm. Its use for (large) integer calculations is common in such areas as number theory. It has also become a popular technique (sometimes known as the modular homomorphism technique) for computations with polynomials, particularly in symbolic and algebraic computational algorithms and systems. However, the use there is almost exclusively restricted to moduli which are linear polynomials of the form $x-b$. The modular technique with linear moduli corresponds precisely to the more special technique of evaluation and interpolation of polynomials. This is perhaps the reason for some recent literature on this subject to be content in only describing and analyzing such special versions. Clearly, if such restrictions to *single-precision* integers were made, it would immediately be deemed unnecessary and unnatural. In view of the need to complete the picture for this important and useful class of algebraic mapping techniques and the renewed interests in computations in the general case, the rest of this paper intends to present, analyze, and demonstrate the equal computational cost for the general case. Perhaps, it should be stressed that this equal upper bound for the computational cost of this general pair of mutually invertible algebraic mappings does not imply the equivalence of these two problems.

The overall measure for the size of a problem will be n which is assumed to be the degree of P, and hence bounds the size of every intermediate expression that appears during computation with the modular technique. Furthermore, let $\deg(P_i)=n_i$; then $n=n_1+n_2+\cdots+n_k$. The following result on residue computation appears in special versions in [1] and [2]. Since our version is slightly more general, we include a derivation here for completeness.

The problem of residue computation is finding $R_i = A \pmod{P_i}$, for $i = 1, 2, \cdots, k$, given A and P_i. The technique to be employed is that of divide-and-conquer. A simplifying assumption to be made without loss of generality for results that follow is $k = 2^l$. We first compute P and all the intermediate *binary products* ([2] calls them *super moduli*)

$$P_1P_2, \cdots, P_{k-1}P_k, \prod_{i=1}^{4} P_i, \cdots, \prod_{i=k-3}^{k} P_i, \cdots, \prod_{i=1}^{k/2} P_i, \prod_{i=k/2+1}^{k} P_i.$$

Let the *level*-0 binary products be simply the given $P_1, P_2, \cdots, P_k$. Thus, the *level-m* binary products will consist of consecutive products of m P_i's in the form

$$\prod_{i=j2^m+1}^{(j+1)2^m} P_i \quad \text{for } j = 0, 1, \cdots, 2^{l-m} - 1,$$

so that the level-l binary product is P. The computation of any level-$(m+1)$ binary product is a multiplication of two consecutive level-m binary products which were computed before. Thus, at the mth level for $m = 1, 2, \cdots, \log k (= l)$, there are 2^{l-m} multiplications of level-(m-1) binary products. But Assumption M implies the cost of all level-m multiplications is bounded by $M(n)$. Therefore we have derived the following lemma:

LEMMA 4.1. *Given $P_1, P_2, \cdots, P_k$ with degrees $n_1, n_2, \cdots, n_k$ respectively, $P = P_1 \cdots P_k$ of degree n and all binary products can be computed with a cost of $O(M(n) \log k)$, hence $O(M(n) \log n)$ since $k \leqq n$.*

Armed with the binary products, the divide-and-conquer technique is now applied in a reversed direction to compute the R_i's. Initially, if the given A has degree $\geqq n$ then its remainder upon division by P is computed. This *level-l remainder* is used to compute two new *level-$(l-1)$ remainders* upon division by the level-$(l-1)$ binary products. In general, at level-m for $0 \leqq m < l$, each of 2^{l-m-1} already computed *level-$(m-1)$ remainders* are sequentially divided by two of the level-m binary products to get 2^{l-m} *level-m remainders*. Since $D(n) = M(n)$, it is clear that the computation at each level is $M(n)$ so that total cost for computing the R_i's is $O(M(n) \log k)$, or we have

LEMMA 4.2. *Given A of degree $\leqq 2n$ and $P_1, P_2, \cdots, P_k$ of degree $n_1, \cdots, n_k$ respectively where $n = n_1 + n_2 + \cdots + n_k$, $R_i = A \pmod{P_i}$ can be computed in cost of $O(M(n) \log k)$, hence $O(M(n) \log n)$.*

It is well known that evaluation is a special case of residue computation where moduli P_i's are taken to be linear polynomials of the form $x - b_i$ for a set of n distinct points. Thus it is clear that the general result, Lemma 4.2, implies evaluation at n distinct points can also be done in $O(M(n) \log n)$ cost. The same divide-and-conquer technique applies with these linear polynomial moduli. In the case of FFT, these n points b_i became the special nth *roots of unity* which are ith powers, $i = 0, 1, \cdots, n-1$, of the *principal n-th root of unity*, ω. The special (cyclic multiplicative) properties of ω are responsible for the reduction of the cost from $O(M(n) \log n)$ for evaluation at arbitrary points to $O(M(n))$ or $O(n \log n)$ for FFT. Let us now backtrack in this type of deduction. IFFT is also $O(n \log n)$ or $O(M(n))$, again due to the special properties of ω. The similarity of the formula for FFT and IFFT is striking. This kind of similarity is exploited in general when interpolation at n points with n functional values is shown reducible to evaluation; cf. [1]. Specifically, a critical step of this reduction is the computation of the derivative of $P = (x - b_1)(x - b_2) \cdots (x - b_k)$ and the evaluation of that derivative at $b_1, b_2, \cdots, b_k$. Thus, interpolation, as the inverse mapping of evaluation, has the same cost $O(M(n) \log n)$ as evaluation. In view of the discussion on the Lagrange interpolation formula above and the fact that all operations in the formula can

be performed for the general Chinese remainder process, it is clear that interpolation is a special case of CRA. What remains is to show that CRA, as inverse mapping of residue computation, also has a cost of $O(M(n)\log n)$. The plan for showing this result is precisely parallel to the case of reducing interpolation to evaluation. Here we reduce CRA to residue computation and show that the additional computations for the completion of CRA are also bounded by $O(M(n)\log n)$.

First, we state the most general theorem with the desired computational complexity on CRA to date. This result relies on preconditioning and it can be most directly specialized to the case of interpolation. By *preconditioning* we mean that all quantities depending only on the fixed portion of the input are precomputed and supplied as if they are also inputs. Thus preconditioning is most advantageous when a number of problems depend on a common portion of the input or the same input is used for several different problems. The disadvantage of it is that computational cost improvements are not realizable when inputs vary with the problem or when the common portion of input is hard to recognize. We follow the version in [1, Theorem 8.13] with only slight modifications.

LEMMA 4.3 (residue computation). *Suppose $P_1, P_2, \cdots, P_k$ are pairwise relatively prime polynomials in $\mathrm{F}[x]$ of degree at most d, and $M(n)$ is the number of arithmetic steps needed to multiply two n-th-degree polynomials. Then given polynomials $R_1, R_2, \cdots, R_k$, where the degree of R_i is less than that of P_i, for $1 \leqq i \leqq k$, there is a preconditioned algorithm to compute the unique polynomial A of degree less than that of $P = P_1P_2 \cdots P_k$ such that $R_i = A \pmod{P_i}$ for $i = 1, 2, \cdots, k$ in a cost of $O(M(n)\log k)$, hence $O(M(n)\log n)$.*

The algorithm for this form of Chinese remaindering is based on the generalized Lagrange interpolation formula given earlier

$$\sum_{i=1}^{k} R_iC_iD_i \text{ modulo } P, \quad \text{where } C_i = P/P_i \text{ and } D_i = C_i^{-1} \text{ modulo } P_i.$$

If this formula or CRA is used for several different sets of R_i's but the same set of P_i, then it is clearly advantageous to precompute the C_i's and the D_i's since they depend only on the P_i's. However, there are certainly many useful situations where the set of P_i's will vary with the set of R_i's. If it is possible to establish the same computational complexity without preconditioning, then the concern for common input of different problems naturally disappears. In particular, interpolation through n points can be done with a cost of $O(M(n)\log n)$, based on Lemmas 4.3 and 4.4:

LEMMA 4.4. *Let $P_i = x - b_i$ for $1 \leqq i \leqq k$, where the b_i's are distinct (i.e., the P_i's are relatively prime). Let $P = P_1P_2 \cdots P_k$, $C_i = P/P_i$ and D_i be the constant polynomial such that $D_iC_i = 1 \pmod{P_i}$. Then $D_i = 1/v$ where $v = P'(b_i)$ (i.e., derivative of P with respect to x evaluated at $x = b_i$).*

We first note that the critical point of reducing interpolation to evaluation is in the computation of the derivative of $P = P_1P_2 \cdots P_k$ and then evaluating that derivative. The derivative of P in the case of the general CRA has the form

$$P' = \sum_{i=1}^{k} P_i' \prod_{j \neq i} P_j.$$

However, in the case of interpolation where $P_i = (x - b_i)$, P_i' is always 1. In the more general setting of the Chinese remainder theorem, P' can be used for the subsequent computations leading to the final sum in the Lagrange interpolation

formula. In order not to digress excessively, however, we simply set all P_i' to 1, define

$$P^1 = \sum_{i=1}^{k} \prod_{j \neq i} P_j$$

and use P^1 for the reduction of CRA to residue computation. It is clear that $P^1 \equiv C_i \pmod{P_i}$ for all $1 \leqq i \leqq k$. Thus, the results of residue computation with P^1 and P_i's as input are all the $E_i = (C_i \bmod P_i)$. D_i's satisfying $D_i C_i \equiv 1 \pmod{P_i}$, can now be computed from each pair of E_i and P_i via the algorithm EEGCD. With all the P_i's, C_i's, D_i's, and for any set of R_i's, carrying out the general CRA via Lagrange interpolation formula is simply the computation for the sum modulo P. We can now state the main theorem and prove the cost bound which holds for general CRA without preconditioning.

THEOREM CRA. *Suppose $P_1, P_2, \cdots, P_k$ are pairwise relatively prime polynomials in $\mathrm{F}[x]$ of degree $n_1, n_2, \cdots, n_k$ respectively. Then given polynomials $R_1, R_2, \cdots, R_k$, with $\deg(R_i) < \deg(P_i)$ for $1 \leqq i \leqq k$, there is an algorithm with cost $O(M(n) \log n)$ to compute the unique polynomial A with $\deg(A) < n = \deg(P = P_1 P_2 \cdots P_k) = n_1 + n_2 + \cdots + n_k$ such that $R_i = A \pmod{P_i}$ for $i = 1, 2, \cdots, k$.*

Proof. The steps for this CRA has been given, so it only has to be shown that each is bounded by $O(M(n) \log n)$.

(1) The first step is the computation of P and all binary products. By Lemma 4.1, the cost for this step is $O(M(n) \log k)$.

(2) Since $\deg(P^1) \leqq \deg(P') < \deg(P) = n$, the step for residue computation of P^1 with respect to the P_i's (i.e., getting the E_i's) is also bounded by $O(M(n) \log k)$, by Lemma 4.2. The actual computation of P^1 is a special case of the recursive formula given below (in (4) with R_i and $C_i = 1$), so its bound is also $O(M(n) \log k)$.

(3) The next step involves computing D_i from E_i and P_i for $i = 1, 2, \cdots, k$. But each computation of D_i for a particular i is the application of the EEGCD algorithm which, by Theorem EEGCD, costs $O(M(n_i) \log n_i)$. The total cost for all $i \leqq k$ is

$$\sum_{i=1}^{k} O(M(n_i) \log n_i) \leqq (\log n) \sum_{i=1}^{k} O(M(n_i)) \leqq (\log n) M(n),$$

by a special form of Assumption M.

(4) That the sum can be formed fast follows from the same identity as for the case of preconditioned CRA

$$\begin{aligned}
\sum_{i=1}^{k} R_i C_i D_i &= \sum_{i=1}^{k} R_i D_i \prod_{j=1, j \neq i}^{k} P_j \\
&= \sum_{i=1}^{k/2} R_i D_i \prod_{j=1, j \neq i}^{k} P_j + \sum_{i=k/2+1}^{k} R_i D_i \prod_{j=1, j \neq i}^{k} P_j \\
&= \prod_{j=k/2+1}^{k} P_j \sum_{i=1}^{k/2} R_i D_i \prod_{j=1, j \neq i}^{k/2} P_j + \prod_{j=1}^{k/2} P_j \sum_{i=k/2+1}^{k} R_i D_i \prod_{j=k/2+1, j \neq i}^{k} P_j.
\end{aligned}$$

This recursive formula naturally lends itself to the divide-and-conquer technique. Notice also that all the binary products needed here have already been computed while computing P (in Step (1)). Realizing that the cost of each step of the recursion is bounded by $M(n)$ (again by Assumption M), $\log k$ steps of recursion gives the total cost of $O(M(n) \log k)$.

(5) The final division of the sum by P is clearly $M(n)$. □

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] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, Elsevier, New York, 1975.

[3] W. DIFFIE AND M. HELLMAN, *New directions in cryptography*, IEEE Trans. Infor. Theory, IT-22 (1976), pp. 644–654.

[4] R. L. RIVEST, A. SHAMIR AND L. ADLEMAN, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM, 21 (1978), pp. 120–126.

[5] S. WINOGRAD, *On computing the discrete Fourier transform*, Math. Comp., 32 (1978), no. 141, pp. 175–199.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0008$01.00/0

A STRUCTURE THEOREM FOR EXPONENTIAL AND PRIMITIVE FUNCTIONS*

MICHAEL ROTHSTEIN† AND B. F. CAVINESS‡

Abstract. In this paper a new theorem is proved that generalizes a result of Risch. The new theorem gives all the possible algebraic relationships among functions that can be built up from the rational functions by algebraic operations, by taking exponentials, and by integration. The functions so generated are called exponential and primitive functions. From the theorem an algorithm for determining algebraic dependence among a given set of exponential and primitive functions is derived. The algorithm is then applied to a problem in computer algebra.

Key words. algebraic dependence, differential algebra, Liouville extension fields, computer algebra, structure theorem, exponential and primitive functions

1. Introduction. In this paper we generalize a theorem of Risch [10] (see also [5]). The new theorem gives all the possible algebraic relationships among functions that can be built up from the rational functions by algebraic operations, by taking exponentials, and by integration. We call this class of functions the exponential and primitive functions.

The Risch theorem gives all possible algebraic relationships among a set of elementary functions. His theorem essentially says that except for explicit algebraic relationships all other algebraic relations among elementary functions result from either the law of logarithms, i.e., $\log(ab) = \log(a) + \log(b) +$ a constant, that depends on the branch of the logarithm function, or the exponential law, i.e., $\exp(a+b) = \exp(a)\exp(b)$. The new theorem, proved herein, extends Risch's result to include in addition to the elementary functions other functions defined by indefinite integrals. Examples of functions covered by the new results are the error function, the Fresnel integrals, the normal distribution function from statistics, the sine and cosine integrals (see [8] for definitions of these functions). The new theorem also shows that the only implicit algebraic relationships that can occur among these functions come from the law of logarithms and the exponential law provided that logarithms are not hidden under integral signs.

In § 2 we state some results of Rosenlicht that form the basis for this paper. The results of § 2 are then used in an induction proof of the new structure theorem which is given in § 3 as Corollaries 3.2 and 3.3.

In § 4 an algorithm is described for determining algebraic relationships among the exponential and primitive functions. In § 5 some examples and applications are discussed.

2. Rosenlicht's results. In this section we present the concepts and results from Rosenlicht's paper [12] that are used in § 3. Let k be a subfield of K. If M is a K-module, we say that D *is a* k*-derivation of* K *into* M if D is a k-linear map $D: K \to M$ such that $D(xy) = xD(y) + yD(x)$ for all $x, y \in K$. It follows that $Dx^n = nx^{n-1}Dx$ for all $x \in K$. With $x = 1$, $n = 2$ we get $D1 = 0$; hence $D(x \cdot 1) = x \cdot D1 = 0$ for $x \in k$, that is D vanishes on k. We also obtain that $D(x/y) = (yDx - xDy)/y^2$ for all x and all nonzero $y \in K$.

* Received by the editors May 23, 1978. This work was supported in part by the National Science Foundation under Grant MCS76-23762.

† Department of Mathematics and Computer Science, Universidad Simon Bolivar, Caracas, Venezuela.
‡ Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Troy, New York 12181.

Let Φ be the free K-module generated by the symbols $\{\delta x : x \in K\}$ and Ψ be the K-submodule of Φ generated by all $\delta(x+y)-\delta x-\delta y$, $\delta(xy)-x\delta y-y\delta x$, for $x, y \in K$ and all δx for $x \in k$. Let $\Omega_{K/k}$ denote the K-module Φ/Ψ and d be the k-derivation of K into $\Omega_{K/k}$ with the property that dx is the equivalence class of $\Omega_{K/k}$ to which δx belongs. Then given any k-derivation D of K into K there exists a unique K-homomorphism $\Omega_{K/k} \to K$ which composed with d gives D. $(\Omega_{K/k}, d)$, unique to within isomorphism, is called the *module of k-differentials of K*.

LEMMA 2.1. *Let $k \subset K$ be fields of characteristic zero, let $u_1, \cdots, u_n, v$ be elements of K, with $u_1, \cdots, u_n$ nonzero, and let $c_1, \cdots, c_n$ be elements of k that are linearly independent over the rational numbers Q. Then the element $dv + \sum_{i=1}^{n} c_i\, du_i/u_i$ of $\Omega_{K/k}$ is zero if and only if v and each u_i is algebraic over k.*

THEOREM 2.2. *Let k be a differential field of characteristic zero, K a differential extension field of k with the same field of constants C. For each $i = 1, \cdots, n$ and $j = 1, \cdots, \nu$ let $c_{ij} \in C$ and let v_i be an element of K, u_j a nonzero element of K. Suppose that for each $i = 1, \cdots, n$ and each given derivation D of K*

$$Dv_i + \sum_{j=1}^{\nu} c_{ij} Du_j/u_j \in k.$$

Then either $\operatorname{tr\,deg} k(u_1, \cdots, u_\nu, v_1, \cdots, v_n)/k \geqq n$ (tr deg = *transcendence degree*) *or the n elements of $\Omega_{K/k}$ given by $dv_i + \sum_{j=1}^{\nu} c_{ij}\, du_j/u_j$, $i = 1, \cdots, n$, are linearly dependent over C.*

3. Structure theorem. In this section we present the main result, stated as Corollaries 3.2 and 3.3 of Theorem 3.1. Henceforth all fields are assumed to have characteristic zero. Let k be a differential field and K a differential extension field of k such that $K = k(t)$ for some $t \in K$. t is called a *regular monomial* over k if t is transcendental over k, the field of constants of K = the field of constants of k (i.e., the introduction of t does not introduce any new constants), and t is such that

(i) $Dt = a \in k$. In this case we write $t = \int a$ and call t *primitive* over k, or

(ii) $Dt/t = Da$ for some $a \in k$. In this case we write $t = \exp(a)$ and call t *exponential* over k. D is an arbitrary derivation operator in K. In case (i) if $a = Du/u$ for some $u \in k$, we write $t = \log(u)$, and call t *logarithmic* over k.

Given two differential fields $k \subset K$ we say that K is a *regular Liouville extension of k* if there exist $t_1, \cdots, t_n \in K$ such that $K = k(t_1, \cdots, t_n)$ and each t_i is a regular monomial over $k(t_1, \cdots, t_{i-1})$. K is called a *generalized Liouville extension* if each t_i is either algebraic or a regular monomial over $k(t_1, \cdots, t_{i-1})$, and k and K have the same field of constants.

Let $t \in K$ be primitive over k, t is *simple logarithmic over k* if there exist $u_1, \cdots, u_m \in k$ $(m \geqq 1)$ such that for some constant c, $t + c \in k(\log u_1, \cdots, \log u_m)$. We say that t is *nonsimple* if it is not simple-logarithmic over k.

K is a regular (respectively *generalized*) *log-explicit extension of k* if there exist $t_1, \cdots, t_n$ such that $K = K_n = k(t_1, \cdots, t_n)$ is a regular (generalized) Liouville extension of k and if for each t_j at least one of the following conditions holds:

(i) $t_j = \exp(v_j)$, $v_j \in K_{j-1}$ $(K_0 = k)$.

(ii) t_j is primitive and nonsimple over K_{j-1}.

(iii) $t_j = \log(u_j)$, $u_j \in K_{j-1}$.

(iv) t_j is algebraic over K_{j-1}.

Thus in a log-explicit extension t_j is simple-logarithmic over K_{j-1} if and only if $t_j = \log(u_j)$, $u_j \in K_{j-1}$.

We now introduce some index sets.

Let $K_n = k(t_1, \cdots, t_n)$ be a regular (generalized) Liouville extension of k. For $1 \leqq j \leqq n$, let

$$E_j = \{i : t_i = \exp(a_i),\ a_i \in K_{i-1},\ 1 \leqq i \leqq j\},$$

$$P_j = \{i : t_i' \in K_{i-1},\ 1 \leqq i \leqq j\},$$

$$L_j = \{i : t_i = \log(a_i),\ a_i \in K_{i-1},\ 1 \leqq i \leqq j\}.$$

Note that L_j is a subset of P_j. In most cases of interest $j = n$, in which case we simply write E, P or L.

We wish to pay particular attention to fields that are finite extensions of Q, the field of rational numbers. We let C denote a field of constants which is a finite extension of Q. Let $x_1, \cdots, x_m$ denote $m \geqq 1$ independent variables. We let $F_m = C(x_1, \cdots, x_m)$ be the usual differential field of rational functions in $x_1, \cdots, x_m$ with m (partial) derivations. For notational consistency we denote $x_1, \cdots, x_m$ by $t_1, \cdots, t_m$. Note that each x_i is nonsimple over F_{i-1}. In general $F_n = C(t_1, \cdots, t_n)$ denotes an extension field $F_n = C(x_1, \cdots, x_m, t_{m+1}, \cdots, t_n)$.

We will prove the following:

THEOREM 3.1. *Let $F_n = C(t_1, \cdots, t_n)$ be a generalized log-explicit extension field of C, where C is the field of constants of F_n. If u and v are members of F_n such that $Du/u = Dv$ for all D, then there exist rational numbers r_i and a constant $c \in C$ such that*

$$v = c + \sum_{i \in L} r_i t_i + \sum_{i \in E} r_i a_i$$

where $t_i = \exp a_i$ for $i \in E$.

Before proving this result we show how our desired results follow from it.

COROLLARY 3.2. *Let $F_n = C(t_1, \cdots, t_n)$ be a generalized log-explicit extension of C. Let $a \in F_n$ and assume that $\exp(a)$ is not a regular monomial over F_n. Then there exist rational numbers r_i and a constant $c \in C$ such that*

$$a = c + \sum_{i \in L} r_i t_i + \sum_{i \in E} r_i a_i$$

where $t_i = \exp a_i$ for $i \in E$.

Proof. It is well known [10] if $\exp(a)$ is not a regular monomial then there is a nonzero integer n, a $u \in F_n$, and a (possibly new) constant c_1 such that $(\exp a)^n = c_1 u$. Thus $Du/u = nDa = D(na)$. Let $v = na$ and apply the above theorem to obtain

$$na = c + \sum_{i \in L} r_i t_i + \sum_{i \in E} r_i a_i.$$

Divide both sides of this equation by n to obtain the desired result. □

We now obtain the corresponding result for logarithms.

COROLLARY 3.3. *As in the previous corollary let $F_n = C(t_1, \cdots, t_n)$ be a generalized log-explicit extension of C with $a \in F_n$. If $\log(a)$ is not a regular monomial over F_n then there exist rational integers r, r_i with $r \neq 0$ and a constant $c \in C$ such that*

$$a^r = c\left(\prod_{i \in L} a_i^{r_i}\right)\left(\prod_{i \in E} t_i^{r_i}\right)$$

where $t_i = \log(a_i)$ for $i \in L$.

Proof. Since $\log(a)$ is not a regular monomial over F_n there exists $w \in F$ such that $Dw = Da/a$ (see [10]) which implies that $\exp(w) = c_1 a$ for some constant c_1. But this implies that $\exp(w)$ is not a regular monomial over F_n. Thus by the previous theorem there exist integers r, r_i (with $r \neq 0$) and a constant c_2 such that $rw =$

$c_2+\sum_{i\in L} r_i t_i+\sum_{i\in E} r_i a_i$ where $t_i=\exp(a_i)$ for $i\in E$. Take exponentials of both sides of this equation to obtain

$$(c_1 a)^r=(\exp(w))^r=c_3\Big(\prod_{i\in L} a_i^{r_i}\Big)\Big(\prod_{i\in E} t_i^{r_i}\Big)$$

where c_3 is a constant. Thus

$$a^r=c\Big(\prod_{i\in L} a_i^{r_i}\Big)\Big(\prod_{i\in E} t_i^{r_i}\Big)$$

where $c=c_3/c_1^r$. Note that $c\in F_n$ and hence in C since from the above equation it is a quotient of elements of F_n. □

Now we return to the proof of Theorem 3.1. The proof is by induction on μ the number of nonsimple primitives among $t_1,\cdots,t_n$.

For $\mu=0$ it is easy to see that $F_n=C$ so that $v\in C$.

Induction step: Among the μ ($\geqq 1$) t_i's that are nonsimple primitives with respect to $C(t_1,\cdots,t_{i-1})$ let t_j be the one with the largest subscript. For notational simplicity, let $k=C(t_1,\cdots,t_{j-1})$ and $t=t_j$. There are $u_1,\cdots,u_p,v_1,\cdots,v_p\in F_n$ such that

(i) $Du_i/u_i=Dv_i$ for $i=1,\cdots,p$ and all derivations D of F_n.

(ii) Precisely one member of each pair (u_i,v_i) is algebraic over $k(t,u_1,\cdots,u_{i-1},v_1,\cdots,v_{i-1})$.

(iii) F_n is algebraic over $k(t,u_1,\cdots,u_p,v_1,\cdots,v_p)$. (This follows because tr deg $k(t,u_1,\cdots,u_p,v_1,\cdots,v_p)/k=$ tr deg F_n/k.)

In fact one member of each pair (u_i,v_i) will be some t_i. We will apply Rosenlicht's theorem. Note that:

$$Dt\in k,$$

$$\frac{Du_i}{u_i}-Dv_i=0\in k,\quad \text{for } i=1,\cdots,p,$$

$$\frac{Du}{u}-Dv=0\in k,$$

and that tr deg $k(u,u,\cdots,u_p,v,v_1,\cdots,v_p,t)/k<p+2$. We can conclude that the elements dt, $du_1/u_1-dv_1,\cdots,du_p/u_p-dv_p$, and $du/u-dv$ of $\Omega_{F_n/k}$ are linearly dependent over C. In fact $du/u-dv$ depends linearly on dt and the du_i/u_i-dv_i. We can therefore find constants $\gamma_1,\cdots,\gamma_p,\gamma$ and C such that

$$\frac{du}{u}-dv+\sum_{i=1}^{p}\gamma_i\frac{du_i}{u_i}-dv_i+\gamma\,dt=0. \tag{3.1}$$

Let $c_0=1,c_1,\cdots,c_q$ be a vector space basis for the Q-span of $\gamma_0=1,\gamma_1,\gamma_2,\cdots,\gamma_p$ and write $\gamma_i=\sum_{j=0}^{q} n_{ij}c_j$ with each $n_{ij}\in Q$. Replacing each c_i by $c_i/\mathrm{LCD}\,\{n_{ij}\}$ if necessary we can assume $n_{ij}\in Z$ (LCD means least common denominator). This means, in particular,

$$1=\gamma_0=\sum_{j=0}^{q} n_{0j}c_j=n_{00}c_0;\quad \text{that is } n_{01}=n_{02}=\cdots=n_{0q}=0.$$

We can rewrite (3.1) as

$$\sum_{j=0}^{q} c_j\frac{d(u^{n_{0,j}}u_1^{n_{1,j}}\cdots u_p^{n_{p,j}})}{u^{n_{0,j}}u_1^{n_{1,j}}\cdots u_p^{n_{p,j}}}-d(n_{0,j}v+n_{1,j}v_1+\cdots+n_{p,j}v_p)+\gamma\,dt=0.$$

For $j = 0, \cdots, q$ let $z_j = u^{n_{0j}} u_1^{n_{1j}} \cdots u_p^{n_{pj}}$, $y_j = n_{0j}v + n_{1j}v_1 + \cdots + n_{pj}v_p$ and we have that $Dz_j/z_j = Dy_j$ for all derivations D of F_n and

$$\sum_{j=0}^{q} c_j \frac{dz_j}{z_j} - d\left(\sum_{j=0}^{q} c_j y_j - \gamma t\right) = 0.$$

Since the $\{c_i\}$ are linearly independent over Q, we have, by Lemma 2.1 above, that each z_i and $w = \sum_{j=0}^{q} c_j y_j - \gamma t$ are algebraic over k. We will derive several conclusions from this.

First of all γ must be zero. To see this, note that $\sum_{j=0}^{q} c_j(Dz_j/z_j - Dy_j) = 0$ and hence $\gamma Dt = \sum_{j=0}^{q} c_j(Dz_j/z_j) - Dw$ for all derivations D of F_n. $z_0, \cdots z_q$, and w are algebraic over k, so taking traces with respect to $k_1 = k(z_0, \cdots, z_q, w)$ and dividing by $[k_1 : k]$ gives

$$\gamma Dt = \sum \frac{c_i}{[k_1 : k]} \frac{DN(z_i)}{N(z_i)} - D\frac{1}{[k_1 : k]} \operatorname{Tr}(w)$$

where Tr = trace and N = norm.

If $\gamma \neq 0$, then t would be simple logarithmic over k, contrary to the hypotheses. In particular, we can conclude that $\sum_{j=0}^{q} c_j y_j = w$ (since $\gamma = 0$) is algebraic over k.

Now, let $F' = k(z_0, \cdots, z_q, w, y_1, \cdots, y_q) \subset F_n$. $k(z_0, \cdots z_q, w)$ is an algebraic extension of k and F' is an extension of $k(z_0, \cdots z_q, w)$ by regular logarithmic monomials and possibly algebraics (in any case the logarithms we introduce appear among $y_1, \cdots, y_q$). Furthermore $v = y_0 = (w - \sum_{i=1}^{q} c_i y_i) \in F'$ and $Du/u = Dz_0/z_0 = Dy_0 = Dv$ for all D. F' has one less nonsimple primitive than F_n so by the induction hypothesis, we have:

$$v = c + \sum_{i \in L'} r_i t_i + \sum_{i \in E'} r_i a_i + \sum_{i=1}^{q} \bar{r}_i y_i \tag{3.2}$$

where

$$E' = \{i : t_i = \exp a_i,\ a_i \in F_{i-1} \text{ and } t_i \in k\},$$
$$L' = \{i : t_i = \log a_i,\ a_i \in F_{i-1} \text{ and } t_i \in k\}$$

and r_i, $\bar{r}_i$ are rational numbers. Now recall:

$$\begin{aligned} y_1 &= n_{01}v + n_{11}v_1 + \cdots = n_{11}v_1 + n_{21}v_2 + \cdots \quad \text{since } n_{01} = 0, \\ &\vdots \\ y_q &= n_{0q}v + n_{1q}v_1 + \cdots = n_{1q}v_1 + n_{2q}v_2 + \cdots \quad \text{since } n_{0q} = 0 \end{aligned} \tag{3.3}$$

so v does not appear in $y_1, \cdots, y_q$. Substitute the expressions (3.3) in (3.2) and note that each v_i either equals some t_i with $i \in L$ or equals some a_i where $t_i = \exp(a_i)$, $i \in E$, to obtain

$$n_{00}v = c + \sum_{i \in L} \hat{r}_i t_i + \sum_{i \in E} \hat{r}_i a_i.$$

Divide by n_{00} to obtain the desired result. □

We conclude this section by showing that every generalized Liouville extension of a given field can be embedded in a generalized log-explicit extension of the same field. Then we use this result to obtain a structure theorem for generalized Liouville fields.

LEMMA 3.4. *Let k be a differential field. For every generalized Liouville extension K of k there exists a generalized log-explicit extension L of k and $a_1, \cdots, a_m$ $(m \geqq 0)$ in K such that*

(i) *$K(\log a_1, \cdots, \log a_m) = \bar{K}$ is a regular Liouville extension of K.*

(ii) *$\bar{K}$ and L are differentially isomorphic and the isomorphism holds k fixed.*

Proof. Let $K = K_n = k(t_1, \cdots, t_n)$, $n \geqq 0$. The proof is by induction on n. For $n = 0$, $K_n = k$ and the desired result is trivially true.

Assume that we are given a log-explicit extension L_{n-1} of k and $a_1, \cdots, a_m \in K_{n-1}$ such that

(i) $K_{n-1}(\log a_1, \cdots, \log a_m) = \bar{K}_{n-1}$ is a regular Liouville extension of K_{n-1}.

(ii) $\bar{K}_{n-1}$ and L_{n-1} are differentially isomorphic with k fixed by the isomorphism.

Let σ mapping $\bar{K}_{n-1}$ onto L_{n-1} denote the isomorphism. There are five cases to consider.

(a) t_n is algebraic over K_{n-1}. Let P be the monic irreducible polynomial of least degree over K_{n-1} such that $P(t_n) = 0$. It is not difficult to see that P must be irreducible over $\bar{K}_{n-1}$ for otherwise $\{\log a_1, \cdots, \log a_m\}$ would not be algebraically independent over $K_{n-1}(t_n)$. Thus $\sigma(P)$ must be irreducible over L_{n-1}. Since t_n is algebraic, $\bar{K}_n = K_n(\log a_1, \cdots, \log a_m)$ is a regular Liouville extension of K_n and $L_n = L_{n-1}(s_n)$ is isomorphic to $\bar{K}_n$ where s_n is a root of $\sigma(P)$.

(b) $t_n = \exp(v)$, $v \in K_{n-1}$. By a result of Ostrowski [9] (also see [7]) t_n is a regular monomial over $\bar{K}_{n-1}$ and thus $\bar{K}_n = K_n(\log a_1, \cdots, \log a_m)$ is a regular Liouville extension of K_n. Since L_{n-1} is differentially isomorphic to $\bar{K}_{n-1}$, $\exp(\sigma(v))$ is a regular monomial over L_{n-1} and $L_n = L_{n-1}(\exp(\sigma(v))$ is differentially isomorphic to $\bar{K}_n$.

(c) t_n is primitive and nonsimple over K_{n-1}. t_n nonsimple over K_{n-1} implies that it is nonsimple over $\bar{K}_{n-1}$ and thus by Lemma 3.9 of [6] t_n is a regular monomial over $\bar{K}_{n-1}$. Hence $\bar{K}_n = K_n(\log a_1, \cdots, \log a_m)$ is a regular Liouville extension of K_n. Let s_n be an element of the universal extension of L_{n-1} such that $D(s_n) = \sigma(Dt_n)$. Then $L_n = L_{n-1}(s_n)$ is differentially isomorphic to $\bar{K}_n$.

(d) t_n is primitive and simple-logarithmic over K_{n-1} but is not a regular monomial over $\bar{K}_{n-1}$. Then from the Liouville theorem

$$t_n = c + w + \sum_{i=1}^{m} c_i \log a_i$$

where $w \in K_{n-1}$ and c, c_i are constants in K_{n-1}. Let l be the smallest i such that $c_i \neq 0$. Then $\bar{K}_n = K_n(\log a_1, \cdots, \log a_{l-1}, \log a_{l+1}, \cdots, \log a_m)$ is a regular Liouville extension of K_n. Furthermore $\bar{K}_n$ is isomorphic to $\bar{K}_{n-1}$(solve the above equation for $\log a_l$ to get the isomorphic image of $\log a_l$—everything else remains fixed), and thus $L_n = L_{n-1}$.

(e) t_n is simple logarithmic over K_{n-1} and a regular monomial over K_{n-1}. Then by the weak Liouville theorem [2] (see also Theorem 3 of [12]) there exist constants c_i and u, u_i, all in K_{n-1}, such that

$$Dt_n = Du + \sum_{i=1}^{\nu} c_i Du_i/u_i.$$

Let $\{a_{m+1}, \cdots, a_l\}$ be a maximal subset of $\{u_1, \cdots, u_\nu\}$ with the property that $\hat{K}_{n-1} = K_{n-1}(\log a_1, \cdots, \log a_m, \cdots, \log a_l)$ is a regular Liouville extension of K_{n-1}. Then $\hat{K}_{n-1}$ is differentially isomorphic to $\hat{L}_{n-1} = L_{n-1}(\log \sigma(a_{m+1}), \cdots, \log \sigma(a_l))$ which is log-explicit. Now t_n is not a regular monomial over $\hat{K}_{n-1}$ so apply (d) to $\hat{K}_{n-1}$ and $\hat{L}_{n-1}$ to obtain K_n and L_n. □

We can say more about the way the isomorphism σ behaves. If $K = k(t_1, \cdots, t_n)$ and $L = k(s_1, \cdots, s_\nu)$ and $t_i = \exp(v_i)$ then $\sigma(t_i) = \exp(\sigma(v_i)) = s_j$. If t_i is primitive $\sigma(t_i) = w + \sum_{i=1}^{j} c_i s_i$ where the c_i are constants and w is in L_{j-1}.

Now we give a structure theorem for generalized Liouville fields.

THEOREM 3.5. *Let F_n be a generalized Liouville field $F_n = F_0(t_1, \cdots, t_n)$. Let $a \in F_n$ and assume that $\exp(a)$ is not a regular monomial over F_n. Then there exist constants c_i and r_i, r_i being rational, and w all in F_n such that*

$$a = w + \sum_{i \in P} c_i t_i + \sum_{i \in E} r_i a_i \tag{3.4}$$

where $t_i = \exp(a_i)$ for $i \in E$. Furthermore there exists a positive integer l such that $l(\sum_{i \in P} c_i t_i + w) = \log(t)$ for some $t \in F_n$.

Proof. The proof is an immediate consequence of Corollary 3.2, Lemma 3.4 and the immediately preceding comments. □

Note that in (3.4) we cannot restrict the index in the first sum to be over the index set L for if $\log a$ is a regular monomial over F then for *any* $g \in F$, $t = \int (a'/a + g')$ is a regular monomial over F, but $\exp(t - g)$ is not a regular monomial over $F(t)$.

4. An algorithm to determine algebraic dependence. From the structure theorem for log-explicit fields one can derive an algorithm for finding algebraic relationships among functions built up from the rational functions by integration and by taking exponentials. The class of functions that can be constructed in this manner includes the elementary functions as well as special functions such as the Fresnel integrals, the error function, dilogarithms, exponential and logarithmic integrals, sine and cosine integrals, Spence functions, and other functions defined by indefinite integrals. Functions defined by definite integrals, such as the gamma and beta functions, are not covered by these methods.

To describe the algorithm assume that we have built up a generalized log-explicit field $F = C(t_1, t_2, \cdots, t_m)$ where $t_1 = z$ and $z' = 1$. That is each t_i, $i \geqq 2$, is either algebraic over F_{i-1}, is a regular exponential monomial over F_{i-1}, is a regular logarithmic monomial over F_{i-1}, or is a regular primitive, nonsimple monomial over F_{i-1}.

Now we assume that we are given a t which is one of the following four forms.

(i) $t = \exp(a)$, $a \in F$.

(ii) $t = \log(a)$, $a \in F$.

(iii) $t = \int a\, dz$, $a \in F$.

(iv) t is algebraic over F and we are given the monic polynomial P over F, irreducible and of minimal degree so that $P(t) = 0$.

In case (iv) we can add t to F and carry out all arithmetic modulo $P(t)$.

In case (i) t is either a regular monomial over F or else there exist rational numbers r_i and a constant c such that

$$a = c + \sum_{i \in L} r_i t_i + \sum_{i \in E} r_i a_i \tag{1}$$

where $t_i = \exp(a_i)$ for $i \in E$. Now a is a rational function in $t_1, \cdots, t_n$, each a_i is a rational function in $t_1, \cdots, t_{i-1}$. Thus in equation (1) above we clear denominators and equate coefficients of the monomials in the t_i's to obtain a set of linear equations (with rational coefficients) for the r_i. If the set of equations has no solution then t is a regular monomial over F. Otherwise t is not a regular monomial and taking exponentials of both sides of (1) we have

$$t = c_1 \Big(\prod_{i \in L} a_i^{r_i} \Big) \Big(\prod_{i \in E} t_i^{r_i} \Big)$$

where $t_i = \log(a_i)$ for $i \in L$ and $c_1 = \exp(c)$. Thus we have the relationship between t and the previous t_i's. However there are no general methods known for determining the algebraic relationship of the constant c_1 to the constants in C. For further discussion of this problem see [1].

In case (ii) t is either a regular monomial over F or there exist rational numbers r_i and a constant c such that

$$a = c\left(\prod_{i\in L} a_i^{r_i}\right)\left(\prod_{i\in E} t_i^{r_i}\right) \tag{2}$$

where $t_i = \log a_i$ for $i \in L$. In [3] and [4] Epstein describes algorithms for solving equation (2) for the r_i's. The basic idea is to consider a and the a_i's as products of polynomials (with possibly negative exponents) in $t_1, \cdots, t_n$. Then factor these polynomials into factors that are square-free and pairwise relatively prime. Then equation (2) can hold only if the powers of each factor in the equation are equal. This yields a set of linear equations for the r_i. If the equations have no solution then t is a regular monomial. If not then

$$t = c_1 + \log(c) + \sum_{i\in L} r_i t_i + \sum_{i\in E} r_i a_i$$

where $t_i = \exp(a_i)$ for $i \in E$ and c_1 depends on the branch of the logarithm function. Once again there is no general method for determining the algebraic dependence of $c_1 + \log(c)$ on C but [1] gives some suggestions.

In case (iii) we apply a modification of an integration algorithm for elementary functions (see Appendix A of [13]) to determine if there exist constants c_i, u, and u_i, all in F, such that

$$t = c + u + \sum_{i=1}^{m} c_i \log u_i$$

where c is a constant.

If no such expression exists then by Lemma 3.9 of [6] t is a regular monomial over F. Otherwise we apply the procedure for case (ii) to each $\log u_i$. The constant c depends on which branch of $\int a\,dz$ is used.

5. Examples and applications. We now apply the algorithm of § 4 to an example. Consider the Fresnel integrals

$$S(z) = \int_0^z \sin\left(\frac{\pi}{2}z^2\right) dz,$$

$$C(z) = \int_0^z \cos\left(\frac{\pi}{2}z^2\right) dz$$

and the error function

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_0^z e^{-z^2}\, dz.$$

It is well known that

$$C(z) + iS(z) - \left(\frac{1+i}{2}\right) \operatorname{erf}\left\{\frac{\sqrt{\pi}}{2}(1-i)z\right\} = 0.$$

The reduction of the left-hand side by the above methods, proceeds as follows: First let

$C = Q(i, \pi, \sqrt{\pi})$. Rewrite $\sin((\pi/2)z^2)$ and $\cos((\pi/2)z^2)$ in exponential form. The algorithm then scans the resulting expression from the inside to the outside (that is, if we have nested logarithms, exponentials, and/or integrals the arguments of the outer functions must be processed before the outer function itself is processed) for exponentials, logarithms and integrals that become candidates for regular monomials. For this example let the first subexpression found be $t_2 = \exp(i(\pi/2)z^2)$. (Recall that t_1 denotes the variable z.) According to Theorem 3.3 t_2 is a regular monomial over $F_1 = C(z)$ if there does not exist a constant $c \in C$ such that $i(\pi/2)z^2 = c$. Since z is algebraically independent over F_0 there cannot exist such a constant and t_2 is thus a regular monomial.

Let the next subexpression found be $t_3 = \exp(-i(\pi/2)z^2)$. t_3 is a regular monomial over $F_2 = C(z, t_2)$ only if there do not exist $c \in C$ and $r_1 \in Q$ such that $-i(\pi/2)z^2 = c + r_1(i(\pi/2)z^2)$. This is possible only if the set of linear equations

$$\left\{\begin{array}{rl} c & = 0 \\ \left(i\frac{\pi}{2}\right)r_1 + \left(i\frac{\pi}{2}\right) & = 0 \end{array}\right\}$$

has a solution. A solution is easily computed to be $c = 0$, $r_1 = -1$. If we let a_2, a_3 denote the arguments of t_2, t_3 respectively, our solution gives $a_3 = -a_2$. We take the exponential of both sides of this equation to obtain $t_3 = t_2^{-1}$. Hence we replace $\exp(i(\pi/2)z^2)$ in our original expression by t_2 and $\exp(-i(\pi/2)z^2)$ by t_2^{-1}. The resulting rational expressions for $\sin((\pi/2)z^2)$ and $\cos((\pi/2)z^2)$ are $(t_2^2 - 1)/(2it_2)$ and $(t_2^2 + 1)/(2t_2)$.

We now apply the integration algorithm (an explanation of which is beyond the scope of this paper) to discover that the Fresnel integral $t_3 = C(z)$ is not simple over $F_2 = C(z, t_2)$ and that $t_4 = S(z)$ is not simple over F_3. Hence both are regular monomials.

Now consider $\operatorname{erf}\{(\sqrt{\pi}/2)(1-i)z\} = (2/\sqrt{\pi})\int_0^w e^{-z^2}\,dz$ where $w = (\sqrt{\pi}/2)(1-i)z$. Apply Leibnitz's rule to rewrite this integral as

$$(1-i)\int_0^z \exp\left(\frac{i\pi}{2}z^2\right)dz.$$

Let $t_5 = \int_0^z t_2\,dz$. The integration algorithms discovers that

$$t_5 = t_3 + it_4 + \text{a constant}.$$

If we have some method to determine that the constant is zero, then we have that

$$C(z) + iS(z) - \left(\frac{1+i}{2}\right)\operatorname{erf}\left\{\frac{\sqrt{\pi}}{2}(1-i)z\right\} = t_3 + it_4 - \left(\frac{1+i}{2}\right)(1-i)[t_3 + it_4] = 0.$$

In general the constant depends on the path of integration that is used (i.e. on the choice of the particular single-valued branch of the function defined by the integral). Since this information is not given by the integral sign more information must be given. Even when the path of integration is specified, the authors know of no general method for determining the constant.

If there were methods for determining the constant of integration, problems still arise. The problems arise from the fact that the constant of integration may not lie in our

given field of constants. In this case there are no general methods for determining the algebraic dependence of the new constant on the previous constants.

The above procedures can also be applied to find closed form solutions to all first-order linear differential equations in a rather general sense. If we are given such a differential equation

$$y' + f(z)y = g(z)$$

where $f(z)$ and $g(z)$ belong to some regular Liouville extension field of C, we can first find the representations of $f(z)$ and $g(z)$ and then using the fact that

$$y(z) = \exp\left(-\int f\,dz\right)\left[\int \exp\left(\int f\,dz\right) g(z)\,dz + c\right]$$

we can then find a unique representation for y in terms of the functions used to build up the regular Liouville extension of $F_1 = C(z)$. Note that other logarithmic and primitive functions, other than the ones occurring in f and g, can be included in the extension field if one is interested in knowing whether or not the solution of the differential equation can be expressed in terms of the other functions. If it is possible, the methods described herein will find the proper expression; otherwise one will know that the solution of the differential equation does not lie in any algebraic or simple logarithmic extension of the given field of functions.

It would be desirable to have structure theorems for classes of functions satisfying more complex differential equations. Such theorems would lead to zero-equivalence algorithms for still larger classes of functions as well as contribute to a general theory on closed form solutions of differential equations.

Acknowledgment. Theorem 3.1 and its proof are due to Michael Singer. His methods are shorter and more elegant than our original ones. For his contribution we are deeply grateful. We are also indebted to B. David Saunders for several helpful discussions regarding this paper. Finally the careful reports of the referees lead us to consider substantially revising and, hopefully, substantially improving the original draft of the paper.

REFERENCES

[1] B. F. CAVINESS AND M. J. PRELLE, *A note on algebraic independence of logarithmic and exponential constants*, SIGSAM Bull., 12 (1978), pp. 18–20.

[2] B. F. CAVINESS AND M. ROTHSTEIN, *A Liouville theorem on integration in finite terms for line integrals*, Comm. Algebra, 3 (1975), pp. 781–795.

[3] HARVEY I. EPSTEIN, *Algorithms for elementary transcendental function arithmetic*, Ph.D. Thesis, Univ. of Wisconsin, Madison, 1975.

[4] ———, *Using basis computation to determine pseudo-multiplicative independence*, Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation, R. D. Jenks, ed., August, 1976, pp. 229–237.

[5] HARVEY I. EPSTEIN AND B. F. CAVINESS, *A structure theorem for the elementary functions and its application to the identity problem*, Internal. J. Computer Information Sciences, 3 (1979), pp. 9–37.

[6] IRVING KAPLANSKY, *An Introduction to Differential Algebra*, Hermann, Paris, 1957.

[7] E. KOLCHIN, *Algebraic groups and algebraic dependence*, Amer. J. Math., 90 (1968), pp. 1151–1164.

[8] ERWIN KREYSZIG, *Advanced Engineering Mathematics*, John Wiley and Sons, New York, 1963.

[9] ALEXANDRE OSTROWSKI, *Sur les relations algébriques entre les intégrales indéfinies*, Acta Math., 78 (1946), pp. 315–318.

[10] ROBERT H. RISCH, *The problem of integration in finite terms*, Trans. Amer. Math. Soc., 139 (1969), pp. 167–189.

[11] ———, *Algebraic properties of the elementary functions of analysis*, preprint.

[12] MAXWELL ROSENLICHT, *On Liouville's theory of elementary functions*, Pacific J. Math., 65 (1976), pp. 485–492.

[13] MICHAEL ROTHSTEIN, *Aspects of symbolic integration and simplification of exponential and primitive functions*, Ph.D. Thesis, Univ of Wisconsin, Madison, 1976.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0009$01.00/0

THE COMPLEXITY OF PATTERN MATCHING FOR A RANDOM STRING*

ANDREW CHI-CHIH YAO†

Abstract. We study the average-case complexity of finding all occurrences of a given pattern α in an input text string. Over an alphabet of q symbols, let $c(\alpha, n)$ be the minimum average number of characters that need to be examined in a random text string of length n. We prove that, for large m, almost all patterns α of length m satisfy

$$c(\alpha, n) = \theta\left(\left\lceil \log_q\left(\frac{n-m}{\ln m}+2\right)\right\rceil\right) \quad \text{if } m \leqq n \leqq 2m,$$

and

$$c(\alpha, n) = \theta\left(\frac{\lceil \log_q m\rceil}{m} n\right) \quad \text{if } n > 2m.$$

This in particular confirms a conjecture raised in a recent paper by Knuth, Morris, and Pratt (*Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

Key words. Algorithm, average-case complexity, complexity, decision tree, pattern matching, random string, string, weighted q-ary tree

1. Introduction. A basic string pattern matching problem is to find all occurrences of a given string (called *pattern*) as a contiguous block in an input string (called *text string*). Thus, for the pattern 0 0 1 0 0, there are three occurrences of it to be located in the text string 1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 1 1. Several efficient algorithms have been devised to solve this problem [1], [3], [4]. For example, Knuth, Morris, and Pratt [4] constructed an algorithm that has a worse-case running time of $O(m+n)$, where m and n are the lengths of the pattern and the text string, respectively.

The optimality question of algorithms for the above problem was investigated in Knuth, Morris, and Pratt [4] and in Rivest [6]. In their model, an algorithm is a decision tree that examines the text string one character at a time, and the cost is measured in terms of the number of characters examined. (For a similar model in a related problem, see Aho, Hirshberg, and Ullman [2]). Rivest [6] proved that, for any pattern, an algorithm has to inspect $n-m+1$ characters for some text string. This means that, when $n \gg m$, almost the entire text string has to be examined in the worst case. A different situation exists for the average-case complexity. Let $c(\alpha, n)$ be the minimum average number of characters that need to be examined in a random text string of length n, in order to locate all occurrences of α. Knuth described an algorithm [4, § 8] to show that, for any given pattern α, $c(\alpha, n) \leqq O(n\lceil \log_q m\rceil / m)$ for an alphabet of size q. Thus, for large m, only a small fraction of the characters in the text string need to be looked at. Such "sublinear" algorithms are particularly attractive in situations when a text string is input only once, but will be updated and searched for patterns many times. Knuth conjectured that the algorithm is optimal in the following sense: there exist patterns α of arbitrarily large length m such that $c(\alpha, n) \geqq \Omega(n\lceil \log_q m\rceil / m)$ for all sufficiently large n. This conjecture is interesting since, as shown in [4], there are patterns such as 0^m for which only $O(n/m)$ characters need to be tested on the average.

In this paper, we study the average-case complexity of pattern matching in the model of [4]. We prove that, for large m, almost all patterns α of length m satisfy

* Received by the editors September 8, 1977, and in revised form August 18, 1978. This research was supported in part by National Science Foundation under Grant MCS 72-03752 AO3.

† Computer Science Department, Stanford University, Stanford, California 94305.

$c(\alpha, n) = \theta(\lceil \log_q((n-m)/(\ln m)+2)\rceil)$ if $m \leqq n \leqq 2m$, and $c(\alpha, n) = \theta((\lceil \log_q m \rceil / m)n)$ if $n > 2m$. Moreover, all lower bounds actually apply to the *best-case* performance of any algorithms, not just their average case. These results in particular confirm the above-mentioned conjecture when $n \geqq 2m$. We may add that the case $m \leqq n \leqq 2m$ is mainly of theoretical interest, as the text strings are usually much longer than the patterns in practice.

Definitions and precise statements of the main results are given in § 2. In § 3, we familiarize ourselves with some useful concepts by analyzing the algorithm in [4] for $m \leqq n \leqq 2m$. In the course of analysis, we shall also develop insight into the design of a faster algorithm. An improved algorithm is then described and analyzed in § 4 to establish the upper bounds. In § 5, we define the complexity notion of a "certificate". Our lower bounds then follow from stronger results that we can prove about the length of a minimum certificate. Certain properties of a type of optimal digital search trees (cf. Knuth [5]) are needed in the paper; their derivations are given in the appendices.

2. Definitions and main results. An *alphabet* is a finite, nonempty set of symbols. Throughout our discussions, we will assume a unique underlying alphabet Σ of size q. A *string ζ of length l* is a concatenation of l symbols from Σ, i.e., $\zeta = a_1 a_2 \cdots a_l$ where $l \geqq 0$ and each $a_i \in \Sigma$. We use $\zeta[i]$ to denote a_i, the ith symbol of ζ, and $\|\zeta\|$ to denote l, the length of ζ. The collection of all strings of length l is denoted by Σ^l. Given two strings $\alpha \in \Sigma^m$ and $\beta \in \Sigma^n$ with $m \leqq n$, α is said to be a *substring* of β if $\alpha = \beta[i]\beta[i+1]\cdots\beta[i+m-1]$ for some i, $1 \leqq i \leqq n-m+1$. Alternatively, we say α *occurs* in β, or β contains an *occurrence* of α, etc.; the index i is called the (leftmost) *position* of the occurrence. The substring $\beta[i]\beta[i+1]\cdots\beta[j]$ of $\beta \in \Sigma^n$, where $1 \leqq i \leqq j \leqq n$, will be denoted by $\beta[i:j]$.

A *pattern* is a distinguished string of positive length. Given a pattern α of length m and an integer $n \geqq m$. we shall be interested in locating all occurrences of α in any input string $\zeta \in \Sigma^n$ (ζ is called the *text string*). Let us refer to this as the *pattern-matching problem* with respect to α and n. From now on, the notations α, ζ and m, n will be used exclusively for the pattern, the text string, and their respective length in a pattern-matching problem. Since the problem is trivial when $q = |\Sigma| = 1$, we shall assume $q \geqq 2$.

As our computation model, we consider algorithms that proceed by asking a series of questions $\zeta[i_1] = ?, \zeta[i_2] = ?, \cdots$, where the choice of each position i_r may depend on answers to all previous probes at $\zeta[i_1], \zeta[i_2], \cdots, \zeta[i_{r-1}]$. When the algorithm halts, it must have enough information to determine $A(\alpha, \zeta)$, the set of all leftmost positions of α's occurrences in ζ. Formally, $A(\alpha, \zeta) = \{i | \zeta[i:i+m-1] = \alpha\}$. We shall assume that no question is repeated twice in a series $\zeta[i_1] = ?, \zeta[i_2] = ?, \cdots$, so that an algorithm may be represented by a decision tree with q-ary branchings at each query. (For basic definitions regarding q-ary trees, see Knuth [5].) An example of such a decision tree is shown in Fig. 1, with $\Sigma = \{a, b, c\}$, $\alpha = bb$ and $n = 3$. The queries are enclosed in circles, and an answer $A(\alpha, \zeta)$ is attached to each leaf of the ternary tree.

For given α and n, let $\mathcal{T}(\alpha, n)$ be the set of all decision trees for the pattern-matching problem. For any $T \in \mathcal{T}(\alpha, n)$, let $h_T(\zeta)$ be the number of queries asked by T for the input text string $\zeta \in \Sigma^n$. In Fig. 1, we have for example $h_T(\zeta) = 3$ if $\zeta = abc$. The average (or expected) number of queries asked of a random text string by T is

$$\bar{h}_T = \frac{1}{q^n} \sum_{\zeta \in \Sigma^n} h_T(\zeta). \tag{1}$$

Since the number of text strings that reach the same leaf as ζ does is $q^{n-h_T(\zeta)}$, an

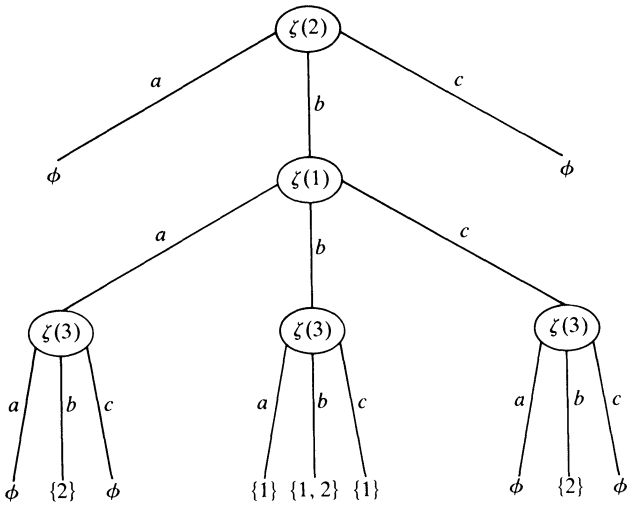


FIG. 1. *A pattern-matching algorithm for* $\Sigma = \{a, b, c\}$, $\alpha = bb$ *and* $n = 3$.

alternative form of (1) is

$$\bar{h}_T = \sum_{\text{leaf } v} \frac{d_T(v)}{q^{d_T(v)}} \tag{2}$$

where $d_T(v)$ is the distance[1] (path length) from the root to node v. The *average-case complexity* $c(\alpha, n)$ of the pattern-matching problem with respect to α and n, then, is the minimum expected number of queries asked by any algorithm. That is,

$$c(\alpha, n) = \min_{T \in \mathcal{T}(\alpha, n)} \bar{h}_T. \tag{3}$$

In [4] it was shown that, for any pattern $\alpha \in \Sigma^m$,

$$n/m \leqq c(\alpha, n) \leqq \text{const.} \cdot n \lceil \log_q (m+1) \rceil / m. \tag{4}$$

It was also conjectured in [4] that, for infinitely many m, there exists $\alpha \in \Sigma^m$ such that $c(\alpha, n) \geqq a\, n \lceil \log_q (m+1) \rceil / m$ for some constant a when n is sufficiently large. The main results of the present paper are the following theorems. The first theorem stengthens the upper bound given by formula (4) in the range $m \leqq n \leqq 2m$. The second theorem proves the conjecture mentioned above in a somewhat stronger form. In fact, Theorem 2 as stated below follows from a result (Theorem 4) proved in § 5, which implies that the lower bound in Theorem 2 actually holds even for the "best-case" complexity. (See § 5.1 for precise formulations.)

DEFINITION. For $n \geqq m > 0$, let

$$f_1(m, n) = \lceil \log_q ((n-m)/\ln (m+1) + 2) \rceil,$$

and

$$f_2(m, n) = n \lceil \log_q (m+1) \rceil / (2m).$$

[1] We shall write $d(v)$ for $d_T(v)$ when T is understood.

Define

$$f(m, n) = \begin{cases} f_1(m, n) & \text{if } m \leqq n \leqq 2m, \\ f_2(m, n) & \text{if } n > 2m. \end{cases}$$

THEOREM 1. *There exists a positive constant a_1 such that, for any $q \geqq 2$, $\alpha \in \Sigma^m$, and $n \geqq m > 0$, we have $c(\alpha, n) \leqq a_1 f(m, n)$.*

THEOREM 2. *There exists a positive constant a_2 such that, for any $q \geqq 2$ and $m > 0$, there exists a set of strings $L \subseteq \Sigma^m$ satisfying*

(i) $$|L| \geqq \left(1 - \frac{1}{m^9}\right) q^m,$$

and

(ii) *for each $\alpha \in L$, $c(\alpha, n) \geqq a_2 f(m, n)$ for all $n \geqq m$.*

In the definition of $f_i(m, n)$ above, the constants $+1$ and $+2$, as well as the ceiling function $\lceil\ \rceil$ are just to insure that $f(m, n)$ is well-defined and bounded away from zero. Indeed, as we have defined it, $f(m, n) \geqq 1$ for all $n \geqq m$. Notice also that, when $n \approx 2m$, we have $f_1(m, n) \approx f_2(m, n) \approx \lceil \log_q (m+1) \rceil$. Figure 2.1 shows the qualitative behavior of $f(m, n)$ as a function of n when m is fixed.

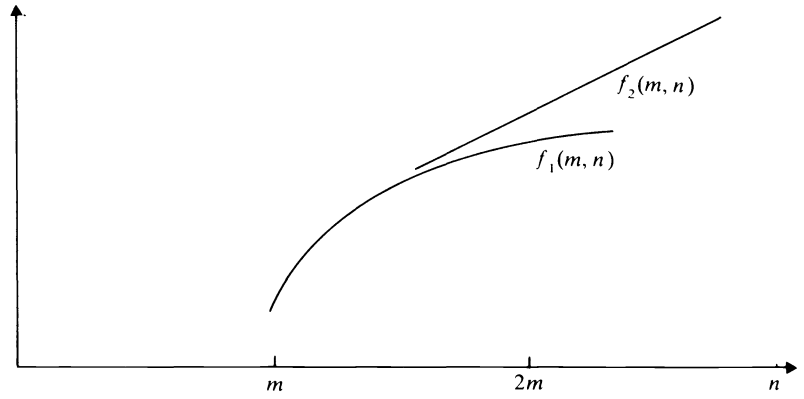


FIG. 2.1. *The behavior of $f(m, n)$ for a fixed m.*

Remark. All the constants implied in the "O", "Ω", and "θ" notations, as well as other constants used in the paper (e.g. a_1, a_2 above), are absolute constants (independent of q, n, m, etc.).

3. Analysis of a simple algorithm. In [4, § 8], a simple algorithm for pattern-matching was described and shown to have an average running time of $O(n \lceil \log_q (m+1) \rceil / m)$. This establishes the desired upper bound of Theorem 1 for $n \geqq 2m$. In fact, since $f_2(m, n) = O(f_1(m, n))$ for $(1+\varepsilon)m \leqq n \leqq 2m$ where ε is any positive constant, Theorem 1 is true as long as $n - m$ is at least a positive fraction of m. Therefore, in our discussions of upper bounds in §§ 3 and 4, we shall only be concerned with the case when $n - m$ is less than some fraction of m, say $n - m \leqq m/2$.

In § 3.1, we first show that the above-mentioned algorithm of [4] (which we shall refer to as the *Basic Algorithm* from now on) has a tight bound of $O(\lceil \log (n - m + 2) \rceil)$ for the present range $n - m \leqq m/2$. Note that this performance is still weaker than the

$O(f_1(m, n))$ bound we wish to establish. In § 3.2 we then introduce an alternative, and perhaps less obvious way for looking at the behavior of the Basic Algorithm. This new analysis will shed light on how a better algorithm may be devised. In § 4 we then present an improved algorithm and show that it achieves the time bound $O(f_1(m, n))$.

3.1 The Basic Algorithm and its analysis. We begin with a description of the Basic Algorithm from [4], slightly modified to fit our purpose.

The Basic Algorithm. Let $\alpha \in \Sigma^m$ be the pattern. For any input text string $\zeta \in \Sigma^n$, the algorithm examines ζ character by character, in the order $\zeta[m]$, $\zeta[m-1], \cdots, \zeta[1]$, $\zeta[m+1]$, $\zeta[m+2], \cdots, \zeta[n]$. The algorithm halts as soon as enough information is known to determine $A(\alpha, \zeta)$, the set of all (leftmost) positions of α's occurrences in ζ.

We will show that for the case $n-m \leqq m/2$, the Basic Algorithm only looks at $O(\lceil \log_q (n-m+2) \rceil)$ characters on the average. This analysis is a refinement of the approach used in [4] to prove the general $O(n \lceil \log_q (m+1) \rceil / m)$ bound for the same algorithm. The idea is that, for a random text string, it is unlikely that *any* occurrence of α will happen, and the Basic Algorithm can rule out that possibility after examining $O(\lceil \log_q (n-m+2) \rceil)$ characters on the average.

DEFINITION. Let $d = n-m$. For any $\zeta \in \Sigma^n$, write $\zeta = \beta_1 \zeta' \beta_2$ where $\|\beta_1\| = \|\beta_2\| = d$. The substring ζ' of ζ will be called the *prime substring* of ζ, denoted always by ζ'. Let n' be the length of ζ'. Note that $n' = n-2d = m-d \geqq m/2$ as $d \leqq m/2$.

It is easy to see that any occurrence of α in ζ must cover the prime substring ζ' (see Fig. 2.2(a)). Thus, for $A(\alpha, \zeta)$ to be nonempty, ζ' must be a substring of α. In fact, if we write $\zeta' = a_1 a_2 \cdots a_{n'}$, then for $A(\alpha, \zeta)$ to be nonempty, any segment $\zeta'[i:j] = a_i a_{i+1} \cdots a_j$ of ζ' must be a substring of $\alpha[i, j+d]$ (see Fig. 2.2(b)). Based on this observation, let us divide ζ' into consecutive segments of length r, such that $\zeta' = \delta \zeta_t \zeta_{t-1} \cdots \zeta_1$ where $\|\zeta_k\| = r$ for $1 \leqq k \leqq t$ and $\|\delta\| < r$. Then, in order for ζ to contain any occurrence of α, each ζ_k for $1 \leqq k \leqq t$ must occur in a certain substring α_k of α with $\|\alpha_k\| = \|\zeta_k\| + d = r + d$. The probability that this condition is met by all the ζ_k's of a random text string ζ is $\leqq [(d+1)/q^r]^t$. Now, what the Basic Algorithm does is to examine the substrings $\zeta_1, \zeta_2, \cdots, \zeta_t$ in sequence; hence the probability P_k that it will ever look beyond ζ_k is $\leqq [(d+1)/q^r]^k$ for $1 \leqq k \leqq t$. It follows that the average number of characters $\bar{h}$ examined by the Basic Algorithm is

$$\bar{h} \leqq r(1 + P_1 + P_2 + \cdots + P_{t-1}) + n \cdot P_t. \tag{5}$$

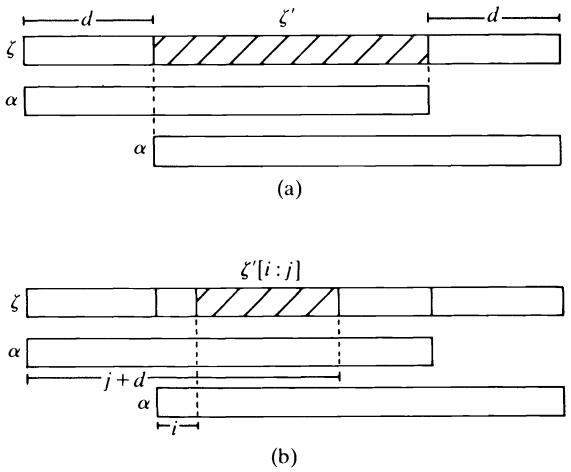


FIG. 2.2. *The prime substring ζ' of ζ relative to α.*

We now choose $r = 2\lceil \log_q (d+2) \rceil$, so that $P_k \leqq [(d+1)/(d+2)^2]^k \leqq [1/(d+2)]^k$. Then,

$$\begin{aligned} \bar{h} &\leqq 2r + n \cdot (d+2)^{-\lfloor n'/r \rfloor} \\ &\leqq 2r + m \cdot O(2^{-m/(4\lceil \log_q (d+2) \rceil)}) \\ &= 2r + O(1) \\ &= O(\lceil \log_q (d+2) \rceil). \end{aligned} \tag{6}$$

We now show that this bound is tight to within a constant factor, i.e., there exist patterns α for which $\Omega(\lceil \log_q (d+2) \rceil)$ characters on the average are examined by the Basic Algorithm. Again let $r = 2\lceil \log_q (d+2) \rceil$. We can assume that $d \geqq 4q^2$ and $r \geqq 4$. Consider a pattern $\alpha \in \Sigma^m$ which contains as a suffix the concatenation of all possible strings of length $\lfloor r/4 \rfloor$. That is, $\alpha = \eta\varphi$, where $\varphi = \varphi_u \varphi_{u-1} \cdots \varphi_1$, $u = q^{\lfloor r/4 \rfloor}$ and $\{\varphi_1, \varphi_2, \cdots, \varphi_u\}$ contains every possible string of length $\lfloor r/4 \rfloor$. Note that such α exists since, with $\lfloor r/4 \rfloor \leqq \lceil \log_q (d+2) \rceil / 2$, the total length of φ is

$$\begin{aligned} \|\varphi\| &= \lfloor r/4 \rfloor \cdot q^{\lfloor r/4 \rfloor} \\ &\leqq \frac{\lceil \log_q (d+2) \rfloor}{2} (q(d+2))^{1/2} \\ &\leqq \frac{\lceil \log_2 (d+2) \rceil}{2} \frac{d^{1/4}}{\sqrt{2}} (d+2)^{1/2} \\ &< d \end{aligned} \tag{7}$$

for $d \geqq 4q^2 \geqq 16$. For such an α, the Basic Algorithm cannot halt after examining the first block of $\lfloor r/4 \rfloor$ characters $\zeta[m], \zeta[m-1], \cdots, \zeta[m - \lfloor r/4 \rfloor + 1]$. The reason is the following: if j is the index such that $\varphi_j = \zeta[m - \lfloor r/4 \rfloor + 1 : m]$, then it is still possible for ζ to contain an occurrence of α exactly where $\zeta[m - \lfloor r/4 \rfloor + 1 : m]$ matches with φ_j (see Fig. 3). Note that the fact $\|\varphi\| < d$ is used here. We have thus shown that for such a pattern α, the algorithm must look at more than $\lfloor r/4 \rfloor$ characters.

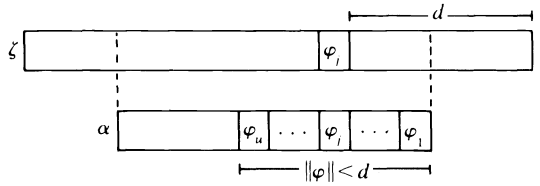


FIG. 3. *ζ may contain α between the dotted lines.*

We have demonstrated that, for any pattern $\alpha \in \Sigma^m$, and a random text string $\zeta \in \Sigma^n$, the Basic Algorithm examines an average of $O(\lceil \log_q (n-m+2) \rceil)$ characters assuming $n - m \leqq m/2$. Furthermore, there exists $\alpha \in \Sigma^m$ such that $\Omega(\lceil \log_q (n-m+2) \rceil)$ characters are examined even in the *best* case for the Basic Algorithm. Thus, to achieve the better time bound of $O(f_1(m, n))$, the algorithm has to be improved even beyond its best-case performance.

3.2. A closer look at the Basic Algorithm. In this subsection we give an alternative proof that the Basic Algorithm examines at most $O(\lceil \log_q (d+2) \rceil)$ characters on the average. This analysis may seem less straightforward than the previous one. However, it

will provide new insight into the pattern-matching process, and help motivate the improved algorithm to be presented in the next section.

Let us refer to the decision tree corresponding to the Basic Algorithm. We will be interested in those nodes where a character of the prime substring ζ' is examined, i.e., those nodes at distance $t < n'$ from the root. Initially, before any query is asked, an occurrence of α may begin at any of the positions $1, 2, \cdots, d+1$ in ζ. After the first character $\zeta[m] = a$ is examined, the feasible positions for α's occurrences in ζ is reduced from $D = \{1, 2, \cdots, d+1\}$ to $D \cap R(m, a)$ where we use $R(i, a)$ for the set $\{j | \alpha[i-j+1] = a\}$. In general, for a node v at distance $t < n'$ from the root, if $\zeta[m] = a_0$, $\zeta[m-1] = a_1, \cdots, \zeta[m-t+1] = a_{t-1}$ is the sequence of probes that led to v, then $D \cap (\cap_{k=0}^{t-1} R(m-k, a_k))$ defines the set of positions in ζ where an occurrence of α is still feasible when computation reaches this point. We shall call $D \cap (\cap_{k=0}^{t-1} R(m-k, a_k))$ the *feasible set* at v, and denote it by $F(v)$. (For $t = 0$, $F(\text{root}) = D$.) The size of $F(v)$ is called the *weight* of v, denoted by $w(v)$. We first show that the weight of an internal node v is equal to the total weights of v's sons, provided that the character examined by v is located inside the prime substring ζ'.

DEFINITION. For an internal node v with query $\zeta[i] = ?$, let $\text{son}_a(v)$ where $a \in \Sigma$ denote the succeeding node corresponding to the outcome $\zeta[i] = a$.

LEMMA 3.1. *If v examines a character inside ζ', then $F(v) = \cup_{a \in \Sigma} (F(\text{son}_a(v)))$ and $F(\text{son}_a(v)) \cap F(\text{son}_b(v)) = \phi$ for $a \neq b$.*

Proof. Let $\zeta[i] = ?$ be the query raised at v. It is easy to see that the family of subsets $R(i, a) = \{j | \alpha(i-j+1) = a\}$, for $a \in \Sigma$, forms a partition of the set $\{1, 2, \cdots, i\}$. It follows that, for any subset B of $\{1, 2, \cdots, i\}$, $\{B \cap R(i, a) | a \in \Sigma\}$ forms a partition of B. Since $i \geqq d+1$ by assumption, we have $F(v) \subseteq \{1, 2, \cdots, d+1\} \subseteq \{1, 2, \cdots, i\}$. Therefore the subsets $F(v) \cap R(i, a) = F(\text{son}_a(v))$, for $a \in \Sigma$, form a partition of $F(v)$. □

LEMMA 3.2. *If v examines a character inside ζ', then $w(v) = \Sigma_{a \in \Sigma} w(\text{son}_a(v))$.*

Proof. This follows immediately from Lemma 3.1. □

Note that Lemmas 3.1 and 3.2 may not be true if v probes outside of ζ', since we may have $F(\text{son}_a(v)) \cap F(\text{son}_b(v)) \neq \varnothing$.

Now, the probability that ζ will be examined outside of ζ' by the Basic Algorithm is quite small. In fact, it happens only if ζ' is a substring of α, which has probability less than $(d+1)/q^{n'}$. Therefore, the cost of the Basic Algorithm is

$$\bar{h} = \sum_{\text{leaf } v} \frac{d(v)}{q^{d(v)}} = \sum_{\substack{\text{leaf } v \\ d(v) \leqq n'}} \frac{d(v)}{q^{d(v)}} + \sum_{\substack{\text{leaf } v \\ d(v) > n'}} \frac{d(v)}{q^{d(v)}} \tag{8}$$

where the second term s_2 is bounded by $n(d+1)/q^{n'} = O(1)$. To study the first term s_1 in (8), we shall use the weight function w. Remember that, when we follow a path in the decision tree from the root, as soon as $w(v) = 0$ the computation terminates. This fact, together with Lemma 3.2, will allow us to bound the quantity

$$\sum_{\substack{\text{leaf } v \\ d(v) \leqq n}} d(v)/(q^{d(v)})$$

by $\log_q (w(\text{root})) + \text{const}$.

DEFINITION. Let T be a finite q-ary tree. Assume each node v (internal or leaf) of T is assigned a nonnegative integer $w(v)$ such that

(i) $w(v) = \sum_{i=1}^{q} w(\text{son}_i(v))$ for any internal node v,

(ii) if $w(v) = 0$ then r is a leaf.

We call such a T a *weighted q-ary tree*. The *initial weight* of T is defined to be $w(\text{root})$, and the *terminal weight* of T is $t(T) = \sum_{\text{leaf } v} d(v)/(q^{d(v)})$, where $d(v)$ is as usual the distance from root to node v.

DEFINITION. Let $\tau_q(W) = \text{l.u.b.}\{t(T) | T$ is any weighted q-ary tree with initial weight $W\}$. (Let $\tau_q(0) = 0$.)

THEOREM A. $\tau_q(W) = \lfloor \log_q W \rfloor + 1 + (W/q^{\lfloor \log_q W \rfloor})(1/(q-1))$, *for* $W \geqq 1$.

Proof. (The theorem is proved in Appendix A.)

COROLLARY. $\tau_q(W) \leqq \lfloor \log_q W \rfloor + 3$, *for* $W \geqq 1$.

(For a related result about optimal digital search trees with n leaves, see Knuth [5, § 6.3, ex. 37].)

Clearly now, since $w(\text{root}) = d+1$ for the decision tree of the Basic Algorithm, we have

$$s_1 = \sum_{\substack{\text{leaf } v \\ d(v) \leqq n'}} \frac{d(v)}{q^{d(v)}} \leqq \tau_q(d+1) \leqq \lfloor \log_q (d+1) \rfloor + 3. \tag{9}$$

Therefore, $\bar{h} = s_1 + s_2 \leqq \lfloor \log_q (d+1) \rfloor + O(1) = O(\lceil \log_q (d+2) \rceil)$, the same result as we showed in § 3.1.

3.3 Discussions. What have we gained by the more involved analysis in § 3.2? First, we notice that the $O(\lceil \log_q (d+2) \rceil)$ behavior is not restricted to the Basic Algorithm. Lemmas 3.1 and 3.2 are true not only for the decision tree corresponding to the Basic Algorithm, but also for an arbitrary decision tree, as long as the character examined at v lies inside ζ'. Therefore, the same analysis that led to (8) and (9) for $\bar{h}$ applies to any algorithm which first examines the substring ζ' of ζ, and halts as soon as $A(\alpha, \zeta) = \varnothing$ can be decided. Hence, the following family of algorithms all have an $O(\lceil \log_q (d+2) \rceil)$ upper bound.

GENERALIZED BASIC ALGORITHM.

1. $G \leftarrow \{d+1, d+2, \cdots, m\}$.
2. **While** $G \neq \varnothing$ **do**
   begin pick any $i \in G$ and examine $\zeta[i]$;
   if it is determined that $A(\alpha, \zeta) = \varnothing$ **then** stop;
   $G \leftarrow G - \{i\}$;
   end;
3. Examine $\zeta[i]$ for $i \in \{1, 2, \cdots, d\} \cup \{m+1, m+2, \cdots, n\}$ in any order.

Second, the successful use of $F(v)$ as a measure of progress for the computation hints on the design of a better algorithm, explicitly exploiting the present $F(v)$ to decide where to probe next. An improved algorithm based on this idea will be given in the next section.

We conclude this section by discussing the following generalization of the Basic Algorithm. Let Λ_n be the set of all permutations on $(1, 2, \cdots, n)$. Let α be any pattern of length m and $\lambda \in \Lambda_n$. We consider the following algorithm.

ALGORITHM—(λ, α). *For any input text string* $\zeta \in \Sigma^n$, *examine the characters in the order* $\zeta[\lambda(1)], \zeta[\lambda(2)], \cdots, \zeta[\lambda(n)]$. *Halt as soon as all occurrences of* α *in* ζ *can be determined.*

The Basic Algorithm is essentially the use of Algorithm—(λ, α), with a particular permutation λ for all α. We have seen that there exists α for which the Basic Algorithm examines on the average $\Omega(\lceil \log_q (n-m+2) \rceil)$ characters. Is it possible to improve over the Basic Algorithm simply by choosing a different λ? The following theorem answers this question in the negative.

THEOREM 3. *Let* $0 < m \leqq n \leqq 2m$. *For any* $\lambda \in \Lambda_n$, *there exists an* $\alpha \in \Sigma^m$ *such that Algorithm—*(λ, α) *examines an expected* $\Omega(\lceil \log_q (n-m+2) \rceil)$ *characters for a random text string in* Σ^n.

The proof of this result follows naturally from a counting technique to be developed in § 5. We shall, therefore, delay the proof to § 5.4. There we shall actually show a stronger result: for large d, most $\alpha \in \Sigma^m$ have the desired property required by Theorem 3.

4. An improved algorithm. We will construct an algorithm whose performance is $O(f_1(m, n))$ for $d = n - m \leqq m/2$. Without loss of generality, we assume $m > 16$. The crucial observation is the following. Suppose we are performing a Generalized Basic Algorithm. After a number of characters have been examined, assume we find ourselves reaching a node v with $w(v) \approx (\log_q m)/2$. Suppose that at this time the set G still has $|G| \geqq m/4$. We claim that it is possible to finish the computation, examining only $O(1)$ additional characters on the average, with a different strategy. Notice that in contrast, the analysis in § 3.2 (Theorem A) only guarantees a $O(\log_q w(r)) = O(\log_q (\log_q m))$ bound if we don't change strategy. Let us now prove the claim.

Let v be a node as described above, with $|F(v)| = w(v) \leqq (\log_q m)/2$ and the present $|G| \geqq m/4$. Consider all the positions $i \in G$ that we may choose to examine at this node v. By Lemma 3.1, any $i \in G$ would induce an (ordered) partition $\{F(v) \cap R(i, a) | a \in \Sigma\}$ of $F(v)$ into q parts. Denote this partition by $\pi(i)$. Note that there are only $q^{w(v)} \leqq \sqrt{m}$ possible partitions of $F(v)$ all together. Let us divide G into $q^{w(v)}$ equivalence classes by the induced partitions; that is, i and j in G are equivalent if and only if $\pi(i) = \pi(j)$. Since $|G| \geqq m/4$, few elements are in an equivalent class consisting of a single element. Indeed, if we arrange the equivalence classes as $E_1, E_2, \cdots, E_s, E_{s+1}, \cdots, E_{q^{w(v)}}$, so that $|E_k| \geqq 2$ if and only if $1 \leqq k \leqq s$, then we have $\sum_{k=1}^{s} |E_k| \geqq \frac{1}{4}m - \sqrt{m}$, which is positive assuming $m > 16$. Now, the key to a faster algorithm is contained in the following lemma.

LEMMA 4.1. *Let i and j be two distinct elements in E_k, where $1 \leqq k \leqq s$. If $\zeta[i] \neq \zeta[j]$, then ζ does not contain any occurrence of α.*

Proof. Assume $\zeta[i] \neq \zeta[j]$, and ζ does contain α as a substring. Let $a = \zeta[i]$, $b = \zeta[j]$, and suppose $\zeta[l]$ is a feasible starting position for pattern α. Since i and j are in G, both $\zeta[i]$ and $\zeta[j]$ lie within the prime substring ζ'. Therefore, $\alpha[i - l + 1] = a$ and $\alpha[j - l + 1] = b$. But this implies that in partition $\pi(i)$ we have $l \in F(v) \cap R(i, a)$, while in partition $\pi(j)$ we have $l \in F(v) \cap R(j, b)$. This contradicts the assumption that $\pi(i)$ and $\pi(j)$ are the same ordered partition of $F(v)$. □

As the string ζ is initially random, the probability that $\zeta[i] = \zeta[j]$ for $i \neq j$ is only $1/q$. Thus, it is advantageous to examine $\zeta[i]$ and $\zeta[j]$ for $i, j \in E_k$, which have probability $1 - 1/q$ to be different, and would thereby terminate the computation with answer $A(\alpha, \zeta) = \varnothing$. This suggests the following procedure:

PROCEDURE CLEANUP (G, F);

comment: G is the set of remaining unprobed positions in $\{d+1, d+2, \cdots, m\}$, and F is the current feasible set.

1. Examine characters $\zeta[i]$ for $i \in E_1$ one by one, then for $i \in E_2$ one by one, $\cdots$, then for $i \in E_s$ one by one. Halt as soon as it is found that $\zeta[i] \neq \zeta[j]$ with $i, j \in E_k$ for some k.
2. Examine $\zeta[i]$ for $i \in (G - \bigcup_{k=1}^{s} E_k) \cup \{1, 2, \cdots, d\} \cup \{m+1, m+2, \cdots, n\}$ in any order.

Analysis of Cleanup. Take t elements $i_1, i_2, \cdots, i_t$ of an equivalence class E_k, the probability that $\zeta[i_1] = \zeta[i_2] = \cdots = \zeta[i_t]$ is $1/q^{t-1}$. Thus the probability P that in step 1, the $t+1$st element of E_{k+1} will be examined is

$$P = \frac{1}{q^{(\sum_{j=1}^{k} |E_j| - k) + \varepsilon(t)}} \quad \text{where } \varepsilon(t) = \begin{cases} t-1 & \text{if } t \geqq 1, \\ 0 & \text{if } t = 0. \end{cases}$$

Since $\sum_{j=1}^{k} |E_j| - k \geqq \sum_{j=1}^{k} |E_j|/2$, and $\varepsilon(t) \geqq (t-1)/2$, we have

$$P \leqq \frac{1}{q^{\lceil (\sum_{j=1}^{k} |E_j| + t - 1)/2 \rceil}}.$$

Therefore, the probability that l or more characters will be read in step 1 is no more than $1/q^{\lceil (l-2)/2 \rceil}$. The cost of step 2 is bounded by $n \leqq 2m$, and it is executed with probability $\leqq 1/q^{\lceil (u-1)/2 \rceil}$, where $u = \sum_{k=1}^{s} |E_k|$. Hence the total expected cost of Cleanup is bounded by

$$\sum_{l=1}^{u} \frac{1}{q^{\lceil (l-2)/2 \rceil}} + \frac{2m}{q^{\lceil (u-1)/2 \rceil}} \leqq O(1) + \frac{2m}{q^{\lceil (1/2)(m/4 - \sqrt{m} - 1) \rceil}} = O(1). \tag{10}$$

This proves our claim. We can now state our new pattern-matching algorithm. Let $t = n' - \lceil m/4 \rceil + 1$ and $\{i_1, i_2, \cdots, i_t\}$ be any fixed subset of $\{d+1, d+2, \cdots, m\}$.

ALGORITHM PM.

1. $G \leftarrow \{d+1, d+2, \cdots, m\}$;
   $F \leftarrow \{1, 2, \cdots, d+1\}$; $j \leftarrow 0$;
2. **while** $(|F| > (\log_q m)/2) \wedge (|G| \geqq m/4)$ **do**
   begin $j \leftarrow j+1$, $i \leftarrow i_j$, examine $a = \zeta[i]$;
   if all occurrences of α can be determined **then** stop;
   $G \leftarrow G - \{i\}$;
   $F \leftarrow F \cap R(i, a)$;
3. **if** $G < m/4$ **then** examine in any order the remaining characters of ζ as needed, and halt.
4. **if** $F \leqq (\log_q m)/2$ **then** call Cleanup (G, F) to finish the computation.

To analyze the cost of Algorithm PM, let p_3, p_4 be the respective probability that steps 3, 4 will be executed, and let h_3, h_4 be upper bounds to the average number of characters examined in steps 3, 4, respectively, once they *are* executed. Then, letting h_2 be the average number of characters examined in step 2, we have

$$\bar{h}_{PM} \leqq h_2 + p_3 h_3 + p_4 h_4. \tag{11}$$

From the analysis of Cleanup, we know that $h_4 = O(1)$. The probability that step 3 is reached is bounded by the probability that the following happens (see Fig. 4):

$$(\exists j)\left[(0 \leqq j \leqq d) \wedge \left(\bigwedge_{k=1}^{t} \alpha(j + i_k) = \zeta'(i_k) \right) \right] \tag{12}$$

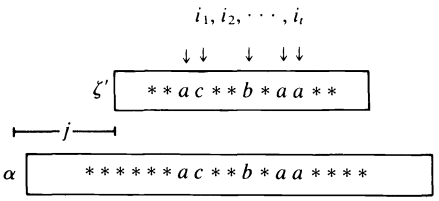


FIG. 4. *Matching ζ' with α.*

where $\{i_1, i_2, \cdots, i_t\}$ is the set of positions in substring ζ' that were examined in step 2, and $t > n' - m/4 \geqq m/4$. Therefore $p_3 \leqq (d+1)/q^t \leqq (d+1)/(q^{m/4})$, and $p_3 h_3 = O(1)$. We shall now show that

$$h_2 = O(f_1(m, n)). \tag{13}$$

This will prove $\bar{h}_{PM} = O(f_1(m, n))$, and hence Theorem 1.

Proof of (13). Let u be a positive integer. A *weighted q-ary tree with initial weight W and cut value u* is the same as in the definition of a weighted q-ary tree with initial weight W, except that condition (ii) is replaced by

(ii)′ if $w(v) < u$, then v is a leaf.

Thus for $u = 1$, it reduces to the original definition.

DEFINITION. Let $\tau_q(W, u) = \text{l.u.b.}\{t(T) | T$ is any weighted q-ary tree with initial weight W and cut value $u\}$.

THEOREM B. $\tau_q(W, u) = \tau_q(\lfloor W/u \rfloor)$.

Proof. (The theorem is proved in Appendix B.)

Now, suppose we draw a decision tree for Algorithm PM beginning from the top, but only going as far down as step 2 of the algorithm. If we designate these exit points from step 2 as "leaves", then clearly what we have is a weighted q-ary tree with initial weight $W = d+1$ and cut value $u = \lceil (\log_q m)/2 \rceil$, since condition (ii)′ is satisfied. Therefore, the cost of step 2 satisfies

$$h_2 \leqq \tau_q(d+1, \lceil (\log_q m)/2 \rceil).$$

If $d+1 < \lceil (\log_q m)/2 \rceil$, then $h_2 = 0$. Otherwise, from Theorems A, B,

$$\begin{aligned} h_2 &\leqq \log_q \left(\frac{d+1}{(\log_q m)/2} \right) + O(1) \\ &= \log_q \left(\frac{d+1}{\ln m} \right) + \log_q \ln q + O(1) \\ &= \log_q ((d+1)/\ln m) + O(1). \end{aligned}$$

Thus, in both cases,

$$h_2 = O(f_1(m, n)).$$

This completes the proof of Theorem 1.

5. Lower bounds to the complexity of pattern-matching. We shall prove Theorem 2 by showing the existence of a set of "hard" patterns for which not only there is not any algorithm with good average behavior, but in fact there is not any algorithm with good best-case behavior. In § 5.1, we define the concept of a "certificate", and carry out some preliminary reductions for the proof of Theorem 2. Section 5.2 proves a central lemma, and in § 5.3 we complete the arguments for the lower bound. In § 5.4 we prove Theorem 3 using a similar argument.

5.1 Preliminary discussions. For any l, $1 \leqq l \leqq n$, let $S_n(l)$ be the set of strings in $(\Sigma \cup \{*\})^n$ with exactly $n-l$ $*$'s. For each $\varphi \in S_n(l)$, let $I(\varphi)$ be the set of those strings in Σ^n that agree with φ except in positions where φ has $*$'s. For example, let $\Sigma = \{0, 1\}$ and $\varphi = *00*1 \in S_5(3)$, then $I(\varphi) = \{00001, 00011, 10001, 10011\}$.

Let $\alpha \in \Sigma^m$ be a pattern. A string $\varphi \in S_n(l)$ is a *certificate* (of *length* l) for α, if all elements in $I(\varphi)$ contain α in exactly the same set of positions. That is, $A(\alpha, \zeta_1) = A(\alpha, \zeta_2)$ for any $\zeta_1, \zeta_2 \in I(\varphi)$.

DEFINITION. Let $g(\alpha, n)$ be the minimum length of a certificate for α, i.e.,

$$g(\alpha, n) = \min \{l | \exists \varphi \in S_n(l) \text{ such that } \varphi \text{ is a certificate for } \alpha\}.$$

Let T be a decision tree that locates all occurrences of α in text strings from Σ^n. It is easy to see that any path in T from the root to a leaf must have length at least $g(\alpha, n)$. In fact, let $\zeta[i_1] = a_1$, $\zeta[i_2] = a_2, \cdots, \zeta[i_l] = a_l$ be the sequence of characters examined

along the path; then $\varphi \in S_n(l)$ is a certificate for α where $\varphi[i_k] = a_k$ for $1 \leqq k \leqq l$, and $\varphi[j] = *$ otherwise. Thus, no algorithm can halt before examining $g(\alpha, n)$ characters even in the best case.

LEMMA 5.1. *$c(\alpha, n) \geqq g(\alpha, n)$ for all α, n.*

We shall prove the following strengthened version of Theorem 2.

THEOREM 4. *There exists a positive constant a_2 such that, for any $q \geqq 2$ and $m > 0$, there exists a set of strings $L \subseteq \Sigma^m$ satisfying*

(I) $$|L| \geqq \left(1 - \frac{1}{m^9}\right) q^m,$$

(II) *for each $\alpha \in L$, $g(\alpha, n) \geqq a_2 f(m, n)$ for all $n \geqq m$.*

Before proceeding, we would like to make one more reduction.

LEMMA 5.2. *Let $n \geqq 2m$; then $g(\alpha, n) \geqq \lfloor n/(2m) \rfloor g(\alpha, 2m)$.*

Proof. For any string $\zeta \in \Sigma^n$, we write it as

$$\zeta = \zeta_1 \zeta_2 \cdots \zeta_{\lfloor n/2m \rfloor} \beta$$

where $|\zeta_j| = 2m$ for $1 \leqq j \leqq \lfloor n/2m \rfloor$. Similarly, we write $\varphi = \varphi_1 \varphi_2 \cdots \varphi_{\lfloor n/2m \rfloor} \eta$ for any $\varphi \in S_n(l)$. If φ is a certificate for α in Σ^n, then each φ_j must be a certificate for α_j in Σ^{2m}. (Note that the reverse may not be true.) Thus $g(\alpha, n) \geqq \lfloor n/2m \rfloor g(\alpha, 2m)$. □

This lemma allows us to reduce condition (II) of Theorem 4 to the following:

(II)′ *for each $\alpha \in L$, $g(\alpha, n) \geqq a_2 f_1(m, n)$ for $m \leqq n \leqq 2m$.*

This is so because $g(\alpha, 2m) \geqq a_2 f_1(m, 2m)$ implies $g(\alpha, n) \geqq \lfloor n/2m \rfloor g(\alpha, 2m) \geqq \lfloor n/(2m) \rfloor \cdot a_2 \lceil \log_q (m/(\ln (m+1)+2)) \rceil \geqq a_2' f_2(m, n)$ for some $a_2' > 0$.

The next two subsections are devoted to a proof of Theorem 4.

5.2 The Counting Lemma. A certificate φ for α is called a *negative certificate* if it disproves the containment of α as a substring, i.e., if $A(\alpha, \zeta) = \varnothing$ for all $\zeta \in I(\varphi)$. We first observe the fact that any certificate shorter than the pattern itself must be a negative certificate.

FACT. *Let $\alpha \in \Sigma^m$ be a pattern. If $\varphi \in S_n(l)$ is a certificate for α and $l < m$, then φ is a negative certificate for α.*

Proof. Since φ does not check as many as m non-$*$ characters, it is impossible for φ to certify the occurrence of α at any position in $\zeta \in I(\varphi)$. Therefore, it must be that $A(\alpha, \zeta) = \varnothing$. □

The next lemma is essential to the proof of Theorem 4. It says that not many patterns in Σ^m can share a common certificate which is short.

DEFINITION. For any $\varphi \in S_n(l)$, where $1 \leqq l \leqq n$, let $\mathscr{P}_m(\varphi)$ be the set of all patterns in Σ^m for which φ is a certificate. That is,

$$\mathscr{P}_m(\varphi) = \{\alpha \mid \alpha \in \Sigma^m \text{ and } \varphi \textit{ is a certificate for } \alpha\}.$$

THE COUNTING LEMMA. *Let $1 \leqq l < m \leqq n$, and $\varphi \in S_n(l)$. Then*

$$|\mathscr{P}_m(\varphi)| \leqq \left(1 - \frac{1}{q^l}\right)^{\lceil d/l^2 \rceil} \cdot q^m \quad \textit{where } d = n - m.$$

Proof. Let $1 \leqq i_1 < i_2 < \cdots < i_l \leqq n$ be the positions where φ has a non-$*$ character. For $0 \leqq j \leqq d$, define

$$B_j = \{b \mid b \in \{1, 2, \cdots, m\} \text{ and } j + b = i_t \text{ for some } 1 \leqq t \leqq l\}.$$

(See Fig. 5.) Clearly $|B_j| \leqq l$ for $1 \leqq j \leqq d$. Also, for any $\alpha \in \mathscr{P}_m(\varphi)$, since φ is a negative

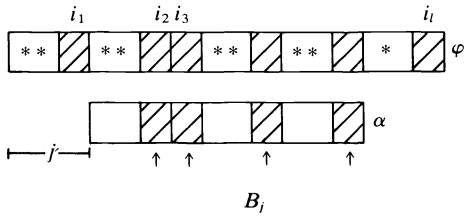


FIG. 5. *Definition of B_j for the Counting Lemma.*

certificate by Fact, there must exist an $i \in B_j$ for each j such that $\alpha[i] \neq \varphi[j+i]$. Now we show that we can find $J \subseteq \{0, 1, \cdots, d\}$, $|J| = \lceil d/l^2 \rceil$, such that $B_{j_1} \cap B_{j_2} = \varnothing$ for $j_1 \neq j_2$ in J.

We find J by a "greedy" procedure. Let $j_1 = 0$. Inductively, j_s is obtained by finding the smallest j such that B_j is disjoint from $B = B_{j_1} \cup B_{j_2} \cup \cdots \cup B_{j_{s-1}}$. We claim that this procedure allows us to find at least $\lceil d/l^2 \rceil$ such sets. In fact, we shall show that $j_s \leqq l^2(s-1)$. The key observation is that B contains at most $l(s-1)$ elements. We claim that at least one of the sets in $\mathcal{F} = \{B_0, B_1, \cdots, B_{l^2(s-1)}\}$ is disjoint from B. If not, for each r, $0 \leqq r \leqq l^2(s-1)$, let (b, i_t) be a *conflict* where $b \in B_r \cap B$ and $r + b = i_t$ for some $1 \leqq t \leqq l$. The total number of such pairs is no more than $|B| \cdot l \leqq l^2(s-1)$. But we have $l^2(s-1)+1$ sets in the family $\mathcal{F}$, and no two produce the same conflict, which is a contradiction.

To prove the lemma, consider a random string from Σ^m. For each $j \in J$, the probability that there exists some $i \in B_j$ with $\alpha[i] \neq \varphi[j+i]$ is $1 - 1/q^{|B_j|}$. Since all the sets B_j for $j \in J$ are disjoint, the probability that this holds for all j is

$$\prod_{j \in J} \left(1 - \frac{1}{q^{|B_j|}}\right) \leqq \left(1 - \frac{1}{q^l}\right)^{|J|} \leqq \left(1 - \frac{1}{q^l}\right)^{\lceil d/l^2 \rceil}.$$

Since each $\alpha \in \mathcal{P}_m(\varphi)$ must satisfy this condition, the lemma follows. □

5.3 Proof of Theorem 4. In this subsection we complete the proof of Theorem 4. Roughly, the idea is to use the Counting Lemma to bound the number of patterns in Σ^m that have any "short" certificate.

DEFINITION. Let x be a positive number such that (i) $x \geqq 256$ and (ii) $y > (\log_2 y)^{12}$ for all $y \geqq x$.

LEMMA 5.3. *Let $m + xq^4 \ln(m+1) \leqq n \leqq 2m$, $l = \lceil \frac{1}{2} \log_q ((n-m)/\ln m) \rceil$, and $\mathcal{P} = \bigcup_{\varphi \in S_n(l)} \mathcal{P}_m(\varphi)$. Then $|\mathcal{P}| \leqq (1/m^l) q^m$.*

Note. The assumption in the lemma ensures $m > 5$; thus $\ln m > 1$.

Proof. Clearly $l < m$. By the Counting Lemma, we have for each $\varphi \in S_n(l)$,

$$|\mathcal{P}_m(\varphi)| \leqq \left(1 - \frac{1}{q^l}\right)^{d/l^2} \cdot q^m.$$

Therefore,

$$\begin{aligned} |\mathcal{P}| &\leqq |S_n(l)| \cdot \left(1 - \frac{1}{q^l}\right)^{d/l^2} \cdot q^m \\ &= \binom{n}{l} q^l \cdot \left(1 - \frac{1}{q^l}\right)^{d/l^2} \cdot q^m \qquad (14) \\ &\leqq (n\ q)^{l^2} \cdot e^{(d/l^2)\ln(1-1/q^l)} \cdot q^m. \end{aligned}$$

Since $n \leqq 2m$, and $\ln(1-q^{-l}) \leqq -q^l$, (14) leads to

$$|\mathscr{P}| \leqq (2mq)^l \exp\left(-\frac{d}{l^2 q^l}\right) \cdot q^m$$

(15)

$$= q^m \cdot \exp\left(-\left(\frac{d}{l^2 q^l} - l \cdot \ln(2mq)\right)\right).$$

FACT.

$$\frac{d}{l^2 q^l} \geqq 2l \cdot \ln(2mq). \tag{16}$$

Proof. (This fact is proved in Appendix C.)

Formula (15) then implies,

$$|\mathscr{P}| \leqq q^m \cdot \exp(-l \cdot \ln m) = (1/m^l) q^m.$$

This proves the lemma. □

We now finish the proof of Theorem 4. As discussed in § 5.1, we can assume that $n \leqq 2m$. We can assume that $m \geqq xq^{20} \ln(m+1)$. Otherwise, $f(m, n) = \lceil \log_q((n-m)/\ln(m+1)+2) \rceil = O(1)$, and we can choose $L = \Sigma^m$ to satisfy the conditions in Theorem 4.

For each n, $m + xq^{20} \ln(m+1) \leqq n \leqq 2m$, let

$$\mathscr{P}^{(n)} = \bigcup_{\varphi \in S_n(l_n)} \mathscr{P}_m(\varphi), \quad \text{where } l_n = \left\lceil \frac{1}{2} \log_q\left(\frac{n-m}{\ln m}\right) \right\rceil \geqq 10. \tag{17}$$

By Lemma 5.2, we have

$$|\mathscr{P}^{(n)}| \leqq \frac{1}{m^{l_n}} q^m \leqq \frac{1}{m^{10}} q^m. \tag{18}$$

We define L as follows.

$$L = \Sigma^m - \bigcup_n \mathscr{P}^{(n)}, \tag{19}$$

where the union is taken over $m + xq^{20} \ln(m+1) \leqq n \leqq 2m$. Now we need only check that L has the properties specified in Theorem 4.

(I) $$|L| = q^m - |\bigcup_n \mathscr{P}^{(n)}| \geqq q^m - m \cdot \frac{1}{m^{10}} q^m = \left(1 - \frac{1}{m^9}\right) \cdot q^m.$$

(II)′ We shall prove, for each $\alpha \in L$, $g(\alpha, n) \geqq a_2 f(m, n)$ for all $m \leqq n \leqq 2m$, and an absolute constant a_2.

There are two cases:

(a) If $m + xq^{20} \ln(m+1) \leqq n \leqq 2m$, then $\alpha \notin \mathscr{P}^{(n)}$ by definition of L. Thus,

$$g(\alpha, n) > \left\lceil \frac{1}{2} \log_q\left(\frac{n-m}{\ln m}\right) \right\rceil \geqq a_2' \left\lceil \log_q\left(\frac{n-m}{\ln(m+1)} + 2\right) \right\rceil = a_2' \cdot f(m, n)$$

for some absolute constant a_2'.

(b) If $m \leqq n < m + xq^{20} \ln(m+1)$, then

$$f(m, n) = \left\lceil \log_q\left(\frac{n-m}{\ln(m+1)} + 2\right) \right\rceil = O(1),$$

and

$$g(\alpha, n) \geqq a_2'' \cdot f(m, n)$$

for some absolute constant a_2''. Thus, in both cases, we have verified property (II)'.

Therefore, the set L defined by (19) satisfies the conditions (I) and (II)' set in Theorem 4. This completes the proof of Theorem 4.

Remark. In the condition $|L| \geqq q^m(1-1/m^9)$ of Theorem 4, the choice of the factor $1-1/m^9$ is somewhat arbitrary. In fact, we can replace it by any factor $1-1/m^b$ where b is any fixed positive number. Then, in the proof, we need to divide cases according to whether n is greater than or less than $m + xq^{2(b+1)} \ln (m+1)$. The resulting constant a_2 in the theorem will be different.

5.4 Proof of Theorem 3. We can assume that $d = n - m > \max\{q^4, x\}$, where x is defined as in § 5.3. Otherwise the bound $\Omega(\lceil \log_q (d+2) \rceil) = \Omega(1)$, and any pattern $\alpha \in \Sigma^n$ will meet the conditions in Theorem 3.

By assumption, $m \leqq n \leqq 2m$, and $\lambda \in \Lambda_n$. Let $l = \lceil (\log_q (n-m+2))/2 \rceil$. Recall that $S_n(l)$ is the set of strings of length n over $\Sigma \cup \{*\}$ with l non-$*$ characters. Let $H \subseteq S_n(l)$ be defined by

$$H = \{\varphi \mid \varphi \in S_n(l);\ \varphi[\lambda(i)] \in \Sigma \text{ for } 1 \leqq i \leqq l, \text{ and } \varphi[j] = * \text{ for all other } j\}.$$

Clearly, there are exactly q^l elements in H, i.e., $|H| = q^l$.

Now, let $\mathscr{P}'$ be the set of patterns $\alpha \in \Sigma^m$ such that Algorithm—(λ, α) halts for some text string in less than or equal to l steps. For any $\alpha \in \mathscr{P}'$, clearly there must be a $\varphi \in H$ such that $\alpha \in \mathscr{P}_m(\varphi)$. Thus,

$$\mathscr{P}' \subseteq \bigcup_{\varphi \in H} \mathscr{P}_m(\varphi). \tag{20}$$

By the Counting Lemma, we have

$$|\mathscr{P}_m(\varphi)| \leqq (1-q^{-l})^{d/l^2} \cdot q^m.$$

Therefore,

$$|\mathscr{P}'| \leqq |H| \cdot (1-q^{-l})^{d/l^2} \cdot q^m = q^l \cdot (1-q^{-l})^{d/l^2} \cdot q^m. \tag{21}$$

Since every α in $\Sigma^m - \mathscr{P}'$ meets the conditions set in Theorem 3, we need only show that $q^l \cdot (1-q^{-l})^{d/l^2} < 1$. Now,

$$q^l \cdot (1-q^{-l})^{d/l^2} \leqq q^l \cdot \exp\left(-\frac{d}{l^2 q^l}\right). \tag{22}$$

By using the definition of l and the condition $d > q^4$, we obtain after some algebraic manipulations

$$q^l \cdot (1-q^{-l})^{d/l^2} \leqq (d+2)^{3/4} \cdot \exp\left(-\frac{(d+2)^{1/4}}{2(\log_2 (d+2))^2}\right). \tag{23}$$

The right-hand side of (23) can be shown to be less than 1 when $d \geqq x$. This proves Theorem 3.

Remark. The right-hand side of (23) is $O(\exp(-d^{1/5}))$ for large d. We have in fact shown that, for any fixed $\lambda \in \Lambda_n$, Algorithm—(λ, α) has to examine $\Omega(\lceil \log_q (d+2) \rceil)$ characters in the best case for all but a $O(\exp(-d^{1/5}))$ fraction of the patterns $\alpha \in \Sigma^m$.

An open question: Is the following statement true?

Let $0<m\leqq n\leqq 2m$. For any $\alpha\in\Sigma^m$, there exists a $\lambda\in\Lambda_n$ such that Algorithm—(λ,α) examines $O(f_1(m,n))$ characters on the average for a random text string of length n.

Appendix A. Proof of Theorem A. In this appendix, we shall prove the following theorem used in § 3.2 in the paper. For definitions and notations, see § 3.2.

THEOREM A. *Let $q\geqq 2$ be an integer. Then*

$$\tau_q(W)=\lfloor\log_q W\rfloor+1+\frac{W}{q^{\lfloor\log_q W\rfloor}}\frac{1}{q-1},\quad \textit{for } W\geqq 1. \tag{A1}$$

We first derive some properties of the function f defined below.

DEFINITION. Let $q\geqq 2$ be an integer; we define a function f by

$$\begin{aligned} &f(0)=0,\\ &f(W)=\lfloor\log_q W\rfloor+1+\frac{W}{q^{\lfloor\log_q W\rfloor}}\frac{1}{q-1},\quad \text{for } W\geqq 1. \end{aligned} \tag{A2}$$

Let $g(W)=f(W+1)-f(W)$, for all integers $W\geqq 0$.

PROPERTY 1.

$$g(W)=\frac{1}{q-1}\frac{1}{q^{\lceil\log_q(W+1)\rceil-1}}\quad \textit{for } W\geqq 0.$$

PROPERTY 2. $g(W)\geqq g(W')$ *if* $0\leqq W\leqq W'$.

Property 2 follows from Property 1, which can be verified directly.

PROPERTY 3. *The function f satisfies the following recurrence relation*:

$$f(0)=0, \tag{A3}$$

$$f(W)=1+\frac{1}{q}\sum_{i=1}^{q} f\left(\left\lfloor\frac{W+i-1}{q}\right\rfloor\right)\quad \textit{for } W\geqq 1. \tag{A4}$$

Proof of Property 3. Equation (A3) is true by definition. To prove (A4), let

$$W=tq+s,\quad \text{with } 0\leqq s<q. \tag{A5}$$

Then

$$\left\lfloor\frac{W+i-1}{q}\right\rfloor=\begin{cases} t & \text{if } 1\leqq i\leqq q-s,\\ t+1 & \text{if } q-s+1\leqq i\leqq q.\end{cases}$$

Thus, we need only prove $f(tq+s)=1+(1/q)((q-s)\cdot f(t)+s\cdot f(t+1))$, i.e.,

$$f(tq+s)=1+f(t)+\frac{s}{q}g(t). \tag{A6}$$

From the explicit forms of f and g as given in (A2) and Property 1, it is not difficult to verify (A6). This implies Property 3. □

Remark. We shall interpret (A4) as follows. Let us write $W=W_1+W_2+\cdots+W_q$ such that $|W_i-W_j|\leqq 1$ for all i,j. Then

$$f(W)=1+\frac{1}{q}\sum_{i=1}^{q} f(W_i).$$

We are now ready to prove Theorem A. Let T be any weighted q-ary tree, and T_i be the subtree rooted at $\text{son}_i(\text{root})$, $1 \leqq i \leqq q$. Then the terminal weight of T satisfies

$$t(T) = 1 + \frac{1}{q} \sum_{i=1}^{q} t(T_i).$$

This leads to the following equations:

$$\text{(A7)} \qquad \begin{aligned} &\tau_q(0) = 0, \\ &\tau_q(W) = 1 + \frac{1}{q} \max \left\{ \sum_{i=1}^{q} \tau_q(W_i) \,\middle|\, \text{integer } W_i \geqq 0, \sum_{i=1}^{q} W_i = W \right\}, \quad \text{for } W \geqq 1. \end{aligned}$$

Clearly (A7) determines $\tau_q(W)$ uniquely. Therefore, in order to prove (A1), we need only prove that $f(W)$ satisfies (A7). Because of Property 3, it suffices to prove

$$\text{(A8)} \qquad \max \left\{ \sum_{i=1}^{q} f(W_i) \,\middle|\, W_i \geqq 0, \sum_i W_i = W \right\} = \sum_{i=1}^{q} f\left(\left\lfloor \frac{w+i-1}{q} \right\rfloor \right).$$

That is, the sum $\sum_i f(W_i)$ achieves a maximum value when all the W_i differ from each other by at most 1. This can be demonstrated as follows. If, for some i and j, $W_i \geqq W_j + 2$, we make the changes $W_i \leftarrow W_i - 1$ and $W_j \leftarrow W_j + 1$. The value of $\sum_i f(W_i)$ is increased by an amount $f(W_j+1)+f(W_i-1)-f(W_i)-f(W_j) = g(W_j)-g(W_i-1)$, which is nonnegative because of Property 2. It can be shown that the value of $\sum_{i,j} |W_i - W_j|$ is decreased by at least 2 by such a transformation. Therefore, by a finite number of such transformations, all the W_i will be within 1 to each other. The value of $\sum f(W_i)$ is at least as great as the initial value before the transformations. This proves (A8), and hence Theorem A.

Appendix B. Proof of Theorem B. (See § 4 for notations.)

THEOREM B. Let $q \geqq 2$ *and* $u \geqq 1$. *Then*

$$\text{(B1)} \qquad \tau_q(W, u) = \tau_q(\lfloor W/u \rfloor), \quad \textit{for all } W \geqq 1.$$

By definition, a q-ary tree with initial weight $W < u$ and cutoff u can only consist of a single leaf. There,

$$\text{(B2)} \qquad \tau_q(W, u) = 0 \quad \text{for } 0 \leqq W < u.$$

The following facts can be established by deriving a recurrence relation on $\tau_q(W, u)$ similar to (A7), and performing some simple reductions.

$$\text{(B3)} \qquad \tau_q(W, u) = \frac{q}{q-1} \quad \text{for } u \leqq W < 2u.$$

$$\text{(B4)} \qquad \begin{aligned} \tau_q(W, u) = 1 + \frac{1}{q} \max \Bigg\{ &\sum_{i=1}^{q} \tau_q(W_i, u) \,\Bigg|\, 0 \leqq W_i < W \text{ for } 1 \leqq i \leqq q, \\ &\text{and } \sum_{i=1}^{q} W_i = W \Bigg\} \quad \text{for } W \geqq 2u. \end{aligned}$$

We shall now use (B4) in an inductive proof of formula (B1).

Consider q, u as fixed, and the induction is on variable W. By (B2) and (B3), the formula (B1) is true for $0 \leqq W < 2u$. Now, assume $W \geqq 2u$, and we have proved (B1) for all smaller values of W. We shall prove that it is also true for W.

By (B4), we have

$$\tau_q(W,u)=1+\frac{1}{q}\max\left\{\sum_{i=1}^{q}\tau_q(W_i,u)\,\middle|\,0\leqq W_i<W \text{ for } 1\leqq i\leqq q, \text{ and } \sum_{i=1}^{q}W_i=W\right\}.$$

By inductive hypothesis, $\tau_q(W_i,u)=\tau_q(\lfloor W_i/u\rfloor)$ for $1\leqq i\leqq q$. Therefore,

$$\tau_q(W,u)=1+\frac{1}{q}\max\left\{\sum_{i=1}^{q}\tau_q(\lfloor W_i/u\rfloor)\,\middle|\,0\leqq W_i<W,\sum_i W_i=W\right\}. \tag{B5}$$

We can complete the proof in two steps:

$$\text{(i)}\qquad \tau_q(W,u)\leqq\tau_q(\lfloor W/u\rfloor). \tag{B6}$$

Proof. It is not difficult to verify that $\tau_q(x)$ is a nondecreasing function of its argument. Noting that $\sum_i\lfloor W_i/u\rfloor\leqq\lfloor W/u\rfloor$, then we find

$$1+\frac{1}{q}\sum_{i=1}^{q}\tau_q(\lfloor W_i/u\rfloor)\leqq\tau_q\Big(\sum_i\lfloor W_i/u\rfloor\Big)\leqq\tau_q(\lfloor W/u\rfloor),$$

where we have used (A7) in the first step. This proves (B6) because of (B5). □

$$\text{(ii)}\qquad \tau_q(W,u)\geqq\tau_q(\lfloor W/u\rfloor). \tag{B7}$$

Proof. Let $W=tu+v$, where $0\leqq v<u$. Define

$$W_i=\begin{cases}\left\lfloor\dfrac{t+i-1}{q}\right\rfloor u & \text{for } 1\leqq i\leqq q-1,\\[2ex] \left\lfloor\dfrac{t+q-1}{q}\right\rfloor u+v & \text{for } i=q.\end{cases} \tag{B8}$$

Then

$$\lfloor W_i/u\rfloor=\left\lfloor\frac{t+i-1}{q}\right\rfloor\quad\text{for } 1\leqq i\leqq q. \tag{B9}$$

From the fact that $\sum_i W_i=W$ and (B5), we have

$$\begin{aligned}\tau_q(W,u)&\geqq 1+\frac{1}{q}\sum_{i=1}^{q}\tau_q(\lfloor W_i/u\rfloor)\\ &=1+\frac{1}{q}\sum_{i=1}^{q}\tau_q\left(\left\lfloor\frac{t+i-1}{q}\right\rfloor\right).\end{aligned} \tag{B10}$$

In Appendix A, we have shown that the right-hand side of (B10) is equal to $\tau_q(t)$. (See (A4); remember that $\tau_q(W)=f(W)$.) Therefore, (B10) leads to

$$\tau_q(W,u)\geqq\tau_q(t)=\tau_q(\lfloor W/u\rfloor).$$ □

We have now proved that $\tau(W,u)=\tau(\lfloor W/u\rfloor)$. This completes the inductive step in the proof of Theorem B.

Appendix C. Proof of formula (16). We shall prove formula (16) used in § 5.3. For easy reference, we repeat all the notations and assumptions.

Notations. $d=n-m$, $l=\lceil\frac{1}{2}\log_q((n-m)/\ln m)\rceil$. The number x is a positive number such that (i) $x\geqq 256$, and (ii) for all $y\geqq x$, $y\geqq(\log_2 y)^{12}$.

Assumptions. $q\geqq 2$, and $m\geqq d\geqq xq^4\ln(m+1)>0$.

We wish to prove:

$$\frac{d}{l^2 q} \geqq 2l \cdot \ln (2mq). \tag{C1}$$

Proof. We shall prove

$$\frac{d}{q^l \cdot \ln (2mq)} \geqq 2l^3. \tag{C2}$$

Now $l \leqq 1 + \frac{1}{2} \log_q (d/\ln m)$; hence

$$q^l \leqq q\left(\frac{d}{\ln m}\right)^{1/2}. \tag{C3}$$

Also $m \geqq d \geqq q^4$. Thus, $m \geqq 2q$ because $q \geqq 2$.

$$\ln (2mq) \leqq \ln (m^2) = 2 \ln m. \tag{C4}$$

From (C3) and (C4), we have

$$\frac{d}{q^l \cdot \ln (2mq)} \geqq \frac{d}{q(d/\ln m)^{1/2} \cdot 2 \ln m} = \frac{1}{2q}\left(\frac{d}{\ln m}\right)^{1/2}. \tag{C5}$$

Therefore, (C2) will be proved, if we can show

$$\frac{1}{2q}\left(\frac{d}{\ln m}\right)^{1/2} \geqq 2l^3,$$

i.e.,

$$\frac{d}{\ln m} \geqq 16q^2 l^6. \tag{C6}$$

Notice that, by assumption, $d/(\ln m) \geqq q^4$; hence $\log_q (d/\ln m) \geqq 4$. Therefore

$$l \leqq 1 + \frac{1}{2} \log_q \frac{d}{\ln m} \leqq \log_q \frac{d}{\ln m}. \tag{C7}$$

Because of (C7), we can prove (C6) if the following is true.

$$\frac{d}{\ln m} \geqq 16q^2 \left(\log_q \left(\frac{d}{\ln m}\right)\right)^6. \tag{C8}$$

We shall now prove (C8) to complete the proof of (C1).

By assumption, $d/(\ln m) \geqq xq^4$.

(i) Since $x \geqq 256$, we have

$$\left(\frac{d}{\ln m}\right)^{1/2} \geqq (xq^4)^{1/2} \geqq 16q^2. \tag{C9}$$

(ii) Since $d/\ln m \geqq x$, the following inequality is true. We have $d/(\ln m) \geqq (\log_2 (d/\ln m))^{12}$, which implies that

$$\left(\frac{d}{\ln m}\right)^{1/2} \geqq \left(\log_q \left(\frac{d}{\ln m}\right)\right)^6. \tag{C10}$$

It follows from (C9) and (C10) that

$$\frac{d}{\ln m} \geqq 16q^2\left(\log_q\left(\frac{d}{\ln m}\right)\right)^6.$$

This proves (C8), and hence (C1).

REFERENCES

[1] A. V. AHO AND M. J. CORASICK, *Fast pattern matching: An aid to bibliographic search*, Comm. ACM, 18 (1975), pp. 333–340.

[2] A. V. AHO, D. S. HIRSCHBERG AND J. D. ULLMAN, *Bounds on the complexity of the longest common subsequence problem*, J. Assoc. Comput. Mech., 23 (1976), pp. 1–12.

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

[4] D. E. KNUTH, J. H. MORRIS AND V. R. PRATT, *Fast pattern matching in strings*, this Journal, 6 (1977), pp. 323–350.

[5] D. E. KNUTH, *Sorting and Searching, The Art of Computer Programming*, vol. 3, Addison-Wesley, New York, 1973.

[6] R. L. RIVEST, *On the worst-case behavior of string-searching algorithms*, this Journal, 6 (1977), pp. 669–674.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0010$01.00/0

ON THE NUMBER OF COMPARISONS TO FIND THE INTERSECTION OF TWO RELATIONS*

L. J. STOCKMEYER† AND C. K. WONG†

Abstract. Given two finite sets of k-tuples whose component elements are drawn from an infinite totally ordered set, the problem of identifying the k-tuples which belong to both sets is considered. Attention is restricted to algorithms that perform pairwise comparisons on the component elements of the k-tuples. If the two sets have cardinalities m and n with $m \leqq n$ it is shown that, in the worst case,

$$(m+n)\cdot \log_2 m + (m+n-1)k$$

comparisons are sufficient and

$$\max((m+n)\cdot \log_2 m - 2.9m, (m+n-1)k)$$

comparisons are necessary. Upper and lower bounds are also given for the number of comparisons required to recognize duplicate tuples in a sequence of tuples, and to determine the lexicographic order of a sequence of tuples. In all cases, the disparity between the upper and lower bounds is at most a factor of two asymptotically.

Key words. relational algebra, computational complexity, comparison tree, sorting vectors

1. Introduction. The main purpose of this paper is to establish bounds on the number of comparisons required to find the intersection of two relations. The size of an intersection problem is specified by three parameters: m, the number of rows (or tuples) in the first relation; n, the number of rows in the second relation; and k, the number of columns (or domains) in each relation. Formally, in the *(m, n, k)-intersection problem*, the relations are viewed as two-dimensional arrays

$$A = \{a_{ij} | 1 \leqq i \leqq m, 1 \leqq j \leqq k\}$$

and

$$B = \{b_{ij} | 1 \leqq i \leqq n, 1 \leqq j \leqq k\}.$$

The individual *elements*, a_{ij} and b_{ij}, of A and B take values from some infinite totally ordered set S; there is no harm in imagining that S is the set of integers, for example. We let the k-tuples

$$\mathbf{a}_i = (a_{i1}, a_{i2}, \cdots, a_{ik}), \qquad i = 1, \cdots, m,$$

and

$$\mathbf{b}_i = (b_{i1}, b_{i2}, \cdots, b_{ik}), \qquad i = 1, \cdots, n,$$

denote the rows of A and B, respectively. The relation A (B) is said to be *admissible* iff $\mathbf{a}_i \neq \mathbf{a}_l$ ($\mathbf{b}_i \neq \mathbf{b}_l$) for all i and l with $i \neq l$; that is, A (B) does not have duplicate rows. Given admissible relations A and B, the objective is to recognize the rows of A and B that belong to both relations, or, formally, to find

$$D(A, B) = \{(i, l) | \mathbf{a}_i = \mathbf{b}_l\}.$$

We consider algorithms which find $D(A, B)$ by performing comparisons on the individual elements of A and B. Each comparison involves a pair of elements and has one of three outcomes depending on whether the value of the first element is less than, equal to, or greater than, the value of the second element. The two elements in a comparison can be taken either from the same relation or one each from the two

* Received by the editors March 8, 1978 and in revised form September 22, 1978.
† IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.

relations. Formally, the number of comparisons performed by an algorithm is measured in terms of the well-known *comparison tree* model (see Knuth [8, § 5.3.1]) generalized to permit three-branch ($<$, $=$, $>$) comparisons. Comparison trees are reviewed in § 2 of the paper.

Letting $I(m, n, k)$ denote the number of comparisons required in the worst case to solve the (m, n, k)-intersection problem, and assuming $m \leqq n$ without loss of generality, our principal results are that

$$I(m, n, k) \leqq (m+n) \cdot \log m + (m+n-1)k - m + 1, \tag{1}$$

$$I(m, n, k) \geqq \max((m+n) \cdot \log m - 2.9m, (m+n-1)k). \tag{2}$$

(In this paper, all logarithms are to the base 2.) Thus, $I(m, n, k)$ is determined to within a factor of two in the limit of large m. Also note that in the case $m = 1$, which corresponds to a search for a vector in a set of vectors, the upper and lower bounds are both nk.

Part of our motivation for studying the relational intersection problem is derived from the relational model of data introduced by Codd [2]. In this model, a data base is viewed as a collection of two-dimensional tables, called relations, where all relations are admissible. One formal language which can be used to manipulate (i.e., initialize, query, update) a relational data base is the relational algebra [2], [3] which includes, among other operations, the set-theoretic operations of union, intersection, and difference. It is clear that the number of comparisons required to find the intersection of two relations A and B is identical to the number required to find the union of A and B, and to the number required to find the difference of A and B, since, in all three cases, the information which must be extracted from A and B in order to perform the operation is precisely $D(A, B)$. The feasibility of the relational algebra in a practical system for the manipulation of data has been demonstrated [12]. Of course, in practical terms, the number of comparisons is an unrealistic measure of the complexity of an algorithm, and other measures (such as the number of page faults) would be more reasonable. However, a basic theoretical understanding of the number of comparisons required to solve the relational intersection problem is of the same nature as similar studies concerning sorting [4], [9], merging [6], selecting [1], [5], [7], [10], and performing operations on sets [11]; see also [8, § 5.3].

Another motivation is that the relational intersection problem is a multidimensional generalization of a problem which has been studied previously. In the one-dimensional ($k = 1$) version of the problem, a relation reduces to a set, the set of elements in the single column. The intersection problem for sets has been studied by Reingold [11] and Munro and Spira [9]. Assuming $m \leqq n$ again, it is known that

$$(m+n) \cdot \log m - 2.9m \leqq I(m, n, 1) \leqq (m+n) \cdot \log m + n.$$

From this lower bound on $I(m, n, 1)$ and from the upper bound (1) on $I(m, n, k)$, we see that the k-dimensional problem does not require k times as many comparisons as the 1-dimensional problem. In fact, one can interpret (1) as being (to within $O(m+k)$) the number of comparisons which are necessary to solve just the 1-dimensional problem plus one comparison for each of the $(m+n)k$ elements in the two relations.

In § 4 of the paper we consider a related problem: given a single relation A with m rows and k columns which might contain duplicate rows, partition the rows of A into equality-classes. The number of comparisons $P(m, k)$ required to solve this problem is

bounded as follows:

$$P(m, k) \leqq m \log m + (m-1)(k-1), \tag{3}$$

$$P(m, k) \geqq \max (m \log m - 1.45m, (m-1)k). \tag{4}$$

This problem is related to the projection operation in the relational algebra. Given an admissible relation R with $k+p$ columns and given k specified columns, the result of the projection operation is a relation A with k columns obtained from R by crossing out the p unspecified columns. In order that A be made admissible, the equality-classes must be found and all but one row in each class must be deleted. It is also noted that the upper and lower bounds (3) and (4) apply to the problem of determining the lexicographic order of m, k-tuples.

For both the intersection problem and the duplication-grouping problem, we also give bounds in the case that the only information obtained from a comparison is whether the two elements being compared are equal or unequal.

In § 5 of the paper we suggest other variations of these problems which might provide the basis for further research.

2. Preliminaries.

2.1. Definitions. A *comparison tree* (for the (m, n, k)-intersection problem) is a finite labeled 3-ary tree. With each nonleaf node of a comparison tree there is associated (i) a pair of elements ($a_{ij}:a_{pq}$, $b_{ij}:b_{pq}$, or $a_{ij}:b_{pq}$ for some i, j, p, q) to be compared, and (ii) three branches labeled $<$, $=$, and $>$, to be followed depending on the outcome of the comparison. Each leaf is labeled with a set of pairs of positive integers. Given a comparison tree T, each pair (A, B) of inputs determines, in the obvious way, a *path* from the root of the tree to some leaf $\lambda(T, A, B)$. The tree T *solves* the (m, n, k)-intersection problem iff $D(A, B)$ is the label of $\lambda(T, A, B)$ for all admissible inputs A and B. For example, the tree shown in Fig. 1 solves the $(1, 2, 2)$-intersection problem. In the *weak* (m, n, k)*-intersection problem*, the objective is only to determine whether or not $D(A, B)$ is empty. The tree T *solves* this weak problem iff, for all admissible (A, B), the leaf $\lambda(T, A, B)$ is labeled "empty" if $D(A, B) = \varnothing$ or "nonempty" otherwise.

For $\rho \subseteq \{<, =, >\}$, let $C_\rho(T, A, B)$ denote the number of branches labeled by an outcome in the set ρ which are traversed along the path from the root to the leaf $\lambda(T, A, B)$. Define

$$C_\rho(T) = \max_{A,B} C_\rho(T, A, B)$$

where the maximum ranges over all admissible inputs A and B. The number of comparisons with outcomes in ρ that are required in the worst-case to solve the (m, n, k)-intersection problem is the quantity

$$I_\rho(m, n, k) = \min_T C_\rho(T) \tag{5}$$

where the minimum ranges over all comparison trees which solve the (m, n, k)-intersection problem. The set ρ is introduced because at times we are interested in counting the comparisons whose outcome is equality separately from the comparisons whose outcome is either less-than or greater-than. When a set ρ is not mentioned in these notations, the set $\{<, =, >\}$ is understood. For example, the tree of Fig. 1 shows that

$$I_{\{=\}}(1, 2, 2) \leqq 3, \quad I_{\{<,>\}}(1, 2, 2) \leqq 2 \quad \text{and} \quad I(1, 2, 2) \leqq 4.$$

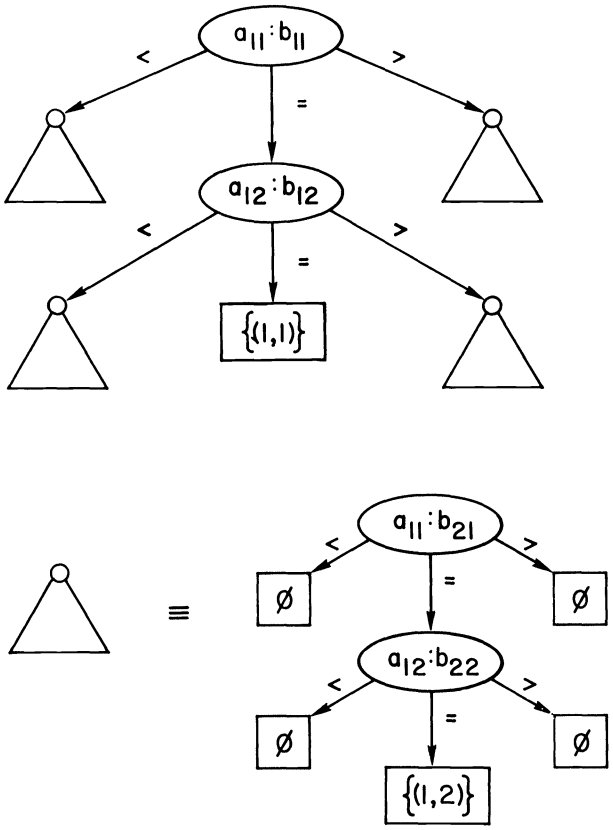


FIG. 1. *A comparison tree which solves the* (1, 2, 2)*-intersection problem.*

The number of comparisons required to solve the weak (m, n, k)-intersection problem, denoted $WI_\rho(m, n, k)$, is defined as in (5) except that the minimum ranges over trees which solve the weak problem. Obviously

$$WI_\rho(m, n, k) \leqq I_\rho(m, n, k).$$

We establish upper bounds on $I_\rho(m, n, k)$ and lower bounds on $WI_\rho(m, n, k)$. We assume $m \leqq n$ without loss of generality in stating bounds.

If v denotes a nonleaf node of a comparison tree and $\gamma \in \{<, =, >\}$, the γ-*successor* of v is the node reached from v by following the branch labeled γ. At certain points in the paper, we modify a comparison tree T to a tree T' by "fixing the outcome of a comparison to be γ" where $\gamma \in \{<, =, >\}$. Formally, if v is the node of T where the comparison is performed, then T' is obtained by replacing the subtree rooted at v by the subtree rooted at the γ-successor of v.

It is convenient to restrict attention to comparison trees which contain no intercolumn comparisons. An *intercolumn comparison* is a comparison of the form $a_{ij}:a_{pq}$, $b_{ij}:b_{pq}$, or $a_{ij}:b_{pq}$, where $j \neq q$. This restriction entails no loss of generality because of the following.

PROPOSITION 2.1. *Given any tree T which solves the* (*weak*) (m, n, k)*-intersection problem, there is a tree T' which solves the* (*weak*) (m, n, k)*-intersection problem such that T' contains no intercolumn comparisons and* $C_\rho(T') \leqq C_\rho(T)$ *for all* $\rho \subseteq \{<, =, >\}$.

Proof. If $x_{ij}: y_{pq}$ is an intercolumn comparison in T, fix the outcome to be $<$ or $>$ accordingly as $j<q$ or $j>q$. We need only observe that the modified T' gives correct answers. Any input (A, B) can be modified to an input (A', B') such that the ordering within each individual column is preserved (so that $D(A, B)=D(A', B')$), but all elements in the jth columns of A' or B' are less than all elements in the $(j+1)$th columns of these relations, for $1 \leqq j<k$. By construction, T' with input (A, B) gives the same answer as T with input (A', B'). □

We also assume that comparison trees contain no "redundant comparisons"; that is, for each node the outcome of the comparison at that node is not implicit in the comparisons that have already been performed on the path to that node.

2.2. Sorting and searching in multisets. To establish an upper bound on $I(m, n, k)$ we first need upper bounds on the number of comparisons to sort and search multisets. We include these bounds here as they may be of independent interest. The material of § 2.2 is used only in the proof of Theorem 3.2. A *multiset* is a sequence $a_1, \cdots, a_m$ where some of the a_i can have the same value. A t-tuple of positive integers $(m_1, \cdots, m_t)$ is said to be the *multiplicity vector* of the sequence $\{a_i\}$ if there are t distinct values, say $v_1<v_2<\cdots<v_t$ in the sequence, and, for $1 \leqq p \leqq t$, exactly m_p elements have the value v_p; of course $\sum_{p=1}^{t} m_p = m$. The multiset $\{a_i\}$ is *sorted* if $a_i \leqq a_{i+1}$ for $1 \leqq i<m$.

We first consider the problem of searching for an element b in a sorted multiset $a_1, \cdots, a_m$. For our purposes it is sufficient to perform conventional binary search [8, § 6.2.1, Algorithm B] ignoring the fact that $\{a_i\}$ might contain duplicate values. In the case that b equals none of the a_i, it is known that the algorithm performs at most $\lceil \log (m+1) \rceil$ $\{<, >\}$-comparisons and no $\{=\}$-comparison. In case that b is found, we have the following.

PROPOSITION 2.2. *Suppose that binary search is performed to find an element b in a sorted multiset $a_1, \cdots, a_m$ and that b is found equal to an element of multiplicity $u \geqq 1$. Then the algorithm performs at most*

$$\lceil \log (m+1) \rceil - \lceil \log (u+1) \rceil$$

$\{<, >\}$-comparisons and exactly one $\{=\}$-comparison.

Proof. Let $N(m, u)$ denote the number of $\{<, >\}$-comparisons performed. The proof of the desired upper bound on $N(m, u)$ is by induction on m. If $u \leqq m<2u$, then $N(m, u)=0$ because the first comparison of b with the "midpoint" of $\{a_i\}$ results in equality; therefore the bound holds in this case. If $m \geqq 2u$, then

$$\begin{aligned} N(m, u) &\leqq N(\lfloor m/2 \rfloor, u)+1 \\ &\leqq \lceil \log (\lfloor m/2 \rfloor +1) \rceil - \lceil \log (u+1) \rceil +1, \end{aligned}$$

the second inequality following by induction. Therefore, it is sufficient to show that

$$\lceil \log (\lfloor m/2 \rfloor +1) \rceil +1 \leqq \lceil \log (m+1) \rceil .$$

If m is even, $m=2z$ and $z \geqq 1$, then

$$\lceil \log (\lfloor m/2 \rfloor +1) \rceil +1 = \lceil \log (2z+2) \rceil = \lceil \log (2z+1) \rceil = \lceil \log (m+1) \rceil ;$$

the second equality holds because there is no integer j such that $\log (2z+1) \leqq j< \log (2z+2)$. If m is odd, $m=2z+1$, then

$$\lceil \log (\lfloor m/2 \rfloor +1) \rceil +1 = \lceil \log (2z+2) \rceil = \lceil \log (m+1) \rceil . \quad \square$$

In sorting a multiset $a_1, \cdots, a_m$ the objective is to find the multiplicity vector $(m_1, \cdots, m_t)$ and find a permutation π of $\{1, \cdots, m\}$ such that $a_{\pi(i)} \leqq a_{\pi(i+1)}$ for $1 \leqq i < m$. Munro and Spira [9] show that

$$m \log m - \sum_{p=1}^{t} m_p \log m_p + m - t + O(m)$$

comparisons are sufficient. However, when used in finding the intersection of two relations the $O(m)$ term gives rise to an $O(mk)$ term in the upper bound on $I(m, n, k)$. Therefore, a bound of this type without the $O(m)$ term is desirable. Consider binary-insertion-sort. In this algorithm, the elements $a_1, \cdots, a_m$ are inserted one by one into an initially empty list using binary search. If the elements of $\{a_i\}$ are all distinct (i.e., $t = m$ and $m_i = 1$ for all i), a known upper bound on the number of $\{<, >\}$-comparisons is

$$F(m) \overset{\text{def}}{=} \sum_{i=1}^{m} \lceil \log i \rceil,$$

and an upper bound on $F(m)$ is

$$F(m) \leqq m \log m;$$

see [8, § 5.3.1].

PROPOSITION 2.3. *When applied to a multiset $a_1, \cdots, a_m$ with multiplicity vector $(m_1, \cdots, m_t)$ binary-insertion-sort performs at most*

$$F(m) - \sum_{p=1}^{t} F(m_p)$$

$\{<, >\}$-comparisons and exactly $m - t$ $\{=\}$-comparisons.

Proof. Consider the number of comparisons required to insert a_i, where a_i is of multiplicity m_p and a_i is the jth element of multiplicity m_p to be inserted ($1 \leqq j \leqq m_p$). By Proposition 2.2, at most $\lceil \log i \rceil - \lceil \log j \rceil$ $\{<, >\}$-comparisons are required. In addition, one $\{=\}$-comparison is performed if $j > 1$. Summing these quantities for $1 \leqq i \leqq m$, $1 \leqq p \leqq t$, and $1 \leqq j \leqq m_p$ gives the desired bounds. □

It should be emphasized that the sorting algorithm need not know the multiplicity vector in advance.

3. The intersection of two relations.

3.1. The case $k = 1$. Before considering the general case of relations with k columns, it is useful to first recall the known upper and lower bounds in the case $k = 1$. The intersection problem for sets has been studied by Reingold [11], and the bounds of [11] have been sharpened by Munro and Spira [9, Corollary 3.8]. The known upper and lower bounds are tight to within $O(m + n)$.

THEOREM 3.1. *Let $m, n \geqq 1$ with $m \leqq n$.*

(a) $(m+n) \cdot \log m - 2.9m \leqq WI_{\{<,>\}}(m, n, 1) \leqq WI(m, n, 1)$.

(b) $I_{\{<,>\}}(m, n, 1) \leqq I(m, n, 1) \leqq (m+n) \cdot \log m + n$.

(c) $I_{\{=\}}(m, n, 1) = m$.

(d) $WI_{\{=\}}(m, n, 1) = 1$.

Proof. We sketch the proof for completeness; further details relating to parts (a) and (b) can be found in [9], [11]. In this proof, we elide the second subscript "1" on elements.

(a) Only the first inequality requires proof. Let T be a comparison tree that solves the weak $(m, n, 1)$-intersection problem. Let π denote a permutation of $\{1, \cdots, m\}$.

Let τ denote a map from $B = \{b_1, \cdots, b_n\}$ onto $\{1, \cdots, m\}$, and let $B_i(\tau)$ denote those elements of B that map to i under τ. (As an occupancy problem τ corresponds to a placement of n distinct balls (the b_j) into m distinct boxes (the $B_i(\tau)$) with no box remaining empty.) A pair (A, B) of sets is said to *satisfy* (π, τ) provided that, for each i, $a_{\pi(i)}$ is less than all elements in $B_i(\tau)$ and all elements in $B_i(\tau)$ are less than $a_{\pi(i+1)}$; the ordering within each $B_i(\tau)$ can be arbitrary. For each π and τ, choose some (A, B) that satisfies (π, τ) and let $\lambda(\pi, \tau)$ denote the leaf $\lambda(T, A, B)$ reached when the input is (A, B); note that $\lambda(\pi, \tau)$ must be labeled "empty". We claim that

$$\text{if} \quad (\pi, \tau) \neq (\pi', \tau') \quad \text{then} \quad \lambda(\pi, \tau) \neq \lambda(\pi', \tau'). \tag{6}$$

To verify (6), assume for contradiction that $(\pi, \tau) \neq (\pi', \tau')$ but $\lambda(\pi, \tau) = \lambda(\pi', \tau')$. Let (A, B) and (A', B') satisfy (π, τ) and (π', τ'), respectively. Since $(\pi, \tau) \neq (\pi', \tau')$, it can be seen that there is an i and j such that either $a_i < b_j$ and $a'_i > b'_j$, or $a_i > b_j$ and $a'_i < b'_j$. Therefore, in the least defined partial order that is consistent with the comparisons performed on the path to $\lambda(\pi, \tau)$, a_i and b_j are incomparable. It follows that we can find an (A'', B'') with $a''_i = b''_j$ and $\lambda(T, A'', B'') = \lambda(\pi, \tau)$. But $D(A'', B'') \neq \varnothing$, and this contradiction proves (6).

There are at least $(m!)^2 m^{n-m}$ ways of choosing π and τ, and therefore at least this many leaves $\lambda(\pi, \tau)$. For each π and τ, only comparisons whose outcome is either $<$ or $>$ are encountered on the path to $\lambda(\pi, \tau)$. Therefore, at least one of these paths must be of length at least $\log((m!)^2 m^{n-m})$, and (a) follows since $\log m! \geqq m \log m - m \log e$.

(b) Only the second inequality requires proof. First sort the set $A = \{a_1, \cdots, a_m\}$ using binary-insertion-sort. Now for $j = 1, \cdots, n$ perform a binary search to see if b_j belongs to the (now sorted) set A. As discussed in § 2.2, the number of comparisons is at most

$$F(m) + n\lceil \log(m+1) \rceil \leqq (m+n) \cdot \log m + n.$$

(c) Since the elements of A and B take on distinct values and since $m \leqq n$, the algorithm of part (b) demonstrates that $I_{\{=\}}(m, n, 1) \leqq m$. On the other hand, given any tree T that solves the $(m, n, 1)$-intersection problem, consider the leaf λ_0 reached by starting at the root and following branches according to the following rule: if the comparison at a node is $a_i : a_j$, $b_i : b_j$, or $a_i : b_j$, then traverse the branch labeled $<$, $=$, or $>$ accordingly as $i < j$, $i = j$, or $i > j$, respectively. It is easy to convince oneself that λ_0 must be labeled $\{(i, i) | 1 \leqq i \leqq m\}$. However, if fewer than m equalities are traversed on the path to λ_0 then there is an a_z $(1 \leqq z \leqq m)$ that is not involved in any comparison with an outcome of equality, so one can find an (A, B) with $\lambda(T, A, B) = \lambda_0$ and $(z, z) \notin D(A, B)$.

(d) If the weak problem is being solved, the algorithm of part (b) terminates as soon as the first equality is found. The proof that $WI_{\{=\}}(m, n, 1) \geqq 1$ is similar to part (c). □

Any comparison tree T_1 which solves the $(m, n, 1)$-intersection problem can be modified to obtain a tree T_k which solves the (m, n, k)-intersection problem. Namely, define a total order on k-tuples by lexicographic ordering, and apply the algorithm T_1 viewing each k-tuple as a single "element". Each comparison in T_1 is replaced by at most k comparisons (exactly k in the worst case) to determine the lexicographic order of the two "elements" being compared. It can be seen that $C(T_k) = k \cdot C(T_1)$. By Theorem 3.1(b), this leads to the upper bound

$$I(m, n, k) \leqq k((m+n) \cdot \log m + n).$$

A better upper bound is possible, however.

3.2. Upper bounds for general k.

THEOREM 3.2. *Let $m, n, k \geqq 1$ with $m \leqq n$.*

(a) $I_{\{<,>\}}(m, n, k) \leqq (m+n) \cdot \log m + n$.

(b) $I_{\{=\}}(m, n, k) \leqq (m+n-1)(k-1) + m$.

(c) $I(m, n, k) \leqq (m+n) \cdot \log m + (m+n-1)(k-1) + n$.

Moreover, the three upper bounds are achieved simultaneously by one comparison tree.

Proof. Actually we prove (b) and

(a') $I_{\{<,>\}}(m, n, k) \leqq F(m) + n\lceil \log (m+1)\rceil$,

(c') $I(m, n, k) \leqq F(m) + n\lceil \log (m+1)\rceil + (m+n-1)(k-1)$,

where $F(m)$ is defined in § 2.2; recall that $F(m) \leqq m \log m$. The basic strategy of the algorithm (comparison tree) is to break the problem into several subproblems, where each subproblem consists of rows of A and B that agree in their first coordinate. The subproblems involve one fewer column, so the proofs of (a'), (b), and (c') proceed by induction on k. Note that the algorithm of Theorem 3.1(b) provides the basis $k = 1$ in all three cases. The algorithm divides naturally into three stages.

Stage 1 (*Sort*). The first goal is to sort the first column of A which is the sequence $\{a_{i1}\} = a_{11}, a_{21}, \cdots, a_{m1}$. In the case $k > 1$ it is possible that certain elements in the first column have the same value. If $(m_1, \cdots, m_t)$ is the multiplicity vector of $\{a_{i1}\}$, then the number of comparisons used by binary-insertion-sort is bounded above by Proposition 2.3 as follows:

$$\{<, >\}\text{: at most } F(m) - \sum_{p=1}^{t} F(m_p); \tag{7}$$

$$\{=\}\text{: at most } m-1. \tag{8}$$

Stage 2 (*Search*). For $j = 1, \cdots, n$ the next goal is to search for b_j in the (now sorted) first column of A. For $1 \leqq p \leqq t$ let n_p be the number of elements in the first column of B that are found equal to an element of multiplicity m_p, and let n_0 be the number not found. The number of comparisons performed in this stage is bounded above by Proposition 2.2 as follows:

$$\begin{aligned}\{<, >\}\text{: at most } &\sum_{p=1}^{t} n_p(\lceil \log (m+1)\rceil - \lceil \log (m_p+1)\rceil) + n_0\lceil \log (m+1)\rceil \\ &= n\lceil \log (m+1)\rceil - \sum_{p=1}^{t} n_p\lceil \log (m_p+1)\rceil;\end{aligned} \tag{9}$$

$$\{=\}\text{: exactly } \sum_{p=1}^{t} n_p. \tag{10}$$

Stage 3 (*Recurse*). For $1 \leqq p \leqq t$, solve the resulting $(m_p, n_p, k-1)$-intersection problem.

By summing the costs from the three stages we obtain inductive upper bounds on the total cost. For part (a'), we have from (7) and (9)

$$\begin{aligned}I_{\{<,>\}}(m, n, k) \leqq\ & F(m) + n\lceil \log (m+1)\rceil - \sum (F(m_p) + n_p\lceil \log (m_p+1)\rceil) \\ &+ \sum I_{\{<,>\}}(m_p, n_p, k-1),\end{aligned} \tag{11a}$$

where the summations are taken from $p = 1$ to t. By induction we have

$$I_{\{<,>\}}(m_p, n_p, k-1) \leqq F(m_p) + n_p\lceil \log (m_p+1)\rceil.$$

(Note that this inequality is valid also for $n_p = 0$ and for $n_p < m_p$.) Substituting this inequality in (11a) proves the induction step.

For part (b), we have from (8) and (10)

$$I_{\{=\}}(m, n, k) \leqq m - 1 + n + \sum I_{\{=\}}(m_p, n_p, k-1). \tag{11b}$$

For part (c'), we sum (7)–(10) to obtain

$$I(m, n, k) \leqq (m+n-1) + F(m) + n\lceil \log (m+1)\rceil - \sum (F(m_p) + n_p \lceil \log (m_p+1)\rceil) + \sum I(m_p, n_p, k-1). \tag{11c}$$

As outlined for part (a'), the upper bounds (b) and (c') follow from these inequalities by simple calculations which are left to the reader. □

3.3. Lower bounds for general k. Lower bounds on the (m, n, k)-intersection problem are stated in the following theorem. Comparing parts (a) and (b) of Theorems 3.2 and 3.3 one sees that, when counting $\{<, >\}$-comparisons and $\{=\}$-comparisons separately, the upper and lower bounds are tight to within $O(m+n)$. When counting the total number of comparisons in part (c), the disparity between upper and lower bounds is at most a factor of two in the limit of large m.

THEOREM 3.3. *Let $m, n, k \geqq 1$ with $m \leqq n$.*

(a) $WI_{\{<,>\}}(m, n, k) \geqq (m+n) \cdot \log m - 2.9m$.

(b) $WI_{\{=\}}(m, n, k) \geqq (m+n-1)(k-1)+1$.

(c) $WI(m, n, k) \geqq \max((m+n) \cdot \log m - 2.9m, (m+n-1)k)$.

Proof. Without loss of generality we assume that comparison trees contain neither intercolumn comparisons nor redundant comparisons.

(a) This lower bound is immediate from Theorem 3.1(a) once it is noted that

$$WI_{\{<,>\}}(m, n, k) \geqq WI_{\{<,>\}}(m, n, 1).$$

Any comparison tree T_k which solves the weak (m, n, k)-intersection problem can be modified to a tree T_1 which solves the weak $(m, n, 1)$-intersection problem by fixing the outcomes of all comparisons in T_k that involve elements not in the first column; any such comparison is fixed to be $=$. Clearly $C_{\{<,>\}}(T_1) \leqq C_{\{<,>\}}(T_k)$.

(b) The proof is by induction on $m+n$. However, it is necessary for the proof to use an induction hypothesis which is stronger than the mere statement of the inequality (b) to be proved. In the stronger hypothesis, equality constraints are known between certain elements, and the comparison tree need perform correctly only for inputs which satisfy the equalities. It is convenient to view the equalities as being specified by a forest, i.e., a collection of trees; in this context, "tree" is the usual graph-theoretic notion of "connected undirected acyclic graph." The nodes of the forest correspond to elements of A and B, and the presence of an edge between two nodes specifies an equality constraint between the corresponding elements. We let the symbol a_{ij} (b_{ij}) denote the node which corresponds to element a_{ij} (b_{ij}) relying on context to resolve any ambiguity. An *edge* is a set of two distinct nodes which correspond to two elements from the same column. A *forest* is a (possibly empty) set of edges with no cycles. An element a_{ij} (b_{ij}) is *constrained* in the forest F iff a_{ij} (b_{ij}) belongs to some edge in F. A pair of inputs (A, B) *satisfies* the forest F provided that $\{x_{ij}, y_{lj}\} \in F$ implies that the two elements x_{ij} and y_{lj} are equal in A and B, where here (and subsequently) the symbols x and y denote either a or b. For example, the forest depicted in Fig. 2 specifies that $a_{11} = b_{11}$, $a_{21} = a_{31}$, and $a_{13} = a_{23} = a_{33} = b_{23}$. These eight elements are constrained. A row $\mathbf{a}_i$ ($\mathbf{b}_i$) is *j-critical* in F iff $m > 1$ ($n > 1$), a_{ij} (b_{ij}) is unconstrained in F, and a_{iz} (b_{iz}) is constrained in F for all z

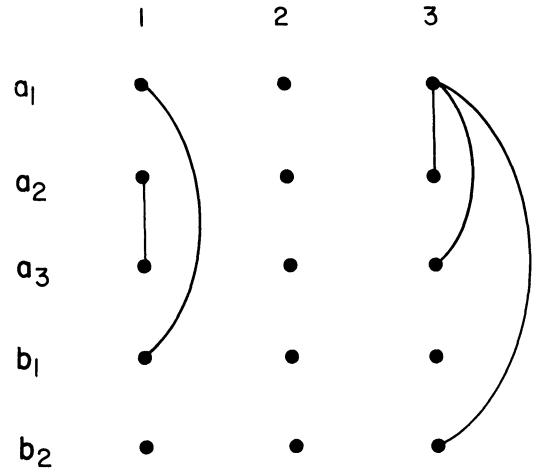


FIG. 2. *A good forest of equality constraints.*

with $1 \leqq z \leqq k$ and $z \neq j$. For example, $\mathbf{a}_1$, $\mathbf{a}_2$, and $\mathbf{a}_3$ are 2-critical in the forest of Fig. 2, while $\mathbf{b}_1$ and $\mathbf{b}_2$ are not j-critical for any j. Row $\mathbf{a}_i$ ($\mathbf{b}_i$) is *good* in F if either $m = 1$ ($n = 1$) or there exists a j, $1 \leqq j \leqq k$, such that a_{ij} (b_{ij}) is unconstrained in F. (If $m = 1$ ($n = 1$), the single row of A (B) can be fully constrained and still be good.) F is good iff all its rows are good. The *size* of F is the number of edges in F. The forest in Fig. 2 is of size 5.

If the only comparisons that an algorithm "knows" are equalities specified by a good forest, then the algorithm cannot yet make a decision as to whether or not $D(A, B) = \varnothing$. This fact is formalized as follows.

FACT 1. Let $m + n > 2$ and let F be a good forest.

(i) There is an admissible (A, B) such that (A, B) satisfies F and $D(A, B) = \varnothing$.

(ii) There is an admissible (A, B) such that (A, B) satisfies F and $D(A, B) \neq \varnothing$.

Proof. (i) Say that $n > 1$, the case $m > 1$ being analogous. Since each row $\mathbf{b}_i$ contains an unconstrained element b_{ij}, the value of that element can be chosen unequal to the values of all other elements of A and B in the jth column.

(ii) Choose A and B such that $\mathbf{a}_1 = \mathbf{b}_1$ and use the unconstrained elements in rows $i > 1$ of A and B to ensure that A and B are admissible. □

The comparison tree T *solves* the weak (m, n, k)-intersection problem *under constraint* F if, for each admissible (A, B) which satisfies F, the leaf $\lambda(T, A, B)$ is labeled "empty" iff $D(A, B) = \varnothing$.

FACT 2. Let F be a good forest and let s be the size of F. Let T be a comparison tree which solves the weak (m, n, k)-intersection problem under constraint F. Then

$$C_{\{=\}}(T) \geqq (m + n - 1)(k - 1) + 1 - s. \tag{12}$$

Note that Theorem 3.3(b) follows immediately by taking F empty and $s = 0$.

Proof of Fact 2. The proof is by induction on $m + n$.

Basis. $m = n = 1$.

In this case, (12) reduces to $C_{\{=\}}(T) \geqq k - s$. From F it is known that $\mathbf{a}_1$ and $\mathbf{b}_1$ are equal in s coordinates. Let $\lambda_=$ denote the leaf of T reached by following equality-branches starting at the root. Each comparison on the path to $\lambda_=$ is of the form $a_{1j}{:}\, b_{1j}$ for some j. If fewer than $k - s$ such comparisons are performed, it is impossible to tell whether or not $\mathbf{a}_1 = \mathbf{b}_1$.

Induction. $m + n > 2$.

First, we describe a procedure (an "adversary") which follows a path in the tree T.

A1. Set $v_0 \leftarrow$ the root of T, $F_0 \leftarrow F$, and $d \leftarrow 0$.

A2. Let $x_{ij}:y_{lj}$ be the comparison at node v_d.

A3. If either $\mathbf{x}_i$ or $\mathbf{y}_l$ is j-critical in F_d, then halt.

A4. Let F_{d+1} be the forest F_d with the edge $\{x_{ij}, y_{lj}\}$ inserted,
let v_{d+1} be the $=$-successor of v_d,
set $d \leftarrow d+1$,
and return to step A2.

Note that at the start of each execution of step A2, F_d is a good forest; therefore, by Fact 1, v_d is not a leaf. Since T is finite, this procedure eventually halts in step A3 while examining some nonleaf node v_d. For the sake of argument, say that $m > 1$, that $a_{ij}:y_{lj}$ is the comparison at node v_d, and that $\mathbf{a}_i$ is j-critical. In order to invoke the induction hypothesis, we "eliminate" row $\mathbf{a}_i$ from the problem. This is done as follows.

let v be the $>$-successor of v_d (that is, at node v it is known that $a_{ij} > y_{lj}$). Let T_d be the comparison subtree rooted at v. For each z with $1 \leqq z \leqq k$ and $z \neq j$, perform the following modifications to T_d and F_d. Let F'_z denote the largest connected component of F_d which contains a_{iz}, and let N_z be the set of nodes (i.e., elements) in F'_z *other than* a_{iz}; by the definition of j-critical, N_z contains at least one node. In F_d, remove the edges of F'_z and insert the edges of some spanning tree of N_z; note that this decreases the size of F_d by exactly one. Let α denote some member of N_z. In T_d, replace a_{iz} by α wherever a_{iz} is mentioned in a comparison. After this has been done for all $z \neq j$, further modify T_d by fixing the outcomes of all comparisons that mention a_{ij} so that a_{ij} is the larger. Let T' and F' denote T_d and F_d, respectively, after these modifications. For example, if the initial forest F is the one shown in Fig. 2 and if T first compares $b_{12}:b_{22}$ and, upon discovering that $b_{12} = b_{22}$, next compares $a_{12}:a_{22}$, then the adversary procedure halts with $d = 1$ since $\mathbf{a}_1$ is 2-critical. After $\mathbf{a}_1$ has been eliminated, F' might be as shown in Fig. 3. Furthermore, a_{11} is replaced by b_{11} in T_d, a_{13} could be replaced by a_{23}, and a_{12} is declared the larger in all comparisons which involve a_{12}.

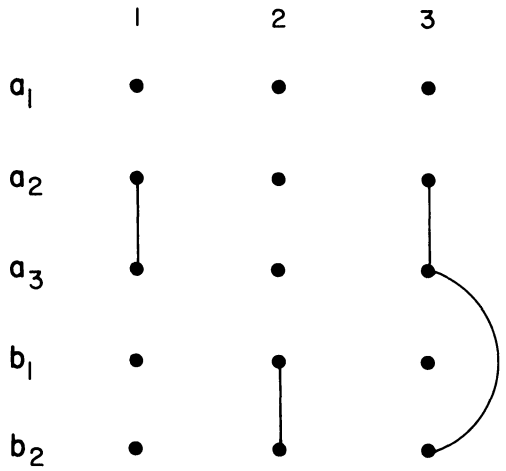


FIG. 3. *The forest of Fig.* 2 *after comparisons* $b_{12}:b_{22}$ *and* $a_{12}:a_{22}$, *and subsequent elimination of row* $\mathbf{a}_1$.

Now by construction, T' solves the weak $(m-1, n, k)$-intersection problem under constraint F'. Note that the size of F' is $s+d-k+1$ since d edges are added to F during the adversary procedure and $k-1$ edges are deleted during the elimination procedure.

By induction,

$$C_{\{=\}}(T') \geqq (m+n-2)(k-1)+1-(s+d-k+1).$$

But

$$C_{\{=\}}(T) \geqq C_{\{=\}}(T')+d,$$

so

$$C_{\{=\}}(T) \geqq (m+n-1)(k-1)+1-s$$

as was to be shown.

(c) Fact 2 is true if (12) is replaced by

$$C(T) \geqq (m+n-1)k-s. \tag{13}$$

The only difference in the proof is that now

$$C(T) \geqq C(T')+d+1;$$

that is, the $\{<, >\}$-comparison performed in going from v_d to v is also counted. Now part (c) of Theorem 3.3 is immediate from part (a) and Fact 2 with (13) replacing (12). □

One of the gaps between Theorems 3.2 and 3.3 can be closed by considering just the weak problem:

$$WI_{\{=\}}(m, n, k) = (m+n-1)(k-1)+1.$$

The algorithm described in the proof of Theorem 3.2 shows that

$$WI_{\{=\}}(m, n, k) \leqq m+n-1+\sum_{p=1}^{t} WI_{\{=\}}(m_p, n_p, k-1)$$

(cf. (11b)), and, together with $WI_{\{=\}}(m, n, 1) = 1$, this shows that $WI_{\{=\}}(m, n, k) \leqq (m+n-1)(k-1)+1$.

More bothersome than the $O(m+n)$ gaps in parts (a) and (b), though, is the factor-of-two gap when counting all comparisons in part (c). We have made no substantial progress in closing this gap. However, by considering a restricted class of comparison trees, it is possible to close the gap by improving the lower bound. Informally, a comparison tree belongs to the restricted class of column-sequential trees if the columns are inspected in increasing order $1, 2, \cdots, k$, and after column k has been inspected the answer is given. For a node v of a comparison tree which compares $x_{ij}: y_{lj}$, define $\text{col}(v) = j$. A comparison tree T is *column-sequential* iff, for all nodes v and v', if v is an ancestor of v' in the tree then $\text{col}(v) \leqq \text{col}(v')$. The comparison tree implicit in the algorithm of Theorem 3.2 is not column-sequential due to the recursive description of the algorithm. However, it is a simple matter to rearrange the comparisons so that the resulting tree is column-sequential and the upper bounds of Theorem 3.2 still hold. The following shows that, among the class of column-sequential comparison trees, this tree is optimal to within $O(m+n)$.

THEOREM 3.4. *Let $m \leqq n$, and let T be a column-sequential comparison tree which solves the weak (m, n, k)-intersection problem. Then*

$$C(T) \geqq (m+n) \cdot \log m + (m+n-1)(k-1) - 2.9m.$$

Proof. The proof is by induction on k with Theorem 3.1(a) providing the basis $k = 1$. Say then that $k > 1$. Starting at the root of T, follow equality-branches until a

node v with $\text{col}(v) \neq 1$ is reached. If at least $m+n-1$ comparisons are performed on the path from the root to v, then the proof by induction is complete. On the other hand, if fewer than $m+n-1$ are performed, then it is easy to derive a contradiction by showing that T can be led to give an incorrect answer. The key fact is that, since there are $m+n$ elements in the first column of A and B, the forest of equalities that is "known" at v must be disconnected, and, therefore, there is an i and l such that a_{i1} and b_{l1} belong to different components. Therefore, both $a_{i1} = b_{l1}$ and $a_{i1} \neq b_{l1}$ are possible. Since the subtree rooted at v contains no comparison involving elements from the first column, this confusion cannot be resolved in the subtree. The actual details of the adversary that leads T to an incorrect answer are left to the reader. □

Theorem 3.4 isolates one of the difficulties in improving the known bounds on $I(m, n, k)$. Any substantial improvement in the upper bound can be attained only by a comparison tree which is not column-sequential. One avenue for improving the lower bound would be to combine the "information theoretic" argument used to prove Theorem 3.1(a) with the "adversary" argument used to prove Theorem 3.3(b) in such a way that the combined lower bound is the sum (rather than the maximum) of the two separate lower bounds. The fact that comparison trees are, in general, not column-sequential has been one stumbling block in our attempts to combine the two arguments and has motivated the inclusion of Theorem 3.4.

3.4. Equal-unequal comparisons. In the case that the elements of a relation are drawn from a set with no natural total order, the only information obtained from a comparison is whether the values of the two elements being compared are equal or unequal. In this situation, each nonleaf node of a comparison tree has only two branches labeled $=$ and $\neq$; we refer to such trees as *equal-unequal comparison trees.* We adopt here the same definitions and notation as in § 2.1 but we affix an asterisk to these notations to indicate the restriction to equal-unequal trees. For example $I^*(m, n, k)$ is the minimum of $C(T)$ taken over all equal-unequal comparison trees that solve the (m, n, k)-intersection problem. For simplicity we here consider only the total number of comparisons, although by counting $=$-comparisons and $\neq$-comparisons separately, a development parallel to that of §§ 3.2 and 3.3 can be carried out.

Again, the case $k = 1$ has been studied previously.

THEOREM 3.5 (Reingold [11]).

$$WI^*(m, n, 1) = I^*(m, n, 1) = mn.$$

Proof. For the upper bound $I^*(m, n, 1) \leqq mn$, perform all comparisons $a_{i1} : b_{j1}$ for $1 \leqq i \leqq m$ and $1 \leqq j \leqq n$. To prove the lower bound $WI^*(m, n, 1) \geqq mn$, let T be an equal-unequal comparison tree which solves the weak $(m, n, 1)$-intersection problem. Let $\lambda_{\neq}$ be the leaf of T reached by following $\neq$-branches starting at the root. Of course, $\lambda_{\neq}$ is labeled "empty". If fewer than mn comparisons are performed on the path to $\lambda_{\neq}$, there must be an i and j such that a_{i1} is not compared with b_{j1} on this path. Therefore, there is an input (A, B) with $a_{i1} = b_{j1}$ and $\lambda(T, A, B) = \lambda_{\neq}$. □

THEOREM 3.6. (a) $I^*(m, n, k) \leqq mn + (m+n-1)(k-1)$.

(b) $WI^*(m, n, k) \geqq \max(mn, (m+n-1)k)$.

Proof. (a) The proof is by induction on k. Theorem 3.5 gives the basis $k = 1$. For $k > 1$, as in Theorem 3.2(a), the strategy is to identify rows of A and B that agree in their first coordinate. For $j = 1, \cdots, n$, perform a linear search for b_{j1} in the first column $\{a_{i1}\}$ of A. When (and if) b_{j1} is found equal to some a_{z1}, see if a_{z1} is flagged. If so, increment j and continue to the next b_{j1}. If a_{z1} is not flagged, then flag a_{z1} and continue the search through $\{a_{i1}\}$ to identify other elements of $\{a_{i1}\}$ that are equal to a_{z1}. The search need

continue only if this is the first time that a_{z1} is found equal to some b_{j1}; this is the purpose of the flag.

Say that there are t distinct values in the sequence $\{a_{i1}\}$ and that these values occur with multiplicities $m_1, \cdots, m_t$. For $1 \leqq p \leqq t$, say that n_p elements of $\{b_{j1}\}$ are found equal to an element of multiplicity m_p in $\{a_{i1}\}$. Among each group of n_p, the searching cost for one of these elements (the one found equal to an unflagged a_{z1}) is at most m comparisons; the searching cost for the remaining $n_p - 1$ is at most $m - m_p + 1$. Let n_0 be the number of elements in $\{b_{j1}\}$ that are not found in $\{a_{i1}\}$; the searching cost for each of these is m. Therefore, the total searching cost is at most

$$\sum_{p=1}^{t} (n_p - 1)(m - m_p + 1) + tm + n_0 m \leqq mn - \sum_{p=1}^{t} m_p n_p + (m + n - 1).$$

This gives the inequality

$$I^*(m, n, k) \leqq mn - \sum_{p=1}^{t} m_p n_p + (m + n - 1) + \sum_{p=1}^{t} I^*(m_p, n_p, k - 1)$$

and the proof by induction follows easily.

(b) Theorem 3.5 implies that

$$WI^*(m, n, k) \geqq mn,$$

and Theorem 3.3(c) implies that

$$WI^*(m, n, k) \geqq (m + n - 1)k. \qquad \square$$

As before, the comparison tree implicit in the proof of part (a) can be made column-sequential while preserving the upper bound of Theorem 3.6(a). The following shows that this upper bound is optimal among the class of column-sequential equal-unequal comparison trees.

THEOREM 3.7. *Let T be a column-sequential equal-unequal comparison tree which solves the weak (m, n, k)-intersection problem. Then*

$$C(T) \geqq mn + (m + n - 1)(k - 1).$$

The proof of this theorem is virtually identical to that of Theorem 3.4 and is omitted.

4. Finding duplicate rows in a relation. In the *(m, k)-duplication problem*, the input is a single $m \times k$ relation A which might contain duplicate rows, and the objective is to identify all duplications, or, more precisely, to find disjoint sets of integers $P_1, \cdots, P_z$ such that $\cup_i P_i = \{1, \cdots, m\}$, and if $i \in P_p$ and $j \in P_q$ then $\mathbf{a}_i = \mathbf{a}_j$ if and only if $p = q$. As described in the Introduction, this problem is related to the projection operation in the relational algebra. In the *weak (m, k)-duplication problem* the objective is only to determine whether or not the rows of A are all distinct. Given a comparison tree T (with three-branch comparisons $<$, $=$, $>$) which solves the (m, k)-duplication problem, let $C(T, A)$ be the number of comparisons performed by T on input A. Define

$$P(m, k) = \min_T \max_A C(T, A).$$

$WP(m, k)$ is defined similarly except that the minimum is over trees which solve the weak problem.

It is implicit in [9, Thm. 3.4] that

$$P(m, 1) = WP(m, 1) = S(m)$$

where, in the notation of Knuth [8], $S(m)$ is the number of comparisons required in the worst case to sort a sequence of m elements. Clearly $P(m, 1) \leqq S(m)$. To argue that $WP(m, 1) \geqq S(m)$, consider the case that the m input elements have distinct values. In the process of concluding that the m inputs are indeed distinct, the comparison tree must discover the entire total order of the elements; for if the order of two elements is unknown, we can make them equal and force the tree into an error. From known upper and lower bounds on $S(m)$, we have

$$m \log m - 1.45m \leqq WP(m, 1) = P(m, 1) \leqq m \log m. \tag{14}$$

For general k, techniques very similar to those of § 3 can be applied.

THEOREM 4.1. (a) $P(m, k) \leqq m \log m + (m-1)(k-1)$.

(b) $WP(m, k) \geqq \max(m \log m - 1.45m, (m-1)k)$.

Proof. (a) The (m, k)-duplication problem can be solved by performing stage 1 (sort) and stage 3 (recurse) of the algorithm described in the proof of Theorem 3.2(a). Letting $(m_1, \cdots, m_t)$ be the multiplicity vector of the first column of A, this gives

$$P(m, k) \leqq F(m) - \sum_{p=1}^{t} F(m_p) + (m-1) + \sum_{p=1}^{t} P(m_p, k-1)$$

and the upper bound (a) follows easily.

(b) The method used in the proof of Theorem 3.3(b) and (c) can be adapted very easily to show that

$$WP(m, k) \geqq (m-1)k. \tag{15}$$

Together with (14), this proves the desired lower bound. □

Note that the algorithm of part (a) discovers the entire lexicographic order of the m rows. The lower bound of part (b) is a lower bound for this problem as well. That $\log(m!)$ is a lower bound follows from the standard "information theoretic" argument; the lower bound $(m-1)k$ is, again, a straightforward adaptation of the argument used to prove Theorem 3.3(b) and (c).

Turning now to equal-unequal comparison trees, we again affix an asterisk to the notation $P(m, k)$ and $WP(m, k)$ to indicate the restriction to equal-unequal trees. First, an argument very similar to that of Theorem 3.5 shows that

$$WP^*(m, 1) = P^*(m, 1) = \binom{m}{2}. \tag{16}$$

THEOREM 4.2. (a) $P^*(m, k) \leqq \binom{m}{2} + (m-1)(k-1)$

(b) $$WP^*(m, k) \geqq \max\left(\binom{m}{2}, (m-1)k\right).$$

Proof. (a) The strategy is the same as before. It is sufficient to note that the first stage of the algorithm, identifying rows of A that agree in their first coordinate, can be realized by an equal-unequal comparison tree using at most

$$\binom{m}{2} - \sum_{p=1}^{t} \binom{m_p}{2} + (m-1)$$

comparisons.

(b) This lower bound is immediate from (15) and (16). □

5. Related questions. One naturally motivated variation of the intersection and duplication problems is to require that the values of elements in the jth column ($1 \leqq j \leqq k$) of a relation be drawn from a *finite* set S_j of known cardinality c_j. Of course, all of the previous upper bounds apply without modification when this restriction to finite sets is imposed. Unfortunately, all of the arguments which establish lower bounds (such as Theorems 3.1(a), 3.3(b), and 3.4) use the liberty of assigning an unlimited number of distinct values to elements; therefore, these arguments are no longer valid under the restriction. For the (m, k)-duplication problem, the upper bound (with three-branch comparisons) can be improved to

$$m \cdot \log (\min (m, C)) + (m-1)k$$

where $C = c_1 c_2 \cdots c_k$. Also, the upper bound for the (m, n, k)-intersection problem can be improved to

$$(m+n) \cdot \log (\min (m, n, C)) + (m+n-1)k$$

if duplicate rows are allowed in either of the two relations. However, if relations must be admissible, there is actually no improvement because then $m \leqq C$ and $n \leqq C$. It is an open question whether the restriction to finite S_j can be exploited to improve the known upper bound for the intersection problem with admissible inputs. On the other hand, it is possible that new lower bounding techniques can be found to show that no improvement is possible.

Another question which we have not pursued is the *average* number of comparisons required to solve the intersection and duplication problems. One possible definition of the average number of comparisons performed by a comparison tree is to assume that all orderings of elements are equally likely, where "ordering" is generalized to allow equalities between elements. Another formulation, suggested by the restriction to finite S_j, would define a "random" relation to be one in which each element in the jth column ($1 \leqq j \leqq k$) is chosen independently from the uniform distribution on S_j. For either of these formulations, the restriction to admissible relations might have to be abandoned for the sake of mathematical tractability.

Acknowledgment. We thank Chee K. Yap for several helpful conversations.

REFERENCES

[1] M. BLUM, R. W. FLOYD, V. R. PRATT, R. L. RIVEST, AND R. E. TARJAN, *Time bounds for selection*, J. Comput. System. Sci., 7 (1972), pp. 448–461.

[2] E. F. CODD, *A relational model of data for large shared data banks*, Comm. ACM, 13 (1970), pp. 377–387.

[3] E. F. CODD, *Relational completeness of data base sublanguages*, Courant Computer Science Symposium 6, Data Base Systems, R. Rustin, ed., Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 65–98.

[4] L. R. FORD AND S. M. JOHNSON, *A tournament problem*, Amer. Math. Monthly, 66 (1959), pp. 387–389.

[5] A. HADIAN AND M. SOBEL, *Selecting the t-th largest using binary errorless comparisons*, Tech. Report 121, Department of Statistics, University of Minnesota, Minneapolis, 1969.

[6] F. K. HWANG AND S. LIN, *A simple algorithm for merging two disjoint linearly ordered sets*, this Journal, 1 (1972), pp. 31–39.

[7] L. HYAFIL, *Bounds for selection*, this Journal, 5 (1976), pp. 109–114.

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

[9] I. MUNRO AND P. M. SPIRA, *Sorting and searching in multisets*, this Journal, 5 (1976), pp. 1–8.

[10] V. R. PRATT AND F. F. YAO, *On lower bounds for computing the i-th largest element*, 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 70–81.

[11] E. M. REINGOLD, *On the optimality of some set algorithms*, J. Assoc. Comput. Mach., 19 (1972), pp. 649–659.

[12] S. J. P. TODD, *The Peterlee relational test vehicle—a system overview*, IBM Systems Journal, 15 (1976), pp. 285–308.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0011$01.00/0

SCHEDULING INTERVAL-ORDERED TASKS*

C. H. PAPADIMITRIOU† AND M. YANNAKAKIS‡

Abstract. We show that unit execution time jobs subject to a precedence constraint whose complement is chordal can be scheduled in linear time on m processors. Generalizations to arbitrary execution times are NP-complete.

Key words. scheduling, chordal graphs, interval orders, NP-complete problems

1. Introduction. The problem of scheduling unit execution time tasks on a number of processors under arbitrary precedence constraints has been studied extensively in the past. The problem is known to be NP-complete when the number of processors is unbounded [10]. On the other hand, efficient algorithms are known for the 2-processor case [3], [2] even if release-times and deadlines are added to the problem [4]. The corresponding problems for fixed (and greater than two) numbers of processors are open. There are also results studying especially structured precedence constraints. For example, if the precedence constraints form a tree—or even forest or reverse forest—an efficient algorithm is known that works for any number of processors [8], [1]. In this note we give an efficient algorithm for a rich class of partial orders other than trees: those whose incomparability graphs are chordal. Definitions follow.

The *incomparability graph* of a partial order $P=(V, A)$ is a graph $G=(V, E)$, where $[v, u]\in E$ iff $(v, u), (u\, v)\notin A$. The complements of incomparability graphs obviously must have a *transitive orientation*, that is, an assignment of direction to their edges such that the resulting digraph is a partial order.

A *chordal graph* is one in which any circuit $[v_1, \cdots, v_k]$, $k\geqq 4$, possesses a chord, that is, an edge $[v_i, v_j]$ with $j\neq i\pm 1 \pmod k$. An *interval* graph $G=(V, E)$ is one whose nodes are closed intervals in the real line, and $[v, u]\in E$ iff $v\cap u\neq\varnothing$. It is well-known that interval graphs are chordal [7]. Finally, an *interval order* is a partial order $P=(V, A)$ where V is again a set of intervals in the real line, and $(u, v)\in A$ iff $x\in u, y\in v\Rightarrow x<y$.

Chordal graphs are known to possess many positive algorithmic properties and efficient algorithms exist for finding the maximum clique, independent set, minimum coloring, and clique cover of a chordal graph [6]. Here we shall describe an algorithm for scheduling tasks on any number of processors, subject to precedence constraints whose incomparability graphs are chordal; we will show (Lemma 3) that this class of partial orders contain exactly the interval orders. This class is incomparable to trees: Figs. 1a and 1b contain counterexamples to both inclusions (partial orders are, as usual, represented with transitive edges omitted).

It is not hard to see that the following is another interpretation of the problem described above: given a set of n unit-time tasks, each with a release time and deadline, find a feasible schedule—one in which a task never starts before its release time or finishes after its deadline—minimizing the total number of steps during which at least one processor is operating. An easy modification of our algorithm solves this problem in $O(n^2)$ time.

* Received by the editors September 14, 1978. This work was supported by NSF Grant MCS 77-1193.

† Aiken Computation Laboratory, Harvard University, Harvard, Massachusetts. Now at Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California 94720.

‡ Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey. Now at Bell Laboratories, Murray Hill, New Jersey 07974.

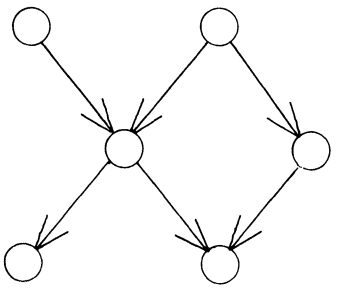


FIG. 1a

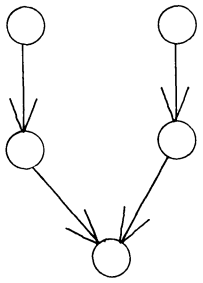


FIG. 1b

2. The Algorithm. We start by some graph-theoretic lemmas. The first paraphrases a lemma of [7]:

LEMMA 1. *An incomparability graph is chordal iff all circuits of length* 4 *have a chord.*

Proof. The *only if* direction is obvious from the definition of chordal graphs. For the *if* direction, consider an incomparability graph G in which all circuits of length 4 have chords, and yet G has a chordless circuit $c=[v_1, \cdots, v_k]$, $k>4$. Consider the subgraph $\bar{G}_c$ of $\bar{G}$ induced by c. $\bar{G}_c$ has by hypothesis a transitive orientation, since it is the subgraph of the complement of an incomparability graph. This orientation must assign to both edges $[v_i, v_j]$, $[v_i, v_{j+1}]$ either the direction towards v_i, or the direction leaving v_i because, otherwise, one of the arcs (v_j, v_{j+1}) would be added by transitivity, contrary to our assumption that $[v_j, v_{j+1}] \in G$, and hence $[v_j, v_{j+1}] \notin \bar{G}_c$. Hence, for all vertices v_i of the circuit either all edges $\{[v_i, v_j]: j \neq i, i \pm 1 \bmod k\}$ are directed to enter v_i (in which case v_i is said to be an *in-vertex*), or to leave v_i (v_i is an *out-vertex*). Now, this means that $\bar{G}_c$ is bipartite, since there are only edges joining in-vertices with out-vertices. But this is impossible when $k \geqq 5$, since for $k=5$ $\bar{G}_c$ is an odd circuit by itself, and for $k \geqq 6$ $\bar{G}_c$ always contains the triangle $[v_1, v_3, v_{k-1}]$. □

For a partial order (V, A) let $A(v)=\{u: (v, u) \in A\}$.

LEMMA 2. *If the incomparability graph of* (V, A) *is chordal then for all* $v, u \in V$, *either* $A(v) \subseteq A(u)$ *or* $A(u) \subseteq A(v)$.

Proof. Consider any two nodes $v, u \in V$. If either of $A(u)$ or $A(v)$ is empty, the lemma holds. Hence we can assume that $(u, u'), (v, v') \in A$ for $v', u' \in V$ (Fig. 2a). If no other arc involving nodes in $\{u, u', v, v'\}$ is in A, we have in the complement of (V, A) a 4-circuit $[v, u, v', u']$ (Fig. 2b), a contradiction by Lemma 1. So, at least one of the remaining 8 arcs—excluding (v', v) and (u', u) since (V, A) is acyclic—must be in A. If

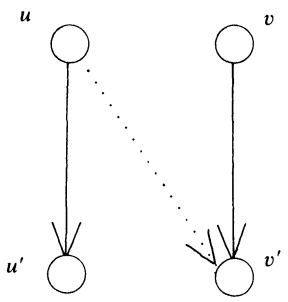


FIG. 2a

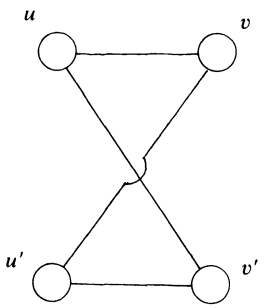


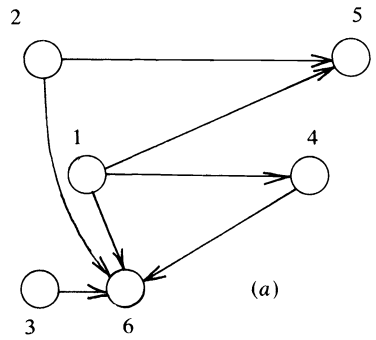
FIG. 2b

any of (v, u), (v', u'), $(v', u) \in A$, then the arc (v, u') is in A by transitivity; similarly, if any of (u, v), (u', v'), $(u', v) \in A$ then so is (u, v'). Hence either $(u, v') \in A$, or $(v, u') \in A$. Since this is true for arbitrary $u' \in A(u)$, $v' \in A(v)$, we have that either $A(u) \subseteq A(v)$ or $A(v) \subseteq A(u)$. □

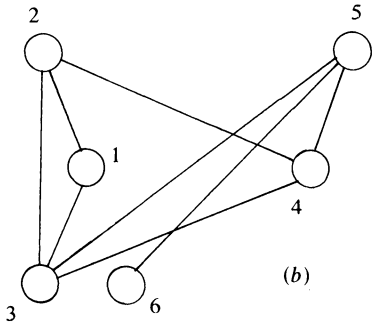
LEMMA 3. *A partial order is an interval order iff its incomparability graph is chordal.*

Proof. An interval order is the complement of a interval graph, and hence certainly of a chordal graph.

For the other direction, let (V, A) be a partial order with chordal complement. We shall construct a set of intervals on the real line (one for each node in V) whose interval order is (V, A). An interval v in V will be represented by its left and right end, $L(v)$ and $R(v)$, respectively. We examine elements of V in order of decreasing $|A(v)|$. For each $v \in V$ we let $L(v) = \max\{R(u) : (u, v) \in A\} + 1$, and $R(v) = \max\{R(u) : A(v) \subsetneq A(u)\} + 2$; by convention, $\max \varnothing = 0$. An illustration appears in Fig. 3. To complete the proof, we have to show that $R(u) < L(v)$ iff $(u, v) \in A$. If $(u, v) \in A$, then certainly $R(u) < L(v) = \max\{R(w) : (w, v) \in A\} + 1$. Conversely, if



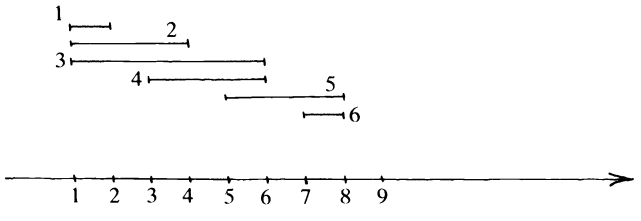
(a)



(b)

v	$A(v)$
1	{4, 5, 6}
2	{5, 6}
3	{6}
4	{6}
5	∅
6	∅

(c)



(d)

FIG. 3a–d

$R(u) < L(v)$ then $R(u) \leqq \max\{R(w) : (w, v) \in A\} = R(w)$. Since $R(u) \leqq R(w)$, it follows that it is not the case that $A(u) \subsetneq A(w)$; by Lemma 2 this means that $A(u) \supseteq A(w)$. Hence $(w, v) \in A$ implies $(u, v) \in A$. □

Our scheduling algorithm consists of the following list-scheduling [1] scheme:

1. Sort the tasks in V in nonincreasing degrees in (V, A)-equivalently, increasing right endpoints in its interval model.

2. Schedule the tasks by always scheduling next the first (in the sorted order) available task.

THEOREM 1. *The above algorithm correctly solves the unit execution time scheduling problem for interval orders* (V, A) *in* $O(|V| + |A|)$ *time.*

Proof. The degrees of elements of V can be computed in $O(|A|)$ time, and sorting can be done in $O(|V|)$ time by bucket sort. Implementing the list-scheduling in step 2 can be done in $O(|V|)$ time.

To prove correctness, assume that for some partial order (V, A) our algorithm is suboptimal, with V being as small as possible. Let $S(i)$ be the set of jobs scheduled at time i by our algorithm, and $S'(i)$ those executed at time i by the optimal schedule. By the minimality of V, $S(1) \not\supseteq S'(1)$—otherwise the tasks $V - S(1)$ would constitute a smaller counterexample to our algorithm. So, let $v \in S'(1) - S(1)$. v is maximal in (V, A), so it could be scheduled at time 1 by our algorithm. So, $|S(1)| = m$, and hence there is a job $v' \in S(1) - S'(1)$. The schedule S'' constructed by swapping v and v' in S' is also feasible (since $A(v') \supseteq A(v)$) and is also optimal. Repeating this $|S'(1) - S(1)|$ times, we end up with an optimal schedule $S^{(k)}$ with $S^{(k)}(1) = S(1)$, and this is a contradiction to the minimality of (V, A). □

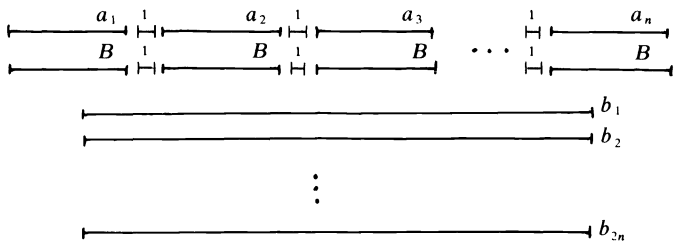
3. Discussion. We shall finally consider the question of whether more general problems can be solved by similar techniques in polynomial time. One direction of generalization would be to consider partial orders whose complements are graphs less restricted than chordal. It is not clear whether there are such "natural" classes of precedence constraints. Another possible question is, what happens if execution times different from one are allowed. Here, a polynomial algorithm becomes unlikely, since for any fixed number of processors the scheduling problem without precedence constraints becomes essentially Karp's *partition* problem [9]. Notice that the empty order is an interval order—to put it another way, the complete graph is chordal. However, what still remains a possibility is the existence of a *pseudopolynomial* algorithm for the problem with arbitrary execution times; an algorithm, that is, which runs in time polynomial in the number of tasks *and* the largest execution time. We show below that the problem of scheduling tasks with arbitrary execution times is *strongly* NP-*complete*—i.e., NP-complete even if the size of execution times is restricted to be at most polynomial of the number of tasks, see also [5]. This suggests that the existence of pseudopolynomial algorithm is extremely unlikely, exactly as NP-completeness makes the existence of polynomial algorithms improbable.

THEOREM 2. *Scheduling interval-ordered tasks on* 2 *processors is strongly* NP-*complete.*

Proof. We shall reduce *the three-way matching with integers* (3MI) problem to it. The 3MI problem is the following: Given $3n$ integers $\{a_1, a_2, \cdots, a_n, b_1, \cdots b_{2n}\}$ summing to nB can the b's be partitioned into n pairs $\{b_{i_1}, b_{j_1}\}, \cdots, \{b_{i_n}, b_{j_n}\}$, such that $a_k + b_{i_k} + b_{j_k} = B$? It is known to be strongly NP-complete [5].

Given an instance of 3MI we shall construct an interval order (V, A) and integer execution times $t(v)$ for each $v \in V$ and an integer T such that the tasks in V can be scheduled in 2 processors within T iff the original 3MI instance has a solution. The

construction is illustrated in Fig. 4, where elements of V are represented by intervals with the corresponding execution times written near them. The straightforward details of the proof are omitted. □



$T = n \cdot B + n - 1$

FIG. 4

Acknowledgment. We wish to thank Paris Kanellakis for helpful discussions.

REFERENCES

[1] E. G. COFFMAN, JR., ED., *Computer and Jobshop Scheduling Theory*, Wiley, New York, 1978.

[2] E. G. COFFMAN, JR. AND R. L. GRAHAM, *Optimal scheduling for two-processor systems*, Acta Informatica, 1, 3, pp. 200–213, 1973.

[3] M. FUJI, KASAMI, AND K. NINOMIYA, *Optimal sequencing on two equivalent processors*, SIAM J. Appl. Math, 17, 3, pp. 784–789, 1969. Erratum, 20, 1, p. 141, 1971.

[4] M. R. GAREY AND D. S. JOHNSON, *Two-processor scheduling with start-times and deadlines*, Bell Labs Memorandum, 1975.

[5] ———, *Computers & Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1978.

[6] F. GAVRIL, *Algorithms for minimum coloring, maximum clique, minimum covering by cliques and maximum independent set of a chordal graph*, this Journal, 1, pp. 180–187, 1972.

[7] P. C. GILMORE AND A. J. HOFFMAN, *A characterization of comparability graphs and of interval graphs*, Canad. J. Math., 16, pp. 539–548, 1964.

[8] T. C. HU, *Parallel sequencing and assembly line problems*, Operations Res., 9, 6, pp. 841–848, 1961.

[9] R. M. KARP, *Reducibilities among combinationatorial problems*, in Complexity of Computer Computations, R. E. Miller & J. W. Thatcher, eds., Plenum, New York, 1972.

[10] J. D. ULLMAN, *Polynomial complete scheduling problems*, Operating Systems Review, 7, 4, pp. 96–101, 1973.

SIAM J. COMPUT.
Vol. 8, No. 3, August 1979

0097-5397/79/0803-0012$01.00/0

THE COMPLEXITY OF ENUMERATION AND RELIABILITY PROBLEMS*

LESLIE G. VALIANT†

Abstract. The class of $\#P$-complete problems is a class of computationally eqivalent counting problems (defined by the author in a previous paper) that are at least as difficult as the NP-complete problems. Here we show, for a large number of natural counting problems for which there was no previous indication of intractability, that they belong to this class. The technique used is that of polynomial time reduction with oracles via translations that are of algebraic or arithmetic nature.

Key words. counting, enumeration, reliability, computational complexity, NP-completeness, permanent, matchings

1. Introduction. It is an empirical fact that for numerous combinatorial problems the detection of the existence of a solution is easy, yet no computationally efficient method is known for counting their number. The purpose of this paper is to show that for a variety of well-known problems this phenomenon can be explained. We define the class of $\#P$-complete functions as in [20]. Typical members of this class are the problems of counting the number of solutions of NP-complete problems. We show that for many natural structures that are apparently unrelated to any NP-complete structure, the problem of counting them is nevertheless $\#P$-complete. The notion of reducibility used is that of polynomial time transduction with oracles. The reductions themselves are characterized by being of an algebraic or arithmetic, rather than combinatorial nature.

The more significant problems that are shown to be $\#P$-complete are: counting perfect matchings in bipartite graphs [20]; counting trees in a directed graph; counting satisfying assignments to monotone Boolean formulae in 2-conjunctive normal form; counting maximal cliques (i.e. nonextendable complete subgraphs); and evaluating the probability that two given nodes in a probabilistic network are connected.

Many apparently difficult counting problems are probably not candidates for being $\#P$-complete. Among these are questions of the form: how many graphs of size n are there that have property X? Since for each n there is just one input, these problems correspond to NP computations over a single-letter input alphabet. We call this class $\#P_1$ and exhibit a natural problem that is complete in it. A variant of the problem has the additional curious property that while it is provably as complex as any $\#P_1$-complete problem, it is not necessarily complete itself.

The completeness results have a direct bearing on the classical study of enumerations. Note, however, that the notion of "effectively counting" that we use here is that of polynomial time computability. Since discrete probabilistic problems can usually be reformulated as counting problems, our techniques can also be applied to reliability problems, of which connectedness is a typical example. A third field of application is to "branch and bound" or search algorithms. Some simple examples of these essentially enumerate some easily detectable structure (e.g. maximal cliques). Our results suggest potential techniques for proving for such algorithms that the problem of predicting from an input the runtime of the algorithm on that input is $\#P$-complete.

2. Preliminaries. In the main we use the definitions introduced in [20]. A more general schema of definitions and some discussion of them can be found there. For background on NP-completeness see [1], [5], [10].

* Received by the editors November 18, 1977.
† Department of Computer Science, University of Edinburgh, Edinburgh, Scotland.

DEFINITION. A *counting Turing machine* is a standard nondeterministic TM with an auxiliary output device that (magically) prints in binary notation on a special tape the number of accepting computations induced by the input. It has (worst-case) *time-complexity* $f(n)$ if the longest accepting computation induced by the set of all inputs of size n takes $f(n)$ steps (when the TM is regarded as a standard nondeterministic machine with no auxiliary device).

DEFINITION. $\#P$ is the class of functions that can be computed by counting TMs of polynomial time complexity. $\#P_1$ is defined similarly for TMs with a unary input alphabet.

We denote the class of functions computed by deterministic polynomial time TMs by *FP*, and the class of predicates by *P*. For convenience we shall often identify a class of machines with the class of functions it computes. It will be assumed that objects are represented in some standard economical manner as words over an alphabet Σ (say $\{0, 1\}$). $|x|$ will denote the size of x if x is a set, and its length if x is a string. A function $f: \Sigma^* \to \Sigma^*$ (or a relation $R \subseteq \Sigma^* \times \Sigma^*$) is *polynomial bounded* iff there is a polynomial p such that for all x, $|f(x)| < p(|x|)$ (or such that $R(x, y) \Rightarrow |y| < p(|x|)$).

The notion of reduction used is one by oracles, in a similar sense to Cook [5] except that the oracles cannot only be predicates but also arbitrary polynomial bounded functions. An *oracle TM* is a TM with a query tape, an answer tape, and some working tapes. To consult the oracle the TM prints a word on the query tape, it goes into a special query state and returns an answer in unit time on the answer tape, and it enters a special answer state. An oracle TM is said to be in *P* (or *FP*, or *NP*, or $\#P$, etc.) iff for all polynomial bounded oracles it behaves like a machine in *P* (or *FP*, or *NP* or $\#P$, etc.).

If α is a class of oracle-TMs and x an appropriate function for it (i.e. polynomial bounded in the present context) then we denote the class of functions that can be computed by oracle-TMs from α with oracles for x by α^x. The class of functions that can be computed by just a single call of the oracle for any input is denoted by $\alpha^{x!}$. A problem y is *$\#P$-hard* iff $\#P \subseteq FP^y$. It is *$\#P$-complete* iff $\#P \subseteq FP^y$ and $y \in \#P$. In expressing reductions between two problems it is useful to abbreviate $x \in FP^y$ by $x \leqq y$ and $x \in FP^{y!}$ by $x \leqq! y$. Notice that both binary relations are transitive.

A relation R is *P-enumerable* iff there is a polynomial p such that for all x the set $\{y | R(x, y)\}$ can be enumerated in time $|\{y | R(x, y)\}| \cdot p(|x|)$.

3. Lemmas. Let SAT be the problem of counting the number of satisfying assignments of a Boolean formula F in conjunctive normal form, and let 3-SAT be the same problem for formulae with at most three disjuncts in each conjunct. Let TM-COMP be the problem of counting the number of accepting computations given an arbitrary polynomial time nondeterministic TM and an input for it. HAMILTONIAN CIRCUITS is the problem of counting the number of such circuits in a graph. (N.B. Here as elsewhere in the paper, graphs can be interpreted either as being directed or as being undirected, unless otherwise indicated.)

FACT 1. TM-COMP $\leqq!$ SAT.

Proof. The transformation of Cook as given in [5] or [1] establishes this. □

FACT 2. SAT $\leqq!$ 3SAT.

Proof. *Suppose* F has a clause with $i \geqq 4$ literals. If we replace in the clause any two of these literals (e.g. x_j, $\bar{x}_k$) by a new variable (say y) and conjoin F with the CNF formula for $(x_j \vee \bar{x}_k) \equiv y$ (which has clearly at most 3 literals per clause) then the new formula will have the same number of solutions as F. The result follows by induction. □

FACT. 3. SAT $\leqq!$ HAMILTONIAN CIRCUITS.

Proof. A direct reduction that preserves the number of solutions is given in [19] for both the directed and undirected cases. □

FACT 4. *Given an $n \times n$ integer matrix with each entry bounded in magnitude by 2^m the determinant and inverse can be computed in time polynomial in n and m.*

Proof. Perform Gaussian elimination with arithmetic modulo 2^k, the smallest power of 2 greater than $2^{mn} \cdot n!$. As pivot always choose a number that has the fewest factors of 2 in its prime decomposition. This ensures that eliminating with respect to that row will multiply the matrix by various numbers coprime with 2^k. When an upper diagonal matrix is achieved the value of the determinant can be computed by dividing the products of the diagonal elements by the product of the multipliers.

The inverse can be computed as a rational number by the determinental rule for example. □

In the remaining two facts the *size* of a rational number will be the sum of the lengths of its numerator and denominator (assumed coprime).

FACT 5. (i) *If $p(x)$ is an n-th degree polynomial and its value is known at each of the rational points $x_1, \cdots, x_{n+1}$, all of size at most m, then the coefficients of p can be deduced in time polynomial in n, m and the size of the largest value.*

(ii) *If the value of*

$$p(x, y) = \sum_{i=1}^{q} \sum_{j=1}^{r} p_{ij} x^i y^j$$

is known for all pairs of points $x = x_h$, $y = y_k$ for $1 \leqq h \leqq q+1$ and $1 \leqq k \leqq r+1$ (all points being of size bounded by m) then the value of each p_{ij} can be deduced in time polynomial in q, r, m and the size of the largest value.

Proof. (i) Let X be the $(n+1) \times (n+1)$ matrix with $X_{ij} = x_i^{j-1}$. Then X is Vandermonde and has an inverse, which, by Fact 4, can be computed fast. But if $\mathbf{p}$ is the vector of coefficients of p and $p(\mathbf{x})$ the vector of values at the $n+1$ points then $p(\mathbf{x}) = X\mathbf{p}$ and hence $\mathbf{p} = X^{-1}p(\mathbf{x})$.

(ii) For each value x_h use (i) to compute the value of $\sum p_{ij} x_h^i$ for each j. Then use (i) again to deduce each p_{ij}. □

FACT 6. *Let $\{a_i\}$ and $\{b_{ij}\}$ be sets of positive integers bounded by $A > 2$. If the value of any one of the following functions is known at a suitable point x_0, or (x_0, y_0), then the value of each a_i, or each b_{ij} can be deduced in time polynomial in n, m and in the sizes of x_0, y_0 and the value.*

(i) $\sum_{0}^{n} a_i x^i$ *if* $x_0 \geqq A^2$ *or* $0 < x_0 \leqq A^{-2}$,

(ii) $\sum_{0}^{n} a_i x^i (1-x)^{n-i}$ *if* $0 < x_0 \leqq A^{-2}$,

(iii) $\sum_{i=0}^{n} \sum_{j=0}^{m} b_{ij} x^i (1-x)^{n-i} y^j (1-y)^{m-j}$ *if* $x_0 = A^{-2}$ *and* $O < y_0 < A^{-3n}$.

Proof. (i) If $x_0 \geqq A^2$ then for each j

$$\sum_{i=0}^{j-1} a_i x_0^i < A(x_0^{j-1}(1 + A^{-2} + A^{-4} + \cdots))$$

$$< 3A \cdot x_0^{j-1}/2.$$

Hence $x_0^j > \sum_0^{j-1} a_i x_0^i$. It follows that $a_n, a_{n-1}, \cdots, a_0$ can be computed in succession.

The alternative case follows similarly by examining the reciprocal of x_0.

(ii) Let $t = 1/x_0$, so that $t > A^2$. Then

$$\sum_{j+1}^{n} a_i x_0^i (1-x_0)^{n-i} = t^{-n} \cdot \sum_{j+1}^{n} a_i (t-1)^{n-i}$$
$$< 3t^{-n} \cdot A(t-1)^{n-j}/2.$$

Hence $(t-1)^{n-j} > \sum_{j+1}^{n} a_i (t-1)^{n-i}$, and $a_0, a_1, \cdots, a_n$ can be computed in succession.

(iii) The substitution for x_0 gives the value

$$\sum_j \sum_i b_{ij} (A^2-1)^{n-i} y^j (1-y)^{m-j}.$$

Applying the argument of (ii) to the outer summation gives $\sum b_{ij}(A^2-1)^{n-i}$. Applying it again gives the coefficients. □

FACT 7. *Suppose $R(x, y)$ is a polynomial bounded relation and the sets $R = \{(x, y) | R(x, y)\}$ and $R^1 = \{(x, \alpha) | \exists \beta \in \{0, 1\}^*$ s.t. $R(x, \alpha\beta)\}$ are both polynomial time recognizable. Then R is P-enumerable.*

Proof. A call, Enumerate $(x, \varnothing)$, of the following recursive procedure clearly suffices:

procedure Enumerate (x, α).
begin if $R(x, \alpha)$ **then** output α;
if $R^1(x, \alpha)$ **then** Enumerate $(x, \alpha 0)$ **and** Enumerate $(x, \alpha 1)$;
end. □

4. Some #P-complete problems. Unless otherwise stated we shall denote a graph, whether directed or undirected, by $G = (V, E)$ where $V = (u_1, \cdots, u_n)$ and $E \subseteq (V \times V)$. $G_1 = (V_1, E_1)$ is a *subgraph* of G if $V_1 = V$ and $E_1 \subseteq E$. By F we shall denote a Boolean formula in conjunctive normal form, with clauses $c_1, \cdots, c_r$ and variables $X = \{x_1, \cdots, x_n\}$. F is *monotone* if no variable is negated. It is in *k-form* if each conjunct has at most k disjuncts. $\mathbf{x}$ will denote an n-tuple from $\{0, 1\}^n$. $F(\mathbf{x})$ denotes the truth value of F when the ith component of $\mathbf{x}$ is substituted as the truth value of x_i.

We shall first specify a list of counting problems (2–14). As can be verified easily each one is in $\#P$. Also, most of them are P-enumerable by virtue of Fact 7.

1. PERMANENT
   Input: Integer matrix A.
   Output: $\sum_\sigma \prod_{i=1}^{n} A_{i,\sigma(i)}$ summed over all permutations σ on $\{1, \cdots, n\}$.
2. PERFECT MATCHINGS
   Input: Bipartite graph G with $2n$ nodes.
   Output: Number of perfect matchings (i.e. sets of n edges such that no pair of edges has a common node).
3. MONOTONE PRIME IMPLICANTS
   Input: Monotone F in 2-form.
   Output: $|\{Y \subseteq X | (\bigwedge_{x \in Z} x \Rightarrow F)$ holds for $Z = Y$ but not for any $Z \subsetneq Y\}|$.
4. MINIMAL VERTEX COVER
   Input: G.
   Output: $|\{V' \subseteq V |$ "$(u, v) \in E \Rightarrow u \in A$ or $v \in A$" holds for $A = V'$ but not for any $A \subsetneq V'\}|$.
5. MAXIMAL CLIQUES
   Input: G
   Output: $|\{V' \subseteq V |$ "$u, v \in A \Rightarrow (u, v) \in E$" holds for $A = V'$ but not for any $A \supsetneq V'\}|$.

6. IMPERFECT MATCHINGS
   Input: Bipartite graph with $2n$ nodes.
   Output: Number of matchings of any size.
7. MONOTONE 2-SAT
   Input: $F = c_1 \wedge c_2 \wedge \cdots \wedge c_r$ where $c_i = (y_{i1} \vee y_{i2})$ and $y_{ij} \in X$.
   Output: $|\{\mathbf{x} | F(\mathbf{x}) \text{ true}\}|$.
8. SAT′
   Input: As for 7.
   Output: $|\{(\mathbf{x}, \mathbf{t}) | \mathbf{t} = (t_1, \cdots, t_n) \in \{1, 2\}^n$; for $1 \leqq i \leqq r$, $\mathbf{x}$ makes $y_{i,k}$ true for $k = t_i\}|$.
9. SAT″
   Input: As in 7.
   Output: $|\{(\mathbf{x}, \mathbf{t}) | \mathbf{t} = (t_1, \cdots, t_n) \in \{\{1\}, \{2\}, \{1, 2\}\}^n$; for $1 \leqq i \leqq r$ $\mathbf{x}$ makes $y_{i,k}$ true for each $k \in t_i\}|$.
10. S-SET CONNECTEDNESS (directed and undirected)
    Input: G; $s \in V$; $V' \subset V$.
    Output: Number of subgraphs of G in which for each $u \in V'$ there is a (directed) path from s to u.
11. S–T CONNECTEDNESS (directed or undirected)
    Input: G; $s, t \in V$.
    Output: Number of subgraphs of G in which there is a (directed) path from s to t.
12. S–T NODE CONNECTEDNESS (directed or undirected)
    Input: G; $s, t \in V$.
    Output: Number of subsets of V whose removal leaves a (directed) path from s to t.
13. DIRECTED TREES
    Input: Directed graph G.
    Output: Number of sets of edges that form a rooted tree, with each edge directed away from the root.
14. S–T PATHS (i.e. SELF-AVOIDING WALKS) (directed or undirected)
    Input: G; $s, t \in V$.
    Output: Number of (directed) paths from s to t that visit every node at most once.

We note that several of these problems have been widely studied. Because of the close resemblance between the permanent and the determinant the apparent computational discrepancy has been observed with surprise for a long time [16]. Despite considerable efforts no general translation from the former to the latter has been found [17], [14]. In special cases, however, such transformations do exist and lead to fast algorithms (G. Borchardt (1855), see [3]), [11], [13], [15], [18]. The maximal cliques problem arises in connection with the numerous algorithms that have been proposed for enumerating them [4], [9]. S–T paths are discussed in [2], [12]. Problems 10–12 are classical examples of reliability problems concerning networks, in the special case that the probability associated with each node or edge is a half. It will be clear, however, that the completeness results follow also for other fixed values (and of course for arbitrary values). Such problems are discussed in [6], [7]. The directed trees problem is to be contrasted with the directed spanning tree problem which can be counted fast via determinants, in analogy with Kirchhoff's matrix-tree theorem [8], [22].

THEOREM 1. *Problems* 2–14 *above are all #P-complete.*

Proof. The result follows by transitivity from the following reductions.

1. 3-SAT $\leqq$!PERMANENT. Proved in [20].
2. PERMANENT $\leqq$ PERFECT MATCHINGS. Proved in [20].
3. PERFECT MATCHINGS $\leqq$!PRIME IMPLICANTS. Given G with $V =$

$\{u_1, \cdots, u_n, v_1, \cdots, v_n\}$, $E \subset \{(u_i, v_j) | 1 \leqq i, j \leqq n\}$, we construct $G' = (V', E')$ with

$$V' = \{u_i^j, v_i^j | 1 \leqq i \leqq n; 1 \leqq j \leqq k = 2n\},$$

and

$$E' = \{(u_i^j, v_p^q) | (u_i, v_p) \in E; 1 \leqq j, q \leqq k\}.$$

We represent each edge of G' by a separate Boolean variable in which truth will denote the absence of the edge. The simultaneous presence of any pair of edges can then be prohibited by the disjunction of the corresponding variables. We can therefore write a polynomial length monotone formula F in 2-form that has the effect of prohibiting the presence of any pair of edges in G' that *either* arise from distinct edges of G that share a node *or* themselves share a node in G'. Thus to each matching of size i in G we intend there to correspond $(k!)^i$ allowed matchings in G'. Note that any prime implicant of F has $|E'| - ik$ literals for some i.

Now if $F(\mathbf{x})$ is true then at least $|E'| - nk$ of the $|E'|$ variables must be true (i.e. representing at most nk edges). If exactly $|E'| - nk$ are true then their conjunction must be a prime implicant, and corresponds to some perfect matching in G'. If I_i is the number of prime implicants of F with exactly $|E'| - ik$ literals and M_i the number of maximal matchings with i edges in G, then

$$I_i = M_i(k!)^i.$$

Hence if $\sum I_i$ is known then, by Fact 6(i), since $k! = (2n)! > M_i^2$, the value of M_n, the number of perfect matchings in G can be deduced.

4. PRIME IMPLICANTS $\leqq !$ MINIMAL VERTEX COVER. Given F construct a $G = (V, E)$ with $V = \{u_1, \cdots, u_n\}$ and $E = \{(u_i, u_j) | (x_i \vee x_j)$ is a clause of $F\}$. Any vertex cover of G is an implicant of F, and any minimal vertex cover is a prime implicant.

5. MINIMAL VERTEX COVER $\leqq !$ MAXIMAL CLIQUES. A minimal vertex cover in G corresponds to a maximal clique in the complement of G, in analogy with [10].

6. PERFECT MATCHINGS $\leqq$ IMPERFECT MATCHINGS. Given $G = (V, E)$ as in 3 we construct for each k $(1 \leqq k \leqq n+1)$ a graph G_k that consists of G with additional nodes $\{u_{ij} | 1 \leqq i \leqq n; 1 \leqq j \leqq k\}$ and additional edges $\{(u_{ij}, v_i) | 1 \leqq i \leqq n; 1 \leqq j \leqq k\}$. If A_r is the number of matchings in G of size exactly $n - r$ then these will be contained in exactly $A_r \cdot (k+1)^r$ imperfect matchings obtainable by adding only new edges. Hence the number of matchings in G_k is $\sum_{r=0}^{n} A_r \cdot (k+1)^r$. If this could be evaluated for $k = 1, \cdots, n+1$, then, by Fact 5, we could compute A_0, the number of perfect matchings.

7. IMPERFECT MATCHINGS $\leqq !$ MONOTONE 2-SAT. For G as defined in 3 above represent (the abence of) each edge by a separate variable. Let F be the formula that prohibits the presence of any pair of edges incident to the same node.

8. MONOTONE 2-SAT $\leqq !$ SAT'. Given F, denote $F \wedge F \wedge \cdots \wedge F$, k times, by F^k. If $\mathbf{x}$ satisfies exactly f clauses of F twice and the rest once, then it contributes 2^f to SAT' for F and 2^{kf} for F^k. If A_f is the number of assignments that satisfy exactly f clauses of F twice and the rest once, then the value of SAT' for F^k is $\sum_{f=0}^{r} A_f (2^k)^f$. By choosing k suitably large it follows from Fact 6(i) that $\sum A_f$ can be deduced.

9. MONOTONE 2-SAT $\leqq !$ SAT''. This is similar to 8.

10. SAT' $\leqq !$ S-SET CONNECTEDNESS. Given F construct a graph $G =$

$(V, E_1 \cup E_2)$ where

$$V = \{c_1, \cdots, c_{r+1}, x_1, \cdots, x_n, \bar{x}_1, \cdots, \bar{x}_n, s\},$$

$$E_1 = \{(x_i, c_j) | x_i \text{ appears in clause } c_j \text{ in } F\} \cup \{(x_n, c_{r+1}), (\bar{x}_n, c_{r+1})\},$$

$$E_2 = \{(x_i, x_{i+1}), (\bar{x}_i, \bar{x}_{i+1}), (x_i, \bar{x}_{i+1}), (\bar{x}_i, x_{i+1}) | 1 \leqq i \leqq n\} \cup \{(s, x_1), (s, \bar{x}_1)\}.$$

Suppose each edge in E_1 is given probability p and each one in E_2 probability q. Let A_{ij} be the number of distinct subsets of $E_1 \cup E_2$ with exactly i edges from E_1 and j from E_2, such that s is connected to each of $c_1, c_2, \cdots, c_{r+1}$. Then the probability that s is connected to each of $c_1, \cdots, c_{r+1}$ is

$$\sum\sum A_{ij} p^i (1-p)^{2r+2-i} q^j (1-q)^{4n-2-j}.$$

By Fact 6(iii), if we can evaluate this for sufficiently small p and q (e.g. $p = 2^{-2m}$, $q = 2^{-3m^2}$ where $m = 4n + 2r$) then we can compute $\{A_{ij}\}$. Such probabilities as 2^{-2m} can be simulated simply by replacing the edge by a chain of $2m$ edges of probability $1/2$. The result follows since $A_{r+1,n}$ is the required solution to SAT' for F. (N.B. The fact that SAT' is defined for monotone 2-form, rather than general 3-form, is inessential to this proof.)

11. S-SET CONNECTEDNESS $\leqq$ S–T CONNECTEDNESS. Given $G = (V, E)$ and $V' = \{n_1, \cdots, n_k\}$ construct G' by adding to G a node t and edges $\{(n_i, t) | 1 \leqq i \leqq k\}$. Suppose A_i is the number of subgraphs of G in which s is connected to exactly i nodes from V'. If each edge incident to t is given probability $1-p$ and all the others probability a half then the probability that s is connected to t is $2^{-|E|} \cdot \sum A_i (1-p^i)$. It follows from Fact 5 that by evaluating this at $k+1$ points the value of A_k can be deduced. The points can be taken as $1-p = 2^{-i}$ for $i = 1, \cdots, k+1$ since chains of suitable length can simulate these probabilities.

12. SAT' $\leqq$ S–T NODE CONNECTEDNESS. This is similar to 10 and 11 combined.

13. SAT' $\leqq$ DIRECTED TREES. Given F we construct exactly the same G as in 11, with edges directed in the sense indicated by their definitions. Suppose G_{pq} is obtained from G by giving each edge in E_1 multiplicity p, and each in E_2 multiplicity q. Let A_{ij} be the number of trees of G rooted as s with i edges from E_1 and j edges from E_2. Then the number of trees in G_{pq} rooted as s is

$$\sum_{i=1}^{2r+2} \sum_{j=1}^{4n-2} A_{ij} p^i q^j.$$

Hence by Fact 5(ii), if this is evaluated for all pairs (p, q) with $1 \leqq p \leqq 2r+3$ and $1 \leqq q \leqq 4n-1$ then the value of any A_{ij} can be deduced, including $A_{r+1,n}$ which is the desired result for SAT'. Now note that G_{pq} can indeed be simulated by an ordinary graph G'. G' consists of G augmented by chains of length p starting from each c_i, and chains of length q starting at each x_i and each $\bar{x}_i$. Finally observe that if we could count trees rooted arbitrarily, then by doing this for G' and again for G' with s removed, we could count the number of trees rooted as s.

14. HAMILTONIAN CIRCUITS $\leqq$ S–T PATHS. Given G for $k = 1, \cdots, n+1$, we generate a graph G_k by replacing each edge (u_i, u_j) by the graph with nodes $\{u_i, u_j\} \cup \{u_{ij}^1, \cdots, u_{ij}^k\}$ and edges $\{(u_i, u_{ij}^q), (u_{ij}^q, u_j) | q = 1, \cdots, k\}$. Then each s–t paths of length p in G corresponds to k^p s–t paths in G_k. Hence if there are A_p s–t paths of length p in G then the number of s–t paths in $G_k = \sum_{p=0}^{n} A_p k^p$. From Fact 5 it follows that A_n can be deduced, which is the number of Hamiltonian paths from s to t. The correspondence between Hamiltonian paths and cycles is immediate. (N.B. There is also a

natural $\leqq!$ reduction consisting of replacing each edge in G by a graph with exponentially many paths through it. Note also that the case of $s = t$ corresponds to enumerating elementary cycles [22], [23].) □

The reductions used above can be classified according to whether (a) there is just one oracle call, or (b) there are several, but all the questions for it are generated without calling the oracle. Note that reductions of the latter type can *always* be replaced by one for the former if the problem to which reduction is being exhibited is appropriate: e.g. SAT, MONOTONE 2-SAT, SAT', S–T CONNECTEDNESS, S–T PATHS. These problems can all exploit Fact 6 by being able to simulate the necessary arithmetic. We illustrate this for MONOTONE 2-SAT: Given F, to multiply its solutions by a large constant 3^k, we simply introduce $2k$ new variables and k new clauses containing two each. To multiply F_1 and F_2 we ensure that they have disjoint alphabets and simply write $F_1 \wedge F_2$. For addition of F_1 and F_2 consider F the conjunction of F_1 and F_2 with

$$(x_i \vee z) \cdots (x_n \vee z)(z \vee t)(y_1 \vee t) \cdots (y_m \vee t)$$

where the x's and y's are the variables of F_1 and F_2 respectively. Then $s(F)$ the number of solutions will equal $s(F_1)+s(F_2)+s(F_1)s(F_2)$. Hence to add we first multiply by a constant larger than the addends and perform this construction.

A further problem that can be shown to be $\#P$-hard is that of counting the number of Hamiltonian subgraphs of an arbitrary directed graph. This problem, however, appears not to belong to $\#P$. Corresponding problems for other NP-complete structures can also be formulated. Some of them appear surprisingly difficult to analyze.

For certain counting problems in $\#P$ for which no polynomial time algorithm is known, it is possible to prove that they can be computed in polynomial time given an oracle for some predicate in the Meyer–Stockmeyer hierarchy. An example is the problem of counting graph isomorphisms. Such a result can be interpreted as circumstantial evidence that the problem is not $\#P$-complete [21].

5. A problem complete in $\#P_1$. Given a complete graph on n nodes and arbitrary probabilities assigned to each edge, the probability that the graph has a Hamiltonian circuit (or some other NP-complete substructure) is easily seen to be $\#P$-complete. If, however, we insist on all the probabilities being equal to a half then the corresponding problems (i.e. of counting the number of Hamiltonian graphs, etc., of a given size) are all open. Many of the classical graph enumeration problems are of this form [8]. Here we shall give an illustrative example to show that some such problem is provably $\#P_1$-complete. We note that the arithmetic reduction needed here takes the form of the "inclusion-exclusion" principle.

We assume a fixed collection of *colours* each associated with a number. By a *graph* we shall here mean a "connected directed graph in which each edge and node is assigned a colour, with the restriction that the number of edges meeting at a node has to equal the number associated with the colour of the node". A *pattern* is such a graph without the latter degree restriction. A pattern G, can be *embedded* in graph G_2 iff there is an injective mapping of the nodes of G_1 into those of G_2 such that all nodes and edges map to corresponding colours, and edges preserve direction.

Let $A = \{A_1, \cdots, A_k\}$ be an arbitrary collection of patterns, and consider the following problem schemes:

A-PATTERNS

Input: Integer n in unary.

Output: Number of labelled graphs with n nodes into which A_i can be embedded for *all* i $(1 \leqq i \leqq k)$.

A-SUBSET-PATTERNS

Input: Integer n in unary; $X \subseteq \{1, \cdots, k\}$.

Output: Solution for A'-PATTERNS where A' is the subset of A indexed by X.

THEOREM 2. *There is a fixed collection B of fixed patterns such that* B-*SUBSET-PATTERNS is $\#P_1$-complete.*

COROLLARY. *There is a fixed collection A of fixed patterns such that A-PATTERNS $\in FP \Leftrightarrow \#P_1 \subseteq FP$.*

Proof. Set B must have a subset A (although we may not be able to identify it). □

Proof of Theorem 2. The problem is obviously in $\#P_1$. To show that it is complete we show how an arbitrary counting multi-tape TM, M, over a single letter input alphabet can be simulated by it.

We first modify M so that all accepting computations on the input of size n run for exactly $S(n)$ steps where S is an easily computed polynomial. We do this by simulating on an extra tape a binary counter that counts up to some simple polynomial that exceeds the complexity of M. The runtime of such a counter (whatever the implementation details) will clearly have some fixed value $S(n)$ that is easily computed from n. In all computation branches the modified machine M' simply runs the counter and simulates M in parallel until the former terminates. M' accepts if and only if it has simulated an accepting state of M at some time in the computation. It will be convenient to assume from now on that M and M' work on semi-infinite tapes.

We next modify M' to M'' so that M'' has just one tape but retains the property that all accepting computations have the same easily predetermined runtime. M'' treats its tape as a multiple-track tape. It initially checks that the input is of the form $1^n b^{S(n)-n-1}\$$ (where b and \$ are special new symbols) and rejects otherwise. These $S(n)$ squares are designated as work-space. In each step of the simulation of M' the workspace is scanned in both directions so as to take a fixed amount of time.

Let M'' have time complexity $T(n)$ and space complexity $S(n)$. We now claim that there is a set $C = \{C_1, \cdots, C_i\}$ of compulsory graphs, and a set $F = \{F_1, \cdots, F_j\}$ of forbidden graphs such that the number of accepting computations of M'' on input n is just the number of $(T(n)+1)(S(n)+1)$-node graphs in which all the compulsory graphs can be embedded but none of the forbidden ones. If we denote the number of graphs of this size that contain all of $C' \subseteq C \cup F$ and none of $F' \subseteq F$ as embedded subgraphs by $X(C', F')$, then clearly

$$X(\{C_1, \cdots, C_i\}, \{F_1, \cdots, F_j\}) = X(\{C_1, \cdots, C_i\}, \{F_2, \cdots, F_j\}) - X(\{C_1, \cdots, C_i, F_1\}, \{F_2, \cdots, F_j\}).$$

It follows by induction that if we can compute $X(\{A\}, \varnothing)$ for all $A \subseteq C \cup F$ then we can compute $X(C, F)$. The theorem therefore follows.

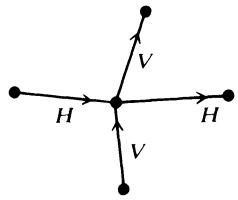
To prove the claim we represent computations by connected rectangular grids as shown in Fig. 1. Each horizontal line encodes a tape symbol and information about whether the head is there and, if so, the state of the machine. We ensure that only rectangular grids are counted by means of the following:

(i) Horizontal and vertical lines have disjoint colour sets H and V.

(ii) Nine distinct colours are used to distinguish from each other nodes that are on the four corners, on the four sides and internal, respectively. They are each assigned the number 2, 3 or 4 as appropriate.

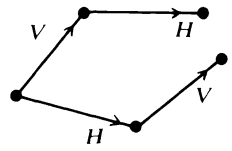
(iii) We forbid internal nodes from having incident edges in any way other than

the following:

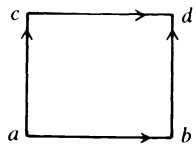


Similar prohibitions are made for the other eight categories of nodes.

(iv) We ensure a grid structure by insisting that all appropriate chains of four edges close. Thus the following scheme is forbidden:



(v) In any subgraph:



if a, b, c are internal we forbid d to be anything other than internal. Similar provisions are made for the other cases.

(vi) A bottom left corner node is compulsory.

We further ensure that the grid represents a correct computation by:

(vii) insisting that the bottom left corner represents a head position and start state, and forbidding anything else on the bottom boundary from representing a head position;

(viii) forbidding illegal transitions;

(ix) insisting on an accepting state in the top right corner.

Finally it remains to observe that the theorem holds even for a *fixed* collection B because it is sufficient to simulate just the following fixed TM, M, which is clearly complete for $\#P_1$: on unary input n, M first verifies that $n = 2pq$ (or $4pq$) where $p \leqq q$ and p and q are the ith and jth prime numbers respectively, and then simulates the ith machine in $\#\mathrm{TIME}(n)$ on input j (or vice versa). □

The corollary above is a natural example of a problem that is provably as hard as any complete problem for the class containing it but not necessarily complete itself. Note, however, that it proves only the existence of such a problem in the sense that we cannot show A-PATTERNS to be in this category for any explicitly given A. It also remains open to determine whether Theorem 2 or the corollary holds when A or B is a singleton set.

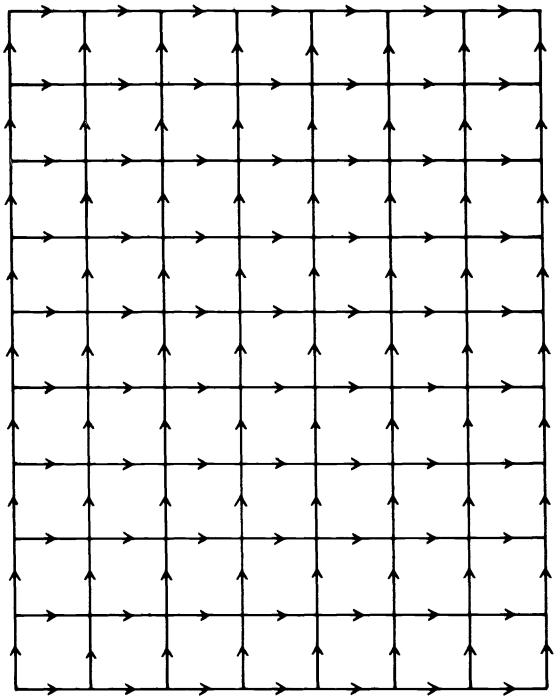


FIG. 1

6. Conclusion. We have shown that the notions of $\#P$-completeness and of algebraic or arithmetic polynomial time reducibilities are useful tools for classifying the relative complexities of counting problems. The completeness class for $\#P$ appears to be rivalled only by that for NP in relevance to naturally occurring computational problems. Because of the richness of potential reductions it is reasonable to suppose that many further ideas will be required before the classification of new counting problems becomes a routine task.

Some possible next steps are the obvious omissions from this paper (e.g. maximal matchings, undirected trees, connectivity of all points in a graph). A more general question is that of tackling the large number of classical enumeration problems for which there is just one input for each size. For example, we have as yet no evidence that it is difficult to determine the number of Hamiltonian graphs of each size.

Acknowledgment. I am grateful to Dana Angluin for several helpful discussions. The proof given for imperfect matchings is a simplification of one found by her earlier.

REFERENCES

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

[2] M. N. BARBER AND B. W. NINHAM, *Random and Restricted Walks*, Gordon and Breach, New York, 1970.

[3] M. MARCUS AND H. MINC, *Permanents*, Amer. Math. Monthly, 72 (1965), pp. 577–591.

[4] C. BRON AND J. KERBOSCH, *Algorithm* 457: *Finding all cliques in an undirected graph*, Comm. ACM, 16 (1973), pp. 575–577.

[5] S. A. COOK, *The complexity of theorem proving procedures*, Proc. 3rd ACM Symp. on Theory of Computing, 1971, pp. 151–158.

[6] D. W. DAVIES AND D. L. A. BARBER, *Communication Networks for Computers*, John Wiley, London, 1973.
[7] H. FRANK AND I. T. FRISCH, *Communication, Transmission and Transportation Networks*, Addison-Wesley, Reading, MA, 1971.
[8] F. HARARY AND E. M. PALMER, *Graphical Enumeration*, Academic Press, New York, 1973.
[9] H. C. JOHNSTON, *Cliques of a graph—Variations on the Bron–Kerbosch algorithm*, Internat. J. Comput. Information Sci., 5 (1976), pp. 209–238.
[10] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972.
[11] P. W. KASTELEYN, *Dimer statistics and phase transitions*, J. Mathematical Phys., 4 (1963), pp. 287–293.
[12] ———, *Graph theory and crystal physics*, Graph Theory and Theoretical Physics, F. Harary, ed., Academic Press, New York, 1967.
[13] C. H. C. LITTLE, *A characterization of convertible* (0, 1)*-matrices*, J. Combinatorial Theory Ser. B, 18 (1975), pp. 187–208.
[14] M. MARCUS AND H. MINC, *On the relation between the determinant and the permanent*, Illinois J. Math., 5 (1961), pp. 376–381.
[15] E. W. MONTROLL, *Lattice Statistics*, Applied Combinatorial Mathematics, E. F. Beckenbach, ed., John Wiley, New York, 1964.
[16] T. MUIR, *On a class of permanent symmetric functions*, Proc. Roy. Soc., Edinburgh, 11 (1882), pp. 409–418.
[17] G. PÓLYA, *Aufgabe* 424, Arch. Math. Phys., 20 (1913), p. 271.
[18] H. N. V. TEMPERLEY AND M. E. FISHER, *Dimer problems in statistical mechanics—An exact result*, Philos. Mag., 6 (1961), pp. 1061–1063.
[19] L. G. VALIANT, *A polynomial reduction from satisfiability to Hamiltonian circuits that preserves the number of solutions*, Manuscript, University of Leeds, 1974.
[20] ———, *The complexity of computing the permanent*, CSR-14-77 Univ. of Edinburgh, 1977; Theor. Comput. Sci., to appear.
[21] D. ANGLUIN, *On counting problems and the polynomial time hierarchy*, Theoret. Comput. Sci., to appear.
[22] C. BERGE, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.
[23] R. E. TARJAN, *Enumeration of the elementary circuits of a directed graph*, this Journal, 2 (1973), pp. 211–216.

SIAM J. COMPUT.
Volume 8, No. 3, August 1979

0097-5397/79/0803-0013$01.00/0

MULTI-TERMINAL 0–1 FLOW

YOSSI SHILOACH†

Abstract. Given an undirected 0–1 flow network with n vertices and m edges, we present an $O(n^2(m+n))$ algorithm which generates all $\binom{n}{2}$ maximal flows between all the pairs of vertices. Since $O(n^2(m+n))$ is also the size of the output, this algorithm is optimal up to a constant factor.

Keywords. Algorithm, multiterminal flow, 0–1 integer flow

1. Introduction. A 0–1 undirected flow network is essentially an undirected graph $G=(V, E)$ since all the edges have one unit capacity, and the flow assumes only integer values, namely 0 or 1. G is assumed to have n vertices and m edges. The edges will be denoted as two element sets such as $\{u, v\}$.

Given $s, t \in V$, and $s \to t$ 0–1 *integer flow* ($s \to t$ *flow* in short) is a function $f: V \times V \to \{0, 1\}$ such that:

(a) $f(u, v)=0$ if $\{u, v\} \notin E$.

(b) $f(u, v)=0$ or 1 if $\{u, v\} \in E$.

(c) If $f(u, v)=1$, then $f(v, u)=0$.

(d) $IN(f, v)=OUT(f, v)$ for all $v \in V-\{s, t\}$, where $IN(f, v)=\sum_{u \in V} f(u, v)$ is the total amount of flow entering v and $OUT(f, v)=\sum_{w \in V} f(v, w)$ is the total amount of flow emanating from v.

The *value* of f denoted by $|f|$ is $OUT(f, s)-IN(f, s)$. An $s \to t$ flow f is *maximal* if $|f| \geqq |f'|$ for any other $s \to t$ flow f'.

The 0–1 integer flow problems are usually associated with finding a maximal number of edge-disjoint or vertex-disjoint paths between two vertices in a graph. They often represent problems in transportation, electricity, layout and any other kind of problems in real-life flow-networks. Such an individual maximal flow problem can be solved in $O(n^{2/3}(m+n))$ time, as shown in [1].

In this paper we present an algorithm which generates the maximal flows between all the pairs of vertices within $O(n^2(m+n))$ time which seems to be optimal regarding the output size. Finding all $\binom{n}{2}$ maximal flow values can be done in $O(n^{5/3}(m+n))$ time, if we use Gomory and Hu's algorithm, (see [2]). (Computing all the maximal flow values is an easy $O(n^2)$ extension of the original algorithm presented in [2].)

2. The multiterminal flow algorithm (MULTEF). MULTEF consists of two routines. The first routine computes a *cut-tree* for G. A cut-tree $T=(V_T, E_T)$ is a weighted tree (i.e., a nonnegative weight $w(e)$ is associated with each $e \in E_T$) with the following properties:

(a) $V_T = V$.

(b) For all $s, t \in V$, the value of a maximal $s \to t$ flow equals $\min_{e \in P_T(s,t)} w(e)$.

$P_T(s, t)$ is the unique path connecting s and t in T. (In the following we will use $d_T(s, t)$ to denote the length of $P_T(s, t)$, and $F_s(t)$ to denote the neighbor of t on $P_T(s, t)$.) The existence of such a cut-tree is proved in [2]. They also provide an algorithm which computes the tree by solving only $n-1$ individual max-flow problems.

* Received by the editors March 9, 1978, and in revised form August, 1978.

† Computer Science Department, Stanford University, Stanford, California 94305. This research was supported in part by a Chaim Weizmann Postdoctoral Fellowship and in part by the National Science Foundation under Grant MCS 75-22870.

The second routine is $MIN(u, v, w)$. Given a $u \to v$ flow f_{uv} and a $v \to w$ flow f_{vw}, $MIN(u, v, w)$ computes a $u \to w$ flow f_{uw} such that

$$|f_{uw}| = \min(|f_{uv}|, |f_{vw}|).$$

The existence of a $u \to w$ flow having this value can be easily proved by using the max-flow = min-cut theorem. $MIN(u, v, w)$ will be described in full in the next section.
MULTEF:

1. *Initialization.* Compute the cut tree $T = (V_T, E_T)$ of G and $n-1$ maximal $s \to t$ flows for all s, t such that $\{s, t\} \in E_T$.
2. For each vertex $s \in V$ **do**
   Begin For $d = 2, \cdots, n-1$ **do**
   Begin For all $t \in V$ such that $d_T(s, t) = d$, compute a maximal $s \to t$ flow by using $MIN\ (s, F_s(t), t)$.
   End
   End.

It can be easily verified that no more than $2(n-1)$ maximal flows have to be stored at the same time, and thus MULTEF has an $O(n(m+n))$ space bound.

The validity of MULTEF can be easily derived from the properties of the cut-tree (using induction on d). The complexity of MULTEF is $O(n^{5/3}(m+n)) + O(n^2 \cdot$ complexity of $MIN)$. In § 3 we will describe a linear time algorithm for MIN which yields an $O(n^2(m+n))$ time bound for MULTEF.

3. *MIN*(u, v, w). Let $u, v, w \in V$. Given a $u \to v$ flow f_{uv} and a $v \to w$ flow f_{vw}, $MIN(u, v, w)$ provides a $u \to w$ flow f_{uw} such that $|f_{uw}| = \min(|f_{uv}|, |f_{uw}|)$. Henceforth we assume that:

$$|f_{uv}| = |f_{vw}|. \tag{3.1}$$

(3.2) Both f_{uv} and f_{vw} are acyclic flows.

If $|f_{uv}| > |f_{vw}|$, we reduce f_{uv} by $|f_{uv}| - |f_{uw}|$ units of flow so that (3.1) holds. The second assumption is justified by a linear time algorithm which eliminates cycles of flow and described in detail in § 5.

The most straightforward way to produce a $u \to w$ flow out of f_{uv} and f_{vw} is to add them up. So let $\phi_{uw} = f_{uw} \oplus f_{vw}$ be defined by:

$$\phi_{uw}(v_1, v_2) = \max(0, f_{uv}(v_1, v_2) + f_{vw}(v_1, v_2) - f_{uv}(v_2, v_1) - f_{vw}(v_2, v_1)). \tag{3.3}$$

It is easy to see that ϕ_{uw} is nonnegative and if $\phi_{uw}(v_1, v_2) > 0$ then $\phi_{uw}(v_2, v_1) = 0$. Moreover, ϕ_{uw} satisfies the conservation rule, i.e., $IN(\phi_{uw}, z) = OUT(\phi_{uw}, z)$ for all $z \in V - \{u, w\}$. (Equation (3.1) implies the conservation of flow at v too.) However, edges may become overflowed as shown in Fig. 1 where $\phi_{uw}(x, y) = 2$.

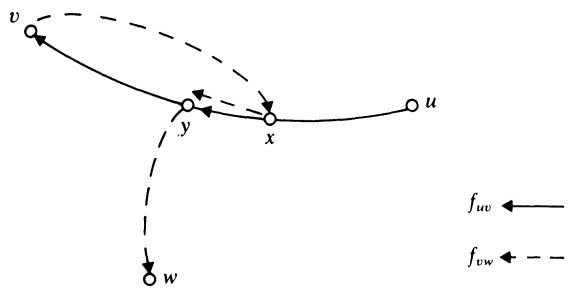


FIG. 1

The basic idea to resolve this problem (speaking in terms of Fig. 1) is to reduce f_{uv} from x to v and reduce f_{vw} from x back to v by one unit. The pseudo-flow of Fig. 1 turns out to be the flow of Fig. 2.

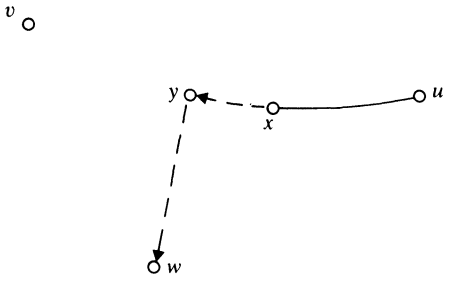


FIG. 2

The process of reducing f_{uv} from x to v propagates in the same direction as f_{uv} itself and will be denoted as "*reducing the flow forward*" or "*redford*" in short. Reducing f_{vw} from x to v has the opposite direction to that of f_{vw} and is called "*reducing backward*" or "*redback*". Thus, in principle, we redford f_{uv} and redback f_{vw} towards v.

Trying to implement the redford-redback idea, we might face a problem which is demonstrated in Fig. 3. After reducing f_{uv} forward and f_{vw} backward from x_1 to v, we obtain the pseudo-flow of Fig. 3(b). Now, we can no longer redford from x_2. We are going to resolve this difficulty *partially* by using the acyclic orientation of f_{uv}.

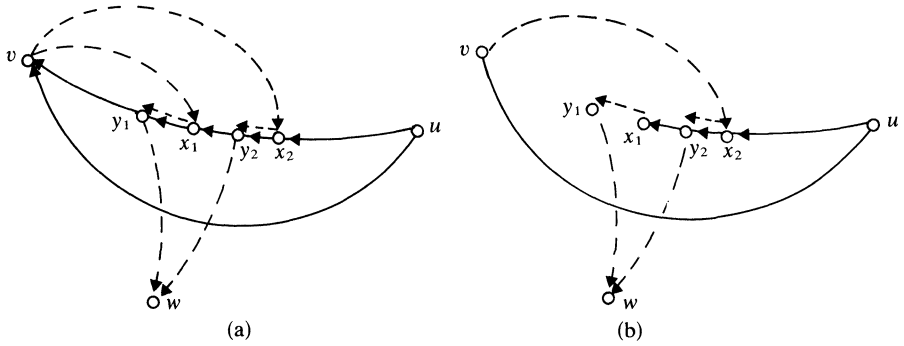


FIG. 3

DEFINITION. Given a flow f in an undirected flow network $G=(V, E)$, $G(f)$ is defined by:

$$G(f)=(V, E(f)) \quad \text{where } E(f)=\{(v_1, v_2): f(v_1, v_2)>0\}.$$

Note that $G(f)$ is a directed graph. Let $\bar{E}=\{\{x_i, y_i\}: i=1, \cdots, k\}$ denote the set of all the overflowed edges (i.e., $\phi_{uw}(x_i, y_i)=2$ for $i=1, \cdots, k$ and $\phi_{uw}(e)<2$ if $e \notin \bar{E}$). Let $X=\{x_1, \cdots, x_k\}$. For any x_i and x_j in X, we say that x_i *precedes* x_j (x_j *follows* x_i) if there exists a directed path in $G(f_{uv})$ from x_i to x_j. Due to the acyclity of $G(f_{uv})$, this relation is irreflexive. If we start reducing forward from x_i's which do not follow other vertices in X, we eliminate the problem which is sketched in Fig. 3. Figure 4 shows what happens if we start to redford-redback from x_2 which precedes x_1 in $G(f_{uv})$. Note that after reducing forward and backward from x_2, no redford-redback is needed at x_1 since $\{x_1, y_1\}$ is not overflowed anymore.

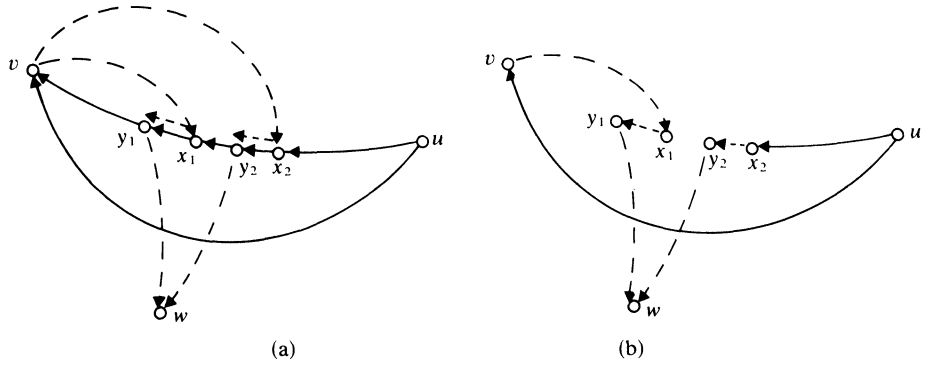


FIG. 4

This is only a partial solution. Since we must redford and redback from the same vertex (otherwise conservation is violated), we might face the same problem in reducing backward. The problem occurs when a redback path enters a vertex x_i from which a previous redback took place and now no f_{vw} flow enters x_i (see Fig. 5).

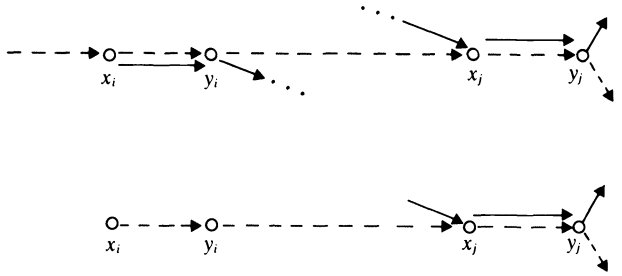


FIG. 5. *If another redford-redback process starts at x_j, the redback path will be stuck at x_i.*

The solution to this problem can be outlined as follows:

Step 1. We redford along the $u \to v$ flow until no overflowed edges are left. We now have to rebalance the vertices in which the redford paths start.

Step 2. We redback starting from the unbalanced vertices. If we get stuck we go to Step 3.

Step 3. We modify the appropriate redford path by increasing the flow along a certain prefix of it.

The algorithm is designed so that Step 3 does not yield to further modifications of the redback paths.

3.1. Detailed implementation.

Step 1. The redford paths, say $P_1, \cdots, P_t$, are a set of edge-disjoint paths in $G(f_{uv})$ which cover all the overflowed edges. Each of the P_i's begins at an overflowed edge and terminates at v. We may assume that P_i begins at (x_i, y_i), $i = 1, \cdots, t$. The P_i's can be produced by $ACYC(f_{uv})$ which is described in full in § 5. As soon as the P_i's are produced, we redford the flow by one unit along each of them.

In the same way we produce a set $\{Q_1, \cdots, Q_s\}$ of edge-disjoint redback paths in $G(f_{vw})$. Each of them starts in an overflowed edge and proceeds "backward" towards v and their union contains $\bar{E}$. The only difference is that we use the Q_j's to redback only if necessary as specified in the implementation of Step 2. The edges at which the current redford paths start are stored in a stack which contains initially $(x_1, y_1), \cdots, (x_t, y_t)$.

Step 2. Let (x_i, y_i) be the first edge in the stack. We start to redback at x_i, following two rules.

R_1: If $(x_i, y_i) \in Q_j$ then the redback starts at the edge which follows[1] (x_i, y_i) on Q_j (see Fig. 6).

R_2: If we start to redback at an edge which belongs to Q_j, we containue to redback along Q_j until we reach v or get stuck. If we get stuck we go to Step 3. If we reach v, we deplete (x_i, y_i) from the stack. If the stack is not empty, we go back to Step 2, otherwise the algorithm terminates.

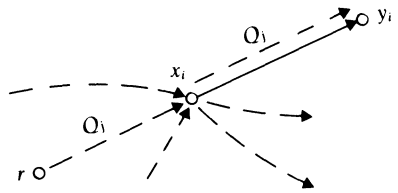


FIG. 6. *By* R_1, *redback starts at* (r, x_i).

DEFINITION. A *redback trail*, (*trail* in short), is a subpath of a redback path, along which we redback the $v \to w$ flow as described in Step 2.

LEMMA 3.1. *A redback trail can get stuck at a vertex z only if* $\exists i$ *such that*: $z = x_i$, (x_i, y_i) *is an overflowed edge, the first on its redford path and* (x_i, y_i) *belong to this redback trail or a previous one.*

The proof of Lemma 3.1 is given after Step 3.

Remark. Due to the following Step 3, redford paths may shorten. Saying that (x_i, y_i) is the first overflowed edge on its redford path, means that this is the situation when the redback trail gets stuck.

Step 3. Assume that our redback trail get stuck at x_i and (x_i, y_i) is the overflowed edge discussed in Lemma 3.1. Since we cannot redback anymore, we shall balance x_i by increasing the $u \to v$ flow forward ("inford") along the old redford path which starts at (x_i, y_i). The idea is that since the flow on (x_i, y_i) has been reduced before (Lemma 3.1), we no longer have to include this edge in our redford program. Thus, we inford along the redford path containing (x_i, y_i) until we reach v or encounter another overflowed edge (x_j, y_j) in which no redback has taken place so far. In both cases, (x_i, y_i) is deleted from the stack and in the latter, (x_j, y_j) is inserted. If the stack is not empty we go back to Step 2, otherwise, the algorithm terminates.

Remark. Since (x_i, y_i) is the first edge on its redford path (Lemma 3.1), the net effect of the inford is to shorten the redford path to the minimum necessary.

Proof of Lemma 3.1. Let us consider two cases.

Case 1. The trail starts at z (i.e., the trail consists of a single vertex). Step 2 implies the $\exists i$ such that $z = x_i$ and an overflowed edge (x_i, y_i) which is the first in its redford path. (In fact, (x_i, y_i) is the top edge in the stack.) Thus (x_i, y_i) belongs to a redback path, say Q_j, and the first edge in our redback trail should have been that which follows (x_i, y_i) on Q_j, (R_1). Since this edge has already been used in a previous redback trail (otherwise we are not stuck), (x_i, y_i) has also been used in the same trail, (R_2).

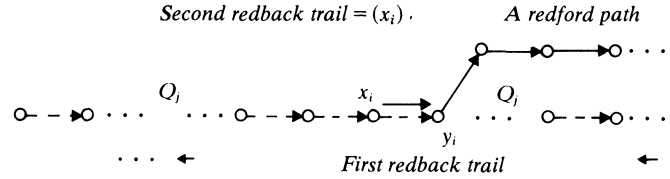
Case 2. The trail did not start at z. The only reason that R_2 cannot be used to continue the trail is that a previous redback trail started at z before. This again means that when the previous trail got stuck, we had an overflowed edge (x_i, y_i) such that $z = x_i$ and at that time (x_i, y_i) was the first edge on its redford path. It also implies that our

[1] Recall that Q_j proceeds backward.

redback trail enters x_i through y_i, (R_1). Applications of Step 3 which took place before our redback trail got stuck, could not change the fact that (x_i, y_i) was the first edge on its redford path, since we start to inford from (x_i, y_i) only after our redback trail got stuck there. Moreover, since (x_i, y_i) is the first on its redford path, it could not belong to any inford trail which did not start at it. □

Both cases are illustrated in Fig. 7.

Case 1.



Case 2.

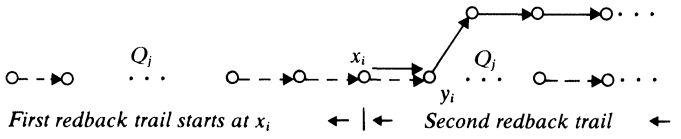


FIG. 7

3.2. A detailed example. In Fig. 8 we illustrate the composition of a $u \to v$ flow f_{uv} and a $v \to w$ flow f_{vw}. Both are acyclic and have the same value, 2. The overflowed edges

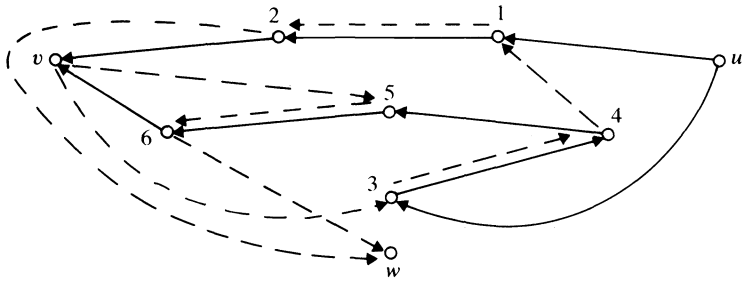


FIG. 8

are (1, 2), (3, 4) and (5, 6). Let the redford paths be:

$$P_1 = (1, 2, v), \qquad P_2 = (3, 4, 5, 6, v).$$

The redback paths are:

$$Q_1 = (2, 1, 4, 3, v) \quad \text{and} \quad Q_2 = (6, 5, v).$$

The stack contains ((1, 2), (3, 4)).

Figure 9 illustrates the situation after redford has been completed. Redback trails should start at 1 and 3 (one at each).

First redback trail = (1, 4, 3, v); stack = ((3, 4))

Second redback trail = (3), got stuck at 3.

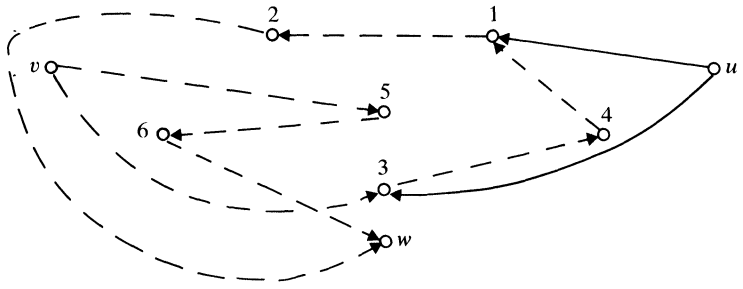


FIG. 9

Figure 10 shows the situation at this moment.

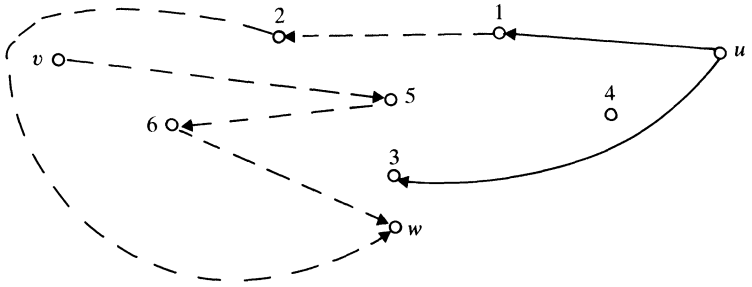


FIG. 10

Now inford takes place, starting at 3.

Inford trail = (3, 4, 5), got stuck at 5; stack = ((5, 6)).

The current situation is shown in Fig. 11.

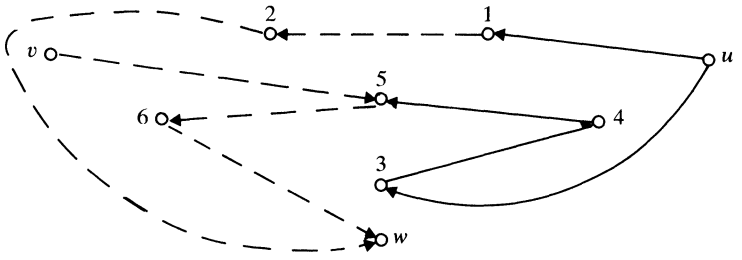


FIG. 11

Finally we redback from 5.
The redback trail = (5, v); stack = ϕ and the final $u \to w$ flow is shown in Fig. 12.

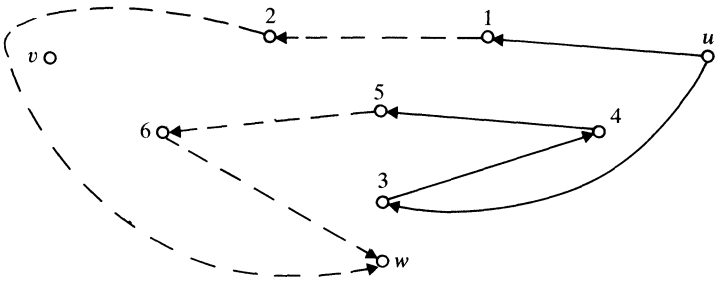


FIG. 12

4. Validity and complexity of *MIN*(u, v, w).

Validity. We have to show that:

1) *MIN*(u, v, w) terminates.

2) The output, f_{uw}, of *MIN*(u, v, w) is a legal $u \to w$ flow and $|f_{uw}| = |f_{uv}|$ $(= |f_{vw}|)$.

1) The termination is quite clear. Step 1 obviously terminates. Step 2 terminates since reducing back is done only on the $v \to w$ flow and this flow is never increased. Step 3 is finite since the $u \to v$ flow along an edge can be increased at most once.

2) Let's first prove the following equation.

$$(4.1) \qquad OUT(f_{uw}, z) - IN(f_{uw}, z) = OUT(\phi_{uw}, z) - IN(\phi_{uw}, z) \quad \text{for all } z \neq v.$$

Proof of Equation (4.1). Equation (4.1) is violated during the execution, in two cases. The first occurs when a redford path starts at an edge incident with z, say (z, y). In this case (z, y) is inserted to the stack and when it reaches its top, z will be rebalanced by the redback routine. The second case occurs when a redback trail gets stuck at z. In this case the inford routine rebalances z. □

All the vertices $\neq u$, v, w are balanced in f_{uw}. This assertion follows immediately from (4.1) and the fact that they are balanced in ϕ_{uw}.

The vertex v is also balanced. By (3.1) we have

$$OUT(\phi_{uw}, u) - IN(\phi_{uw}, u) = IN(\phi_{uw}, w) - OUT(\phi_{uw}, w)$$

and (4.1) then implies that

$$OUT(f_{uw}, u) - IN(f_{uw}, u) = IN(f_{uw}, w) - OUT(f_{uw}, w).$$

This, combined with the fact that all the other vertices are balanced, implies that v is also balanced.

$|f_{uw}| = |f_{uv}|$. By the definition of ϕ_{uw}, $|\phi_{uw}| = |f_{uv}|$ and (4.1) (with $z = u$) implies that $|\phi_{uw}| = |f_{uw}|$ and thus $|f_{uw}| = |f_{uv}|$.

No edge is overflowed. We have taken care of all the overflowed edges at Step 1. We increase the flow again only in Step 3. However, as explained there, this increase does not overflow any edge again.

This completes the validity proof of *MIN*(u, v, w).

Complexity. Producing ϕ_{uw} is obviously linear. An edge of $G(f_{uv})$ is treated at most twice (Steps 1 and 3) and an edge of $G(f_{vw})$ is treated at most once (Step 2). Thus, *MIN*(u, v, w) is linear.

5. *ACYC*(f). Given a $u \to v$ 0–1 flow f, *ACYC*(f) produces an acyclic $u \to v$ 0–1 flow f' of the same value.

Starting from u, we grow a directed path in $G(f)$, proceeding in a depth-first manner. Each vertex z is labeled when encountered and the label is removed when we backtrack through it. The label consists of the name of the (unique) vertex through which we entered z. The edges are labeled by the letter F when encountered. This label is permanent. We proceed only along unlabeled edges until one of the two cases occur:

(a) We reach a labeled vertex z. This means that we have discovered a flow cycle C through z. We backtrack along C and remove its edges. The labels of the vertices of $C - \{z\}$ are also removed and we continue to grow the path from z.

(b) We reach v. This means that we have just found a $u \to v$ flow path. We backtrack along the path and remove the labels from the vertices (not from the edges). If there are unlabeled edges incident with u, we start growing a new $u \to v$ path. Otherwise—we stop.

It is easy to see that the edges which are labeled by F form an acyclic 0–1 flow, f', from u to v and $|f'| = |f|$. It is also easy to verify that $ACYC(f)$ has an $O(m+n)$ time bound and it provides $|f|$ edge-disjoint $u \to v$ paths in $G(f)$. Thus, $ACYC(f)$ can be employed in Step 1 of *MIN*, to produce the redford paths.

6. Related open problems. The most natural open problem is that of generalizing MULTEF to undirected networks with arbitrary capacities. It is easy to see that if *MIN* can be generalized, so can MULTEF. Thus, the main question is whether *MIN* can be generalized to general undirected networks with a reasonable time bound. This question also involves the problem of generalizing $ACYC$ to arbitrary capacities, which is an interesting open question by itself.

REFERENCES

[1] S. EVEN AND R. E. TARJAN, *Network flow and testing graph connectivity*, this Journal, 4 (1975), pp. 507–518.

[2] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551–570.

SIAM J. COMPUT.
Volume 8, No. 3, August 1979

0097-5397/79/0803-0014$01.00/0

A NOTE ON SPARSE COMPLETE SETS*

STEVEN FORTUNE†

Abstract. Hartmanis and Berman have conjectured that all *NP*-complete sets are polynomial time isomorphic. A consequence of the conjecture is that there are no sparse *NP*-complete sets. We show that the existence of an *NP*-complete set whose complement is sparse implies $P = NP$. We also show that if there is a polynomial time reduction with sparse range to a *PTAPE*-complete set, then $P = PTAPE$.

Key words. reduction, polynomial time, nondeterministic polynomial time, complete sets, sparsity

1. Introduction. Hartmanis and L. Berman in [4] conjecture that all the *NP*-complete sets are isomorphic via polynomial time mappings. Of course, proving the conjecture would prove $P \neq NP$ and hence is likely to be hard to do. One consequence of the conjecture that they point out is that there could be no sparse *NP*-complete set, that is, there could be no *NP*-complete set having fewer than $p(n)$ elements of length n, where p is a polynomial. A proof of this consequence could be viewed as evidence for the conjecture, but currently seems to be unobtainable, even under the assumption $P \neq NP$.

In [2], P. Berman does obtain the following result. He shows that if there is a polynomial time reduction with sparse range mapping one *NP*-complete set to another, then $P = NP$. As a corollary, he shows that if there is an *NP*-complete set over 1^*, then $P = NP$. In this note we extend this result to show that if there is a sparse set complete for co*NP*, then $P = NP$. Thus, for example, if the set of tautologies can be reduced to a sparse set, then $P = NP$. We also show that if there is a polynomial time reduction with sparse range to a *PTAPE*-complete set then $P = PTAPE$.

The general idea of the proof is the following. We will give an algorithm to decide if a Boolean formula F written in conjunctive normal form is satisfiable. The running time will be polynomial under the assumption that there is a *NP*-complete set with sparse complement. The algorithm constructs a binary tree where the nodes are labeled with formulas obtained by assigning values to some of the variables in F. In general, a node labeled by formula G will have two sons, one labeled with the formula obtained by setting one of the variables in G to 1, the other labeled with the formula obtained by setting the variable to 0. Of course, if such a tree were completely constructed, it would have exponential size. However, by using information gathered from a mapping to the cosparse complete set, the tree can be pruned to only a polynomial in size.

2. Sparsity of complete sets. In the proof of Theorem 1 we use the fact that *SAT*, the set of satisfiable formulas written in conjective normal form, is *NP*-complete. This was originally shown in [3]; the textbook [1] also contains a proof along with additional information on complete problems.

THEOREM 1. *Suppose there is an NP-complete set L which is cosparse, that is, there is at most a polynomial in n of words of length n not in L. Then P = NP.*

Proof. Since L is *NP*-complete, there is a polynomial time computable function t such that F is in *SAT* if and only if $t(F)$ is in L. The following algorithm will decide if a formula F is satisfiable.

* Received by the editors June 12, 1978, and in revised form October 9, 1978.

† Department of Computer Science, Cornell University, Ithaca, New York 14853. This research was supported in part by the Office of Naval Research under Grant N0014-76-C-0018.

Create the root node and label it with F.
while the root is not marked "unsatisfiable" **do**
Pick the lowest node n in the tree not marked "unsatisfiable".
Let the label of n be G.
Choose a variable x appearing in G and create two sons of n. Label one with the formula obtained by setting $x = 0$ in G (and doing trivial simplifications: $y+0=y$, $y+1=1$, $(y+z)\cdot 1 = y+z$, $(y+z)\cdot 0 = 0$), label the other with the formula obtained by setting $x = 1$.
If there is a node corresponding to a satisfying assignment (i.e. a node labeled with the formula 1) **then** output ("satisfiable"); **stop**.
while there is an unmarked node K with formula H satisfying either
(a) both sone of k are marked "unsatisfiable"
(b) H is trivially unsatisfiable (i.e. has a conjunct which is 0)
(c) there is some node k' with formula H' marked "unsatisfiable", and $t(H) = t(H')$
or (d) some ancestor of k is marked unsatisfiable
do mark k "unsatisfiable" **end**
end
output ("unsatisfiable")

The correctness of the algorithm follows from the assertion that a node is marked "unsatisfiable" only if in fact the formula of the node is unsatisfiable. This in turn follows by examining the four cases in which a node is marked "unsatisfiable".

To see that the algorithm runs in polynomial time, first note that there are only polynomially many different values of $t(H)$ not in L, as H varies over the formulas obtained by assigning values to some of the variables in F. We will show that after at most v iterations of the outer loop, where v is the number of variables in F, either a satisfying assignment is found or a new value of the range of t is discovered to be not in L. Hence the whole algorithm runs in polynomial time.

Consider a node n labeled with formula G chosen at the start of some iteration of the outer loop. Note that $t(G)$ is not known not to be in L as n is unmarked. Suppose G has k variables. We will show by induction that after at most k iterations of the outer loop either a new value of the range of t is discovered to be not in L or a satisfying assignment is found. If $k = 1$ then the two formulas assigned to the sons of n are variable free. Hence either at least one is the formula "1" and a satisfying assignment is found, or both are "0" and $t(G)$ is discovered to be not in L. The inductive step, $k > 1$, breaks into two cases. Either both formulas assigned to the sons of n are immediately marked "unsatisfiable" or at least one of them is not. In the former case node n will also be marked "unsatisfiable" and $t(G)$ will be discovered to be not in L. In the latter case one of the unmarked sons will be chosen at the next iteration of the outer loop, as the son must be the lowest unmarked node in the tree. The induction hypothesis now applies since the formula of the chosen son has at most $v-1$ variables. Hence after at most another $v-1$ iterations either a new element of the range of t is discovered not to be in L, or a satisfying assignment is found. □

As another application of this technique, we have the following theorem. QBF here is the set of valid quantified Boolean formulas; it was shown to be *PTAPE*-complete in [5].

THEOREM 2. *Suppose there is a polynomial time computable function t and a set L such that*

(a) *F is in QBF if and only if $t(F)$ is in L;*
(b) $|\{t(w)$: *w is of length* $n\}| < p(n)$ *for some polynomial p. Then P = PTAPE.*

COROLLARY. *If there is a PTAPE-complete set over* 1^*, *then* $P = PTAPE$.

Proof of Theorem 2. The following algorithm will decide whether a Boolean formula $F = \forall x_1 \exists x_2, \cdots, Qx_v H(x_1, \cdots, x_v)$ is valid.

Create the root mode and label it with F
while the root is unmarked **do**
  Pick the lowest unmarked node, n, in the tree.
  Let the formula labeling node n be G, and y the variable of the outermost quantifier.
  Create two sons of n. Label one with the formula obtained from G by setting $y = 0$, the other with the formula obtained by setting $y = 1$.
while there is an unmarked node n with formula G satisfying one of
  (a) The leading quantifier of G is $\exists$ and n has a son marked "valid"
  (b) The leading quantifier of G is $\forall$ and n has a son marked "invalid"
  (c) G is the formula 1
  (d) G is the formula 0
or (e) There is some marked node n' labeled with a formula G', and $t(G) = t(G')$.
  do
  In cases (a) or (c) mark n "valid".
  In cases (b) or (d) mark n "invalid".
  In case (e) mark n the same as n'.
  end
end
Output the label of the root.

The proof of correctness is similar to that of Theorem 1. The running time analysis depends on the fact that the range of t is sparse and is otherwise analogous to that of Theorem 1. □

REFERENCES

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

[2] P. BERMAN, *Relationships between density and deterministic complexity of NP-complete languages*, Fifth International Colloquium on Automata, Languages and Programming (1978), Springer-Verlag, Berlin-Heidelberg-New York, pp. 63–71.

[3] S. COOK, *The complexity of theorem-proving procedures*, Proceedings of the Third Annual ACM Symposium on Theory of Computation (1971), Association for Computing Machinery, New York, pp. 151–158.

[4] J. HARTMANIS AND L. BERMAN, *On isomorphisms and density of NP and other complete sets*, Proceedings of the Eighth Annual ACM Symposium on Theory of Computing (1976), Association for Computing Machinery, New York, pp. 30–40; also this Journal, 6 (1977), pp. 305–322.

[5] A. R. MEYER AND L. J. STOCKMEYER, *Word problems requiring exponential time*, Proceedings of the Fifth Annual ACM Symposium on Theory of Computing (1973), Association for Computing Machinery, New York, pp. 1–9.

SIAM J. COMPUT.
Volume 8, No. 3, August 1979

0097-5397/79/0803-0015$01.00/0

POLYNOMIAL SPACE AND TRANSITIVE CLOSURE*

RONALD V. BOOK†

Abstract. A characterization of PSPACE in terms of the regular sets and certain algebraic closure operations is developed. It is shown that NP = PSPACE if and only if NP is closed under a form of the transitive closure operation.

Key words. PSPACE, regular sets, homomorphic replication, intersection, transitive closure

Introduction. In [1] it is shown that a number of different classes of languages arising in the study of complexity theory and computability theory can be characterized in terms of the class of regular sets, either intersection or both intersection and complementation, and certain restrictions on the operation of homomorphic replication. These characterizations are simple and uniform and reveal that the various classes have extremely similar structures. Here we consider classes of languages specified by space-bounded machines.

The class NP of languages accepted in polynomial time by nondeterministic Turing machines can be characterized by beginning with the class of regular sets and requiring closure under the operations of intersection and polynomial-erasing homomorphic replication [1]. Here it is shown (Theorem 2) that the class PSPACE of languages accepted by deterministic Turing machines using polynomial space is characterized by beginning with the class of regular sets, requiring closure under intersection and polynomial-erasing homomorphic replication, and, additionally, requiring that the transitive closure of certain relations should remain in the class when the relations are encoded as languages. As a result of Theorem 2, we see that NP and PSPACE differ by at most the requirement of "weak transitive closure."

Our first result (Theorem 1) is a characterization of the class of context-sensitive languages. This characterization is related to the characterization of the class of predicates recognized by nondeterministic linear-bounded automata in terms of transitive closure given by Jones [4], [5]. In his characterization Jones uses the class of strictly rudimentary predicates as a "sub-basis," while here we use the class $\mathscr{L}_{\mathrm{BNP}}$ of languages accepted in linear time by nondeterministic reversal-bounded Turing machines.

In § 1 the basic notation and preliminary notions are outlined. The characterization of the class of context-sensitive languages is the central idea in § 2. In § 3 the characterization of PSPACE and its comparison to NP are established.

1. Basic concepts. It is assumed that the reader is familiar with the basic concepts from the theories of automata, computability, and formal languages. Some of the concepts that are most important for this paper are reviewed here and notation is established.

For a string w, $|w|$ denotes the *length* of w: if e is the empty string, then $|e| = 0$; if a is a symbol and y is a string, then $|ay| = 1 + |y|$.

The *reversal* w^R of a string w is the string obtained by writing w in reverse order: $e^R = e$; for any symbol a, $a^R = a$; if a is a symbol and y is a string, then $(ay)^R = y^R a$. For a string w, $w^1 = w$.

* Received by the editors June 23, 1978 and in revised form October 9, 1978.

† Department of Mathematics, University of California at Santa Barbara, Santa Barbara, California 93106. This research was supported in part by the National Science Foundation under Grant MCS77-11360. Some of these results were reported at the Tagung über Formale Sprachen, Mathematisches Forschungsinstitut, Oberwolfach, West Germany, August 1978.

For an acceptor M, the language accepted by M is denoted by $L(M)$.

Recall that a *homomorphism* (between free monoids) is a function $h: \Sigma^* \to \Delta^*$ such that for all $x, y \in \Sigma^*$, $h(xy) = h(x)h(y)$. A homomorphism $h: \Sigma^* \to \Delta^*$ is *nonerasing* if $|w| > 0$ implies $|h(w)| > 0$ and is *length-preserving* if for all $w \in \Sigma^*$, $|h(w)| = |w|$. A homomorphism $h: \Sigma^* \to \Delta^*$ is *linear-erasing on language* $L \subseteq \Sigma^*$ if there is a constant $k > 0$ such that for all $w \in L$ with $|w| \geqq k$, $|w| \leqq k|h(w)|$, and is *polynomial-erasing on language* $L \subseteq \Sigma^*$ if there is a constant $k > 0$ such that for all $w \in L$ with $|w| \geqq k$, $|w| \leqq |h(w)|^k$. A class $\mathscr{L}$ of languages is *closed under* (*nonerasing, linear-erasing, polynomial-erasing*) *homomorphism* if for every language $L \in \mathscr{L}$ and any homomorphism h (that is nonerasing, linear-erasing on L, polynomial-erasing on L), $h(L) = \{h(w) \mid w \in L\}$ is in $\mathscr{L}$.

Let n be a positive integer and let ρ be a function from $\{1, \cdots, n\}$ to $\{1, R\}$. Let L be a language and let $h_1, \cdots, h_n$ be homomorphisms. The language $\langle \rho; h_1, \cdots, h_n \rangle(L) = \{h_1(w)^{\rho(1)} \cdots h_n(w)^{\rho(n)} \mid w \in L\}$ is a *homomorphic replication of type* ρ *on* L. Let $\mathscr{L}$ be a class of languages. If for every $n > 0$, every function $\rho: \{1, \cdots, n\} \to \{1, R\}$, every language $L \in \mathscr{L}$, and every n homomorphisms $h_1, \cdots, h_n$, each of which is nonerasing (linear-erasing on L, polynomial-erasing on L), the language $\langle \rho; h_1, \cdots, h_n \rangle(L)$ is in $\mathscr{L}$, then $\mathscr{L}$ is *closed under nonerasing* (respectively, *linear-erasing, polynomial-erasing*) *homomorphic replication.*

Clearly a class of langiages closed under (nonerasing, linear-erasing, polynomial-erating) homomorphic replication is closed under (nonerasing, linear-erasing, polynomial-erasing) homomorphism.

Recall that a *semi-AFL* is a nonempty class of languages closed under union, inverse homomorphism, nonerasing homomorphism, and intersection with regular sets.

Let f be an integer-valued function. Let $\mathrm{NTIME}(f) = \{L(M) \mid M$ is a nondeterministic Turing machine that operates within time $f(n)$, where n is the length of the input string$\}$ and let $\mathrm{NP} = \bigcup_{k \geqq 1} \mathrm{NTIME}(n^k)$, so that NP is the class of languages accepted in polynomial time by nondeterministic Turing machines. Let $\mathrm{DSPACE}(f) = \{L(M) \mid M$ is a deterministic Turing machine that uses at most $f(n)$ work space, where n is the length of the input string$\}$, let $\mathrm{NSPACE}(f) = \{L(M) \mid M$ is a nondeterministic Turing machine that uses at most $f(n)$ work space$\}$, and let $\mathrm{PSPACE} = \bigcup_{k \geqq 1} \mathrm{DSPACE}(n^k)$ so that PSPACE is the class of languages accepted by Turing machines that use at most polynomial work space.

There is one class of languages that is particularly useful in this study. Let $\mathscr{L}_{\mathrm{BNP}}$ be the class of languages accepted in linear time by nondeterministic Turing machines whose work tapes are reversal-bounded (i.e., there is a fixed constant that bounds the number of times a read-write head can change directions during any computation). It is known [3] that a language L is in $\mathscr{L}_{\mathrm{BNP}}$ if and only if there are three linear context-free languages L_1, L_2, L_3 and a nonerasing homomorphism h such that $L = h(L_1 \cap L_2 \cap L_3)$, and that $\mathscr{L}_{\mathrm{BNP}}$ is the smallest intersection-closed semi-AFL containing $\{wcw^R \mid w \in \{a, b\}^*\}$.

2. A representation theorem. In this section we develop the first result, a representation theorem showing how to obtain the class of context-sensitive languages from the class of regular sets by using algebraic closure operations. To accomplish this it is necessary to discuss relations on strings and their encodings as languages.

Consider n-ary relations on strings. If R is a binary relation on strings over the alphabet Σ, then the *transitive closure* of R is $R^* = \{\langle x, y \rangle \mid x, y \in \Sigma^*$ and either $x = y$ or there exist $n \geqq 1$ and $z_0 \cdots z_n \in \Sigma^*$ such that $z_0 = x$, $z_n = y$, and for each $i = 1, \cdots, n$, $R(z_{i-1}, z_i)$ holds$\}$. A binary relation R is *length-preserving* if for all x, y, when $R(x, y)$ holds, then $|x| = |y|$.

Let R be an n-ary relation on strings over the alphabet Σ. Let $\#$ be a symbol not in Σ. The language $SE_{\#}(R) = \{w_1 \# \cdots \# w_n \mid \text{for } i = 1, \cdots, n,\ w_i \in \Sigma^*;\ R(w_1, \cdots, w_n) \text{ holds}\}$ is the *sequential #-encoding of R*.

By using sequential encodings, relations can be interpreted as languages. For example, the concatenation relation over an alphabet Σ gives rise to the language $\{x \# y \# z \mid x, y, z \in \Sigma^* \text{ and } xy = z\}$, where $\#$ is a symbol not in Σ.

We are interested in interpreting a language as an encoding of a binary relation. Let L be a language and let Σ be a finite alphabet such that $L \subseteq \Sigma^*$. For any $a \in \Sigma$ the *binary relation a-encoded by L* is $R_a(L) = \{\langle x, y\rangle \mid x, y \in (\Sigma - \{a\})^* \text{ and } xay \in L\}$.

Notice that $SE_a(R_a(L)) = (\Sigma - \{a\})^*\{a\}(\Sigma - \{a\})^* \cap L$ and that if T is a binary relation on strings over Σ and $\# \notin \Sigma$, then $R_{\#}(SE_{\#}(T)) = T$.

To say that a relation R is transitively closed is to say that $R^* = R$. Here we develop the notion of "a class of languages being transitively closed" by considering the relations encoded by the languages in the class.

Let $\mathscr{L}$ be a class of languages. From a language L in $\mathscr{L}$, we consider the relation $R_a(L)$ a-encoded by L. Then we take the transitive closure $R_a^*(L)$ of $R_a(L)$ and consider the language $SE_a(R_a^*(L))$, that is, the sequential a-encoding of the relation $R_a^*(L)$. For our purposes it is sufficient to restrict attention to the cases where $R_a(L)$ is length-preserving and in that case to demand that the language $SE_a(R_a^*(L))$ is in $\mathscr{L}$. More formally, we have the next definition.

A class $\mathscr{L}$ of languages is *weakly transitively closed* if the following condition holds: Let L be any language in $\mathscr{L}$, let Σ be the smallest finite alphabet such that $L \subseteq \Sigma^*$, and let a be a symbol in Σ. If $R_a(L)$ is length-preserving, then $SE_a(R_a^*(L))$ is in $\mathscr{L}$.

Now the representation of the class of context-sensitive languages can be established.

THEOREM 1. *The class of context-sensitive languages* (NSPACE(n)) *is the smallest class of languages that contains all of the regular sets, is closed under intersection and linear-erasing homomorphic replication, and is weakly transitively closed. It is the smallest semi-AFL that is closed under intersection and nonerasing homomorphic replication and is weakly transitively closed.*

Proof. Let $\mathscr{L}$ be the smallest class of languages that contains all of the regular sets, is closed under intersection and linear-erasing homomorphic replication, and is weakly transitively closed. It is clear that the class of context-sensitive languages contains all of the regular sets and is closed under intersection and linear-erasing homomorphic replication. From the work of Jones [4], [5] and of McCloskey [6], it is clear that the class of context-sensitive languages is weakly transitively closed. Since $\mathscr{L}$ is taken to be the smallest class with these properties, every language in $\mathscr{L}$ is a context-sensitive language.

To show that every context-sensitive language is in $\mathscr{L}$, we use the fact that a language is context-sensitive if and only if it is accepted by a nondeterministic linear bounded automaton (LBA), a one-head one-tape Turing machine that uses only those tape squares where the input initially appears. We will show that if M is a nondeterministic LBA, then the set $L(M)$ of strings accepted by M is in $\mathscr{L}$. It will be useful to note that $\mathscr{L}_{\text{BNP}}$ is the smallest class of languages that contains all of the regular sets and is closed under intersection and linear-erasing homomorphic replication [1], so that $\mathscr{L}_{\text{BNP}} \subseteq \mathscr{L}$.

Let M be a nondeterministic LBA. Let Q be the set of internal states of M, let Σ be M's input alphabet, and let Δ be the set of symbols that M writes on its tape during a computation. Without loss of generality, assume that when M's read-write head leaves a tape square it has already written a symbol from Δ in that tape square and assume that $\Delta \cap \Sigma = \varnothing$. With these assumptions an instantaneous description in a computation of M

can be viewed as a string in $\Delta^*Q\Delta^*\Sigma^*$ and the set of initial instantaneous descriptions is $\{q_0\}\Sigma^*$ where q_0 is the initial state. Assuming that M must read its entire input before accepting, the set of accepting instantaneous descriptions is $\Delta^*F\Delta^*$ where $F \subseteq Q - \{q_0\}$ is the set of accepting states.

Let δ_M be the "yield" relation on strings induced by M's transition function, that is, $\delta_M(\alpha, \beta)$ holds if and only if α and β are potentially instantaneous descriptions in some computation of M and α yields β according to M's transition function. Since M is an LBA, δ_M is a length-preserving relation.

Let $\Gamma = \Delta \cup \Sigma \cup Q$ and let $\neq$ be a new symbol not in Γ. It is clear that the language $SE_{\neq}(\delta_M) = \{\alpha \neq \beta \mid \alpha, \beta \in \Delta^*Q\Delta^*\Sigma^*$ and $\delta_M(\alpha, \beta)$ holds$\}$ can be recognized in linear time by a nondeterministic Turing machine with work tapes that are reversal-bounded, and so $SE_{\neq}(\delta_M)$ is in $\mathscr{L}_{\text{BNP}}$. Since $\mathscr{L}_{\text{BNP}} \subseteq \mathscr{L}$, we see that $SE_{\neq}(\delta_M)$ is in $\mathscr{L}$.

Let δ_M^* be the transitive closure of δ_M. Since δ_M is a length-preserving binary relation and $SE_{\neq}(\delta_M)$ is in $\mathscr{L}$, the language $SE_{\neq}(\delta_M^*)$ is in $\mathscr{L}$ because $\mathscr{L}$ is weakly transitively closed. Clearly the language $L_0 = \{\alpha \neq \beta \mid \alpha$ is an initial instantaneous description of M and β is an accepting instantaneous description of $M\}$ is accepted in linear time by a nondeterministic Turing machine with reversal-bounded work tapes, so that $L_0 \in \mathscr{L}_{\text{BNP}} \subseteq \mathscr{L}$. Since $\mathscr{L}$ is closed under intersection, $L_1 = L_0 \cap SE_{\neq}(\delta_M^*)$ is in $\mathscr{L}$. Clearly a string $\alpha \neq \beta$ is in L_1 if and only if α is an initial instantaneous description of M and β is an accepting instantaneous description of M that occurs in some computation of M that begins with α.

Let $h : (\Gamma \cup \{\neq\})^* \to \Sigma^*$ be the homomorphism determined by defining $h(a) = a$ for $a \in \Sigma$ and $h(a) = e$ for $a \in (\Gamma \cup \{\neq\}) - \Sigma$. By the assumptions made above, if $\alpha \neq \beta \in L_1$, then $h(\alpha \neq \beta)$ is an input string accepted by M, $|h(\alpha \neq \beta)| = |\alpha| - 1$, and $|\alpha \neq \beta| \leqq 3|h(\alpha \neq \beta)|$. Also, if w is any string accepted by M, then there exists $z \in \Gamma^*$ such that $q_0 w \neq z \in L_1$ and $h(q_0 w \neq z) = w$. Thus h is linear-erasing on L_1 and $h(L_1)$ is the set $L(M)$ of strings accepted by M.

Since $\mathscr{L}$ is closed under linear-erasing homomorphic replication and $L_1 \in \mathscr{L}$, $L(M) = h(L_1)$ is in $\mathscr{L}$.

Thus every context-sensitive language is in $\mathscr{L}$.

The second characterization follows from the first by using the techniques employed in [1]. □

Professor Jonathan Goldstine (personal communication) has made the following observation. In the proof of Theorem 1, the operations of homomorphic replication and intersection were used to show that $SE_{\neq}(\delta_M)$ is in $\mathscr{L}_{\text{BNP}}$ and $\mathscr{L}_{\text{BNP}} \subseteq \mathscr{L}$. Closure under linear-erasing homomorphic replication was used to show that $h(L_1)$ is in $\mathscr{L}$ and closure under intersection was used to show that L_1 is in $\mathscr{L}$. If we let $L_2 = \{\sigma \neq \beta^R \mid \alpha \neq \beta \in SE_{\neq}(\delta_M)\}$ then L_2 is a linear context-free language. Consider the relation $R_{\neq}(L_2) = \{\langle x, y\rangle \mid x \neq y \in L_2\}$. Clearly, the transitive closure $(R_{\neq}(L_2))^*$ of the relation $R_{\neq}(L_2)$ is only a minor variation of δ_M^*. Also, $L_0 = \{\alpha \neq \beta^R \mid \alpha$ is an initial instantaneous description of M and β is an accepting instantaneous description of $M\}$ is a regular set. Normalizing M so that an accepting computation must have an odd number of steps, we see that $h(L_0 \cap SE_{\neq}(R_{\neq}^*(L_2))) = L(M)$. This yields the following characterization on NSPACE(n).

PROPOSITION. The class of context-sensitive languages (NSPACE(n)) is the smallest class of languages that contains all of the linear context-free languages, is closed under intersection with regular sets and under linear-erasing homomorphism, and is weakly transitively closed.

Professor Peter Deussen (personal communication) has pointed out that it is not necessary to restrict attention to length-preserving relations but only to relations that are nonincreasing, e.g., if $R(x, y)$ holds, then $|x| \geqq |y|$.

Consider the following classes of languages:

(i) MULTI-RESET is the smallest class of languages containing $\{wcw \mid w \in \{a, b\}^*\}$ and closed under intersection and linear-erasing homomorphic duplication (where homomorphic duplication is the restriction of homomorphic replication obtained by specifying $\rho(i) = 1$ for every i) [3];

(ii) $\mathscr{L}_{\text{BNP}}$ has been discussed previously;

(iii) NTIME(n) is the class of languages accepted in linear time by nondeterministic multitape Turing machines;

(iv) RUD is the class of rudimentary languages, the smallest class of languages containing $\{xcycz \mid x, y, z \in \{a, b\}^*$ and $xy = z\}$ and closed under the Boolean operations, nonerasing homomorphism, inverse homomorphism, and intersection with regular sets [9].

Now it is known [2], [9] that MULTI-RESET $\subseteq \mathscr{L}_{\text{BNP}} \subseteq$ NTIME(n) $\subseteq$ RUD $\subseteq$ DSPACE(n) $\subseteq$ NSPACE(n). From the proof of Theorem 1 and the properties of these classes we see that if $\mathscr{L}$ is any one of the classes MULTI-RESET, $\mathscr{L}_{\text{BNP}}$, NTIME(n), RUD, or DSPACE(n), then $\mathscr{L} =$ NSPACE(n) if and only if $\mathscr{L}$ is weakly transitively closed.

In the proof of Theorem 1, consider the case of M being a deterministic LBA. Then for each possible instantaneous description α, there is at most one instantaneous description β such that $\delta_M(\alpha, \beta)$ holds. Jones [4], [5] discusses "transitive closure of deterministic relations" and it is easy to characterize the class DSPACE(n) in a manner similar to Theorem 1 by restricting attention to "deterministic relations" that are length-preserving. This leads to the formal definition of a property θ such that MULTI-RESET or $\mathscr{L}_{\text{BNP}}$ or NTIME(n) or RUD has θ if and only if that class is equal to DSPACE(n).

3. Characterizing PSPACE.

THEOREM 2. *The class PSPACE of languages accepted in polynomial space by Turing machines is the smallest class of languages that contains all of the regular sets, is closed under intersection and polynomial-erasing homomorphic replication, and is weakly transitively closed. It is the smallest semi-AFL that is closed under intersection and polynomial-erasing homomorphic replication and that is weakly transitively closed.*

Proof. It is clear that PSPACE is closed under these operations. If $L_1 \in$ PSPACE, then there is a language $L_2 \in$ NSPACE(n), and a homomorphism h such that $h(L_2) = L_1$ and h is polynomial-erasing on L_2. Thus the closure of NSPACE(n) under polynomial-erasing homomorphism or under polynomial-erasing homomorphic replication is PSPACE. The result now follows from Theorem 1. □

Recall that the class NP of languages accepted in polynomial time by nondeterministic Turing machines is the smallest class containing all of the regular sets and closed under intersection and polynomial-erasing homomorphic replication [1]. Thus we have the following result.

THEOREM 3. *The class* NP *is weakly transitively closed if and only if* NP = PSPACE.

One of the referees has observed that the language $SE_{\#}(\delta_M)$ used in the proof of Theorem 1 is in the class P of languages accepted in polynomial time by deterministic Turing machines, and thus P is weakly transitively closed if and only if $P =$ PSPACE.

The class EXRUD of extended rudimentary languages is the smallest class of languages containing $\{xcycz \mid x, y, z \in \{a, b\}^*$ and $xy = z\}$ and closed under the Boolean operations, polynomial-erasing homomorphism, inverse homomorphism, and intersection with regular sets [8]. It is also the union of the classes in the polynomial-time hierarchy [7], [8]. It is known that NP $\subseteq$ EXRUD $\subseteq$ PSPACE. From Theorem 2 and the

characterization of EXRUD given in [1], we see that EXRUD is weakly transitively closed if and only if EXRUD = PSPACE.

For the characterizations given in Theorems 1 and 2, it is sufficient to use the operation of homomorphic duplication (the restriction of homomorphic replication to the case of $\rho(i) = 1$ for all i) instead of homomorphic replication [2]. We have used homomorphic replication in order to be consistent with [1] and to avoid the construction of more machinery.

It is clear that Theorem 1 can be used to characterize any class of languages specified by space-bounded Turing machines if the class of bounding functions is closed under composition and if each bounding function grows at least as fast as a linear function. In this case the amount of erasing allowed by the homomorphism must be bounded by the space bounds.

REFERENCES

[1] R. BOOK, *Simple representations of certain classes of languages*, J. Assoc. Comput. Mach., 25 (1978), pp. 23–31.

[2] R. BOOK, S. GREIBACH AND C. WRATHALL, *Reset machines*, submitted for publication.

[3] R. BOOK, M. NIVAT AND M. PATERSON, *Reversal-bounded acceptors and intersections of linear languages*, this Journal, 3 (1974), pp. 283–295.

[4] N. JONES, *Formal languages and rudimentary attributes*, Ph.D. dissertation, University of Western Ontario, London, Ontario, 1967.

[5] ———, *Classes of automata and transitive closure*, Information and Control, 13 (1968), pp. 207–229.

[6] T. MCCLOSKEY, *Abstract families of length-preserving processors*, J. Comput. System Sci., 10 (1975), pp. 394–427.

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

[8] C. WRATHALL, *Subrecursive predicates and automata*, Ph.D. dissertation, Harvard University, Cambridge, MA, 1975.

[9] ———, *Rudimentary predicates and relative computation*, this Journal, 7 (1978), pp. 194–209.

SIAM J. COMPUT.
Volume 8, No. 3, August 1979

0097-5397/79/0803-0016$01.00/0

ON THE EXPECTED VALUE OF A RANDOM ASSIGNMENT PROBLEM*

DAVID W. WALKUP†

Abstract. Given an n by n matrix X, the assignment problem asks for a set of n entries, one from each column and row, with the minimum sum. It is shown that the expected value of this minimum sum is less than 3, independent of n, if X consists of independent random variables uniformly distributed from 0 to 1.

Key words. assignment problem, average value, probabilistic method, sparse graphs, minimum weight matching

Given any $n \times n$ matrix $X = \{X_{ij}\}$, the so called assignment problem asks for the permutation $\sigma^*: \{1, \cdots, n\} \to \{1, \cdots, n\}$ such that

$$\alpha(X) = \sum_{i=1}^{n} X_{i\sigma^*(i)} = \min_{\sigma} \sum_{i=1}^{n} X_{i\sigma(i)}.$$

Obviously the assignment problem can be formulated as a problem on a complete bipartite graph K_{nn} with n nodes in each class. The number X_{ij} becomes a weight on the edge from node i of the first class to node j of the second, an assignment σ becomes a matching (a set of n disjoint edges spanning the $2n$ nodes of K_{nn}), and $\alpha(X)$ is the value of a minimum weight matching. The object of this note will be to prove the following.

THEOREM. *If $\{X_{ij}\}$ are independent uniform $[0, 1]$ random variables, then $E\{\alpha(X)\} \leqq 3$.*

Surprisingly little seems to have been published about this aspect of the assignment problem. Kurtzberg [2] gives some simple arguments to show $n/(n+1) \leqq E\{\alpha(X)\} \leqq \ln n$. Donath [1] gives experimental evidence that $E\{\alpha(X)\}$ is increasing with n, approaching a limit of about 1.6. He also describes an algorithm for producing good, but nonoptimal, solutions to the assignment problem and analyses its average behavior to derive the bound $E\{\alpha(X)\} \leqq 2.37\ldots$. However, there are some subtle difficulties with the description of the algorithm and the accompanying analysis. (Specifically, the operation of earlier steps tend to condition the probabilities of events at later steps of the algorithm.) The proof of the Theorem given here avoids these difficulties, but at the expense of replacing Donath's practical algorithmic approach with the following result on sparse graphs proved in [3] using nonconstructive probabilistic methods.

LEMMA 1. *Let $P(n, d)$ be the probability of a matching in a graph selected uniformly from the class $\mathbf{G}(n, d)$ of directed bipartite digraphs with n nodes in each class and outward degree d at each node. Then*

$$1 - P(n, 2) \leqq (5n)^{-1},$$

$$1 - P(n, d) \leqq (122)^{-1} \left(\frac{d}{n}\right)^{(d+1)(d-2)} \qquad \text{for } d \geqq 3.$$

To begin the proof of the Theorem, let $Y = \{Y_{ij}\}$ and $Z = \{Z_{ij}\}$ be $n \times n$ matrices of independent identically distributed random variables with common distribution function

$$F(\lambda) = \Pr\{Y_{ij} \leqq \lambda\} = 1 - (1-\lambda)^{1/2} \quad \text{for } 0 \leqq \lambda \leqq 1.$$

* Received by the editors January 3, 1978 and in revised form September 20, 1978.

† Department of Computer Science, Washington University, St. Louis, Missouri. Now at Department of Applied Physics and Information Science, University of California at San Diego, La Jolla, California 92093.
This research was supported in part by the Office of Naval Research under Contract NR 044-437.

Then $X_{ij} = \min(Y_{ij}, Z_{ij})$ are independent and uniformly distributed on $[0, 1]$, and it suffices to prove the Theorem for this particular set of random variables. Observe that $F(\lambda) \geqq H(\lambda) = \frac{1}{2}\lambda$, where H is the distribution function for a random variable uniform on $[0, 2]$. If $Y_{i,(k)}$ and $U_{(k)}$ denote respectively the kth smallest element in the ith row of Y and the kth smallest of n uniform variables on $[0, 1]$ then

$$E\{Y_{i,(k)}\} \leqq 2\, E\{U_{(k)}\} = \frac{2k}{n+1}. \tag{1}$$

(Note that there is a small technical difficulty here. There is no trouble with the definition of $Y_{i,(k)}$, but the *location* of this value in row i of Y is clearly ambiguous when ties occur. This difficulty is overcome easily in all that follows either by understanding all statements as holding *almost surely* or by deleting the tie set initially from the underlying probability space.)

For $1 \leqq d \leqq n$, let G_d be the random directed bipartite graph on $S = \{s_1, \cdots, s_n\}$ and $T = \{t_1, \cdots, t_n\}$ where

(s_i, t_j) is an arc of G_d if and only if Y_{ij} is one of the d smallest elements in row i of Y.

(t_j, s_i) is an arc of G_d if and only if Z_{ij} is one of the d smallest elements in column j of Z.

For each d, G_d is a member of $\mathbf{G}(n, d)$ and takes on values in $\mathbf{G}(n, d)$ with equal probability, so that Lemma 1 applies. Moreover, these random graphs satisfy

$$G_1 \subset G_2 \subset \cdots \subset G_n = K'_{nn},$$

where K'_{nn} is the complete directed bipartite graph on S and T.

Let $\mathbf{M}$ denote the set of all subgraphs of K'_{nn} containing at least one matching. For each d let M_d be a partial function over $\mathbf{G}(n, d)$ such that $M_d(G)$ is an (arbitrary) matching in G if one exist, and let α_d be the (random) sum of weights of the (random) matching $M_d(G_d)$.

Elementary reasoning shows

$$\begin{aligned} E\{\alpha(X)\} \leqq\ & E\{\alpha_2 | G_2 \in \mathbf{M}\} \cdot \Pr\{G_2 \in \mathbf{M}\} \\ & + E\{\alpha_3 | G_3 \in \mathbf{M},\ G_2 \notin \mathbf{M}\} \cdot \Pr\{G_2 \notin \mathbf{M}\} \\ & + E\{\alpha_n | G_3 \notin \mathbf{M}\} \cdot \Pr\{G_3 \notin \mathbf{M}\}. \end{aligned}$$

(Note that the conditioning events avoid any difficulty with the domain of the partial function α_d.) Substitution of estimates from Lemmas 1 and 2 yields the required inequality of the Theorem:

$$\begin{aligned} E\{\alpha(X)\} &\leqq \frac{3n}{n+1} \cdot 1 + \frac{6n}{n+1} \cdot \frac{1}{5n} + \frac{n(n+4)}{n+1} \cdot \frac{3^4}{122\, n^4} \\ &\leqq 3 \quad \text{if } n \geqq 2. \end{aligned}$$

LEMMA 2. *For* $1 \leqq d' \leqq d \leqq n$,

$$\Pr\{G_d \in \mathbf{M}\} = P(n, d) > 0, \qquad E\{\alpha_d | G_d \in \mathbf{M}\} \leqq (d+1)n/(n+1),$$

$$E\{\alpha_d | G_d \in \mathbf{M},\ G_{d'} \notin \mathbf{M}\} \leqq (d+d'+1)n/(n+1) \quad \textit{if } P(n, d') < 1.$$

Proof. The first equation is elementary. (It shows incidentally that the conditional expectation in the second equation is well defined. However, if d' is sufficiently close to n, the third equation can be vacuous.) The validity of the second equation hinges on the

fact that $M_d(G_d)$ depends only on G_d and not upon Y or Z directly. If some arc (s_i, t_j) (respectively (t_j, s_i)) is in $M_d(G_d)$, then Y_{ij} (respectively Z_{ij}) has equal probability of being any of $Y_{i,(1)}, \cdots, Y_{i,(d)}$ (respectively $Z_{(1),j}, \cdots, Z_{(d),j}$). The expected value of the rank will be $(d+1)/2$. This, combined with (1) yields the second equation of the lemma. The proof of the third equation is similar, but for each arc (s_i, t_j) of $M_d(G_d)$ which is not in G_d, the expected rank of Y_{ij} in $Y_{i,(1)}, \cdots, Y_{i,(d)}$ will be $(d+d'+1)/2$.

REFERENCES

[1] W. E. DONATH. *Algorithm and average-value bounds for assignment problems.* IBM J. Res. Develop., 13 (1969), pp. 380–386.

[2] J. M. KURTZBERG. *On approximation methods for the assignment problem*, J. Assoc. Comput. Mach., 9(1962) pp. 419–439.

[3] D. W. WALKUP. *Matchings in random regular bipartite digraphs*, submitted.

SIAM J. COMPUT.
Volume 8, No. 3, August 1979

0097-5397/79/0803-0017$01.00/0

OPTIMAL EVALUATION OF PAIRS OF BILINEAR FORMS*

JOSEPH JA' JA'†

Abstract. A large class of multiplication problems in arithmetic complexity can be viewed as the simultaneous evaluation of a set of bilinear forms. This class includes the multiplication of matrices, polynomials, quaternions, Cayley and complex numbers. Considering bilinear algorithms, the optimal number of nonscalar multiplications can be described as the rank of a three-tensor or as the smallest member of rank one matrices necessary to include a given set of matrices in their span.

In this paper, we attack a rather large subclass of three-tensors, namely that of $(p, q, 2)$-tensors, for arbitrary p and q, and solve it completely in the case where the field of constants contains the roots of a polynomial associated with the given tensor. In all other cases, we prove that, in general, our bounds cannot be improved. The complexity of a general pair of bilinear forms is determined explicitly in terms of parameters related to Kronecker's theory of pencils and to the theory of invariant polynomials. This reveals unexpected results and shows explicitly the dependence on the algebraic structure of the constants; we display, for example, a pair of 3×3 bilinear forms whose complexity is 3 over the field Z_7 and which, however, requires exactly 4 nonscalar multiplications over the fields Z_5 or Z_{11}. Corresponding optimal algorithms are described and several applications are considered.

Key words. algebraic complexity, bilinear forms, tensor rank

1. Introduction.

Computational complexity is concerned with the analysis of the intrinsic time and space requirements of computational problems. Given a class of problems, one starts by determining computational models and complexity measures suitable for this class. The complexities of specific problems of interest belonging to this class are then studied, coupled with the search for optimal algorithms. In most cases, this is a very difficult task and only very few cases have been solved completely.

In arithmetic complexity, we consider algebraic problems, typically function evaluation, and try to find the number of arithmetic operations needed by any algorithm and algorithms which achieve these bounds. We ordinarily consider the *straight-line arithmetic programs* which consist of a sequence of instructions of the form

$$\begin{aligned} f_1 &\leftarrow a_1 \circ b_1, \\ f_2 &\leftarrow a_2 \circ b_2, \\ &\vdots \\ f_l &\leftarrow a_l \circ b_l, \end{aligned}$$

where $\circ$ is an arithmetic operation from $\{+, -, \times, \div\}$, a_i, b_i either belong to $K \cup \{\text{indeterminates}\}$, K a given "set of constants", or are previously computed f_j's; this program computes P if there exists i, $1 \leqq i \leqq l$, such that $P = f_i$.

A large class of multiplication problems, like the multiplication of matrices, polynomials, quaternions, Cayley and complex numbers, can be viewed as the simultaneous evaluation of a set of bilinear forms; consider, for example, the 2×2 matrix multiplication problem:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11}+a_{12}b_{21} & a_{11}b_{12}+a_{12}b_{22} \\ a_{21}b_{11}+a_{22}b_{21} & a_{21}b_{12}+a_{22}b_{22} \end{bmatrix},$$

and the problem can be thought of as the simultaneous evaluation of the four bilinear

* Received by the editors November 8, 1977, and in final revised form November 10, 1978.

† Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania 16802. This work was supported by the U.S. Office of Naval Research under the Joint Services Electronics Program by Contract N00014-75-C-0648, Division of Engineering and Applied Physics, Harvard University, Cambridge, MA 02138.

forms: $B_1 = a_{11}b_{11} + a_{12}b_{21}$, $B_2 = a_{11}b_{12} + a_{12}b_{22}$, $B_3 = a_{21}b_{11} + a_{22}b_{21}$ and $B_4 = a_{21}b_{12} + a_{22}b_{22}$ with indeterminates $= \{a_{11}, a_{12}, a_{21}, a_{22}\} \cup \{b_{11}, b_{12}, b_{21}, b_{22}\}$.

In general, the problem can be defined as follows: let K be a commutative ring and let $x = (x_1, x_2, \cdots, x_p)^T$ and $y = (y_1, y_2, \cdots, y_q)^T$ be two column vectors of indeterminates; we have to compute m bilinear forms:

$$B_i = \sum_{j=1}^{p} \sum_{k=1}^{q} \gamma_{ijk} x_j y_k = x^T G_i y, \qquad i = 1, 2, \cdots, m, \qquad \gamma_{ijk} \in K,$$

where G_i is a $p \times q$ matrix with elements in K. In the case when the indeterminates do not commute, one can prove [26], [27] that it is no loss of generality to restrict the straight-line programs to, what is called in the literature [7], [14], noncommutative *bilinear programs*, where $\circ \in \{+, -, \times\}$ and each instruction $f_i \leftarrow a_i \circ b_i$ is either

(i) an addition or subtraction or

(ii) a *scalar multiplication*, i.e., either a_i or $b_i \in K$ or

(iii) a multiplication of a linear form in x by a linear form in y over K (*nonscalar multiplication*).

The multiplicative complexity of a bilinear program is defined to be the number of nonscalar multiplications used in step (iii); the reason we ignore the scalar multiplications is that (i) K can be chosen so that multiplications by elements of K are particularly easy, (ii) since the algorithms are noncommutative, we can replace x_i's and y_j's by matrices in the same way Strassen's algorithm to multiply 2×2 matrices is generalized to arbitrary $n \times n$ matrix multiplication. Therefore our goal is reduced to finding a bilinear algorithm using the fewest number of nonscalar multiplications for a given problem. It is apparent from above that if δ is the optimal number of (nonscalar) multiplications needed to evaluate the bilinear forms $\{B_i\}_{i=1}^m$, denoted by $\delta_K\{G_i\}$, then δ is the smallest number such that

$$B_i = \sum_{k=1}^{\delta} a_{ik} r_k(x) r'_k(y) \quad \text{where } a_{ik} \in K,$$

$r_k(x)$ is a linear form in x, say $r_k(x) = \langle b_k, x \rangle$, and $r'_k(y)$ is a linear form in y, say $r'_k(y) = \langle c_k, y \rangle$; hence the above expression can be rewritten as

$$\begin{aligned} B_i = x^T G_i y &= \sum_{k=1}^{\delta} a_{ik} \langle b_k, x \rangle \langle c_k, y \rangle \\ &= \sum_{k=1}^{\delta} a_{ik} x^T b_k c_k^T y = x^T \Big(\sum_{k=1}^{\delta} a_{ik} b_k c_k^T \Big) y, \qquad i = 1, 2, \cdots, m. \end{aligned}$$

Since the above equality must hold for all values of the indeterminates x and y over K, we conclude that

$$G_i = \sum_{k=1}^{\delta} a_{ik} b_k c_k^T, \qquad i = 1, 2, \cdots, m.$$

Since a matrix is of rank one if, and only if, it can be written as the outer product of two vectors, we see that the optimal number δ is equal to the smallest number of rank one matrices necessary to include the G_i's in their span [4], [6], [7], [20]. Another interesting formulation given in [4], [6] is to introduce a set of indeterminates $\{s_i\}_{i=1}^m$ and to consider the trilinear form, called the *defining function*,

$$h(s, x, y) = \sum_{i=1}^{m} s_i B_i = \sum_{i=1}^{m} \sum_{j=1}^{p} \sum_{k=1}^{q} \gamma_{ijk} s_i x_j y_k.$$

It is easy to see that δ is the smallest number such that

$$h(s, x, y) = \sum_{l=1}^{\delta} \langle a_l, s\rangle\langle b_l, x\rangle\langle c_l, y\rangle$$

and we now have a completely symmetric problem with respect to s, x and y; for example, the above problem is equivalent to the evaluation of the p bilinear forms associated with the $m \times q$ matrices

$$G_j = (\gamma_{ijk}), \qquad j = 1, 2, \cdots, p.$$

Because of this *triality*, we can talk about the (m, p, q) problem without any ambiguity. A partial aspect of this property, called *duality*, was discovered by Hopcroft and Musinski [14] and Probert [24] independently. As an immediate corollary, the complexity of multiplying an $m \times n$ matrix by an $n \times p$ matrix is the same as that of multiplying an $m \times p$ matrix by a $p \times n$ matrix for example, and we talk about the (m, n, p) matrix multiplication problem.

Strassen in [26] observed that the optimal number δ can be defined as *the rank* or *length* of the (m, p, q) tensor (γ_{ijk}) i.e., the smallest number such that

$$(\gamma_{ijk}) = \sum_{l=1}^{\delta} a_l \otimes b_l \otimes c_l, \qquad a_l \in K^m, \quad b_l \in K^p, \quad c_l \in K^q, \quad l = 1, 2, \cdots, \delta.$$

Another interesting observation made in ([4], [6]), and which we will find it often useful, is that of the *characteristic matrix* $G(s) = \sum_{i=1}^{m} s_i G_i$, where $\{s_i\}_{i=1}^{m}$, as before, is a set of indeterminates. Note that the defining function $h(s, x, y)$ is given by: $h(s, x, y) = \langle x, G(s)y\rangle$ from which we can get two equivalent characteristic matrices $\hat{G}(y)$ and $\check{G}(x)$ defined as follows:

$$h(s, x, y) = \langle x, G(s)y\rangle = \langle s, \hat{G}(y)x\rangle = \langle y, \check{G}(x)s\rangle.$$

We use $\delta_K\{G(s)\}$, *the degree* of $G(s)$, to mean the complexity of the associated set of bilinear forms over K.

Throughout this paper, we assume that we are dealing with K-nondegenerate multiplication problems, i.e., no nontrivial K-linear combinations of the type $\sum_k \alpha_k \gamma_{ijk}$ or $\sum_j \beta_j \gamma_{ijk}$ or $\sum_i \tau_i \gamma_{ijk}$ vanish.

This problem has been studied by several authors ([4], [6], [7], [12], [13], [14], [15], [20]) and only few results are available about the general case. The special cases of multiplication of matrices, polynomials, quaternions and complex-numbers have received particular attention [5], [7], [10], [14], [16], [19], [25], [28] and we have several interesting results concerning these specific cases although the matrix multiplication problem, which has motivated this type of research, remains wide open.

In this paper, we attack a rather large subclass of bilinear forms, namely those corresponding to $(p, q, 2)$-tensors for arbitrary p and q. There is a mathematical theory which dates back to 1890 and which has made this particular subclass tractable, namely Kronecker's theory of pencils. This theory deals with developing canonical forms for an arbitrary pencil of matrices $G_1 + \lambda G_2$, λ a parameter, under the action of the standard equivalence group [9]. We now recall quickly some basic definitions and facts about Kronecker's canonical forms [9].

A pencil of matrices $G_1 + \lambda G_2$ is called *regular* if (i) both G_1 and G_2 are square matrices and (ii) the determinant of $G_1 + \lambda G_2$ does not vanish identically in λ; in all other cases, the pencil is called *singular.*

If $\{G_1, G_2\}$ form a *nondegenerate* singular pencil, then there exist two nonsingular matrices P and Q such that

$$P(G_1+\lambda G_2)Q=\begin{bmatrix} L_{\varepsilon_1} & & & & & \\ & \ddots & & & & \\ & & L_{\varepsilon_k} & & & \\ & & & L_{\eta_1}^T & & \\ & & & & \ddots & \\ & & & & & L_{\eta_r}^T \\ & & & & & & \tilde{G}_1+\lambda\tilde{G}_2 \end{bmatrix}$$

$$0<\varepsilon_1\leqq\varepsilon_2\leqq\cdots\leqq\varepsilon_k, \qquad 0<\eta_1\leqq\eta_2\leqq\cdots\leqq\eta_r$$

where

$$L_\varepsilon=\underbrace{\begin{bmatrix} \lambda & 1 & & \\ & \lambda & \ddots & \\ & & \ddots & 1 \\ & & \lambda & 1 \end{bmatrix}}_{\varepsilon+1}\Bigg\}\ \varepsilon \text{ and } \tilde{G}_1+\lambda\tilde{G}_2 \text{ is regular.}$$

The sequences $\{\varepsilon_i\}_{i=1}^k$ and $\{\eta_j\}_{j=1}^r$ are called the *minimal indices* of the pencil $G_1+\lambda G_2$.

Note that if $G_1+\lambda G_2$ is regular, then $k=r=0$. If $G_1+\lambda G_2$ is square, then $k=r$ and vice versa.

We start in § 2 by determining the complexity of a singular pair of bilinear forms; the remaining case of regular pairs is handled in § 3, where the dependence on the underlying algebraic structure is explicit and reveals unexpected results. Several interesting results concerning the special cases of fields with small cardinalities and of the integers are discussed in [17] and [18].

It is worth mentioning that there are two prior works concerning this subclass [6], [12], neither of which has ever mentioned Kronecker's theory of pencils or the related theory of invariant polynomials; however, it would be fair to admit that [12] has originally motivated the interest in this problem.

2. The complexity of a singular pair of bilinear forms. Let $B_1=x^TG_1y$ and $B_2=x^TG_2y$ be a pair of bilinear forms over a field F; the pair (B_1, B_2) will be called singular (regular) if the corresponding pencil $G_1+\lambda G_2$ is singular (regular). In this section, we will determine explicitly the complexity of a singular pair of bilinear forms.

It is easy to see from Kronecker's canonical form that the nondegenerate characteristic matrix $G(s)$ corresponding to the above pair of bilinear forms is equivalent to the following "canonical" characteristic matrix (because the complexity is invariant under the action of the "equivalence group" [4])

$$G(s)\approx\begin{bmatrix} L_{\varepsilon_1}(s) & & & & & \\ & \ddots & & & & \\ & & L_{\varepsilon_r}(s) & & & \\ & & & L_{\eta_1}^T(s) & & \\ & & & & \ddots & \\ & & & & & L_{\eta_k}^T(s) \\ & & & & & & \tilde{G}(s) \end{bmatrix}$$

where $0<\varepsilon_1\leqq\varepsilon_2\leqq\cdots\leqq\varepsilon_r$, $0<\eta_1\leqq\eta_2\leqq\cdots\leqq\eta_k$ minimal indices

$$L_\varepsilon(s)=\underbrace{\begin{bmatrix} s_2 & s_1 & & \\ & s_2 & s_1 & \\ & & \ddots & \ddots \\ & & & s_2 \quad s_1 \end{bmatrix}}_{\varepsilon+1}\Bigg\}\varepsilon$$

and $\tilde{G}(s)$ is regular.

These minimal indices together with the elementary divisors of $G_1+\lambda G_2$ completely characterize, to within an equivalence, the pencil $G_1+\lambda G_2$.

Now we will compute the degree of a block $L_\varepsilon(s)$:

$$L_\varepsilon(s)=s_2\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ & & \ddots & & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}+s_1\begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ & & & \ddots & 0 \\ 0 & 0 & & \cdots & 1 \end{bmatrix}=s_2[I_\varepsilon 0]+s_1 J.$$

Consider the following rank one matrix

$$D=\underbrace{\begin{bmatrix} 0 & 0 & & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \\ \alpha_0 & \alpha_1 & \cdots & \alpha_{\varepsilon-1} & 1 \end{bmatrix}}_{\varepsilon+1}\Bigg\}\varepsilon, \qquad \alpha_i\in\mathscr{F}, \quad i=0,1,\cdots,\varepsilon-1.$$

We now assume that Card $\mathscr{F}\geqq\varepsilon$. Note that

$$J-D=\begin{bmatrix} 0 & 1 & & & 0 \\ 0 & 0 & 1 & & \vdots \\ & & \ddots & \ddots & \vdots \\ & & & 1 & 0 \\ -\alpha_0 & -\alpha_1 & \cdots & -\alpha_{\varepsilon-1} & 0 \end{bmatrix}=[C \quad 0],$$

where C is an $\varepsilon\times\varepsilon$ companion matrix whose characteristic (and minimal) polynomial is $p(t)=\alpha_0+\alpha_1 t+\cdots+\alpha_{\varepsilon-1}t^{\varepsilon-1}+t^\varepsilon$. Since Card $\mathscr{F}\geqq\varepsilon$, we can pick the α_i's in $\mathscr{F}$ so that $p(t)$ has ε distinct roots in $\mathscr{F}$ in which case there exists $P\in Gl(\mathscr{F},\varepsilon)$ such that

$$PCP^{-1}=\begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_\varepsilon \end{bmatrix}, \qquad \lambda_i\in\mathscr{F} \text{ i.e.,} \qquad C=\sum_{i=1}^{\varepsilon}\lambda_i P^{-1}E_i P$$

where E_i is the $\varepsilon\times\varepsilon$ matrix with 1 on the position (i,i) and 0 elsewhere. Thus

$$J-D=\sum_{i=1}^{\varepsilon}\lambda_i[P^{-1}E_i P0]$$

and we now have the following decomposition

$$J=\sum_{i=1}^{\varepsilon}\lambda_i[P^{-1}E_i P0]+D,$$

$$[I_\varepsilon 0]=\sum_{i=1}^{\varepsilon}[P^{-1}E_i P0].$$

Therefore $\delta_{\mathcal{F}}\{L_\varepsilon(s)\} \leqq \varepsilon + 1$; but since column rank $\{L_\varepsilon(s)\} = \varepsilon + 1$, $\delta_{\mathcal{F}}\{L_\varepsilon(s)\} \geqq \varepsilon + 1$ and therefore

$$\delta_{\mathcal{F}}\{L_\varepsilon(s)\} = \varepsilon + 1 \quad \text{if Card } \mathcal{F} \geqq \varepsilon.$$

Another way to get the same result is to notice that $L_\varepsilon(s)$ is equivalent to the characteristic matrix corresponding to multiplying a linear polynomial by an $(\varepsilon - 1)$ degree polynomial. However, in this case, the optimal algorithms are "nonequivalent" to the above ones [17].

Now it is easy to see that if a characteristic matrix $G(s)$ is made up of k chains $L_{\varepsilon_i}(s)$, $i = 1, 2, \cdots, k$ on the main diagonal and zero elsewhere, then $\delta_{\mathcal{F}}\{G(s)\} = \sum_{i=1}^{k} \varepsilon_i + k$ if Card $\mathcal{F} \geqq \max_i \{\varepsilon_i\}$. The same arguments apply obviously to $L_\eta^T(s)$. We now investigate the degree of the following combination

$$G(s) = \begin{bmatrix} L_\varepsilon(s) & \\ & L_\eta^T(s) \end{bmatrix} = \begin{bmatrix} \overbrace{\begin{matrix} s_2 & & s_1 \\ & \ddots & & \ddots \\ & s_2 & & s_1 \end{matrix}}^{\varepsilon+1} & \overbrace{\begin{matrix} 0 \\ \\ \end{matrix}}^{\eta} \\ \begin{matrix} 0 \\ \\ \end{matrix} & \overbrace{\begin{matrix} s_2 & \\ s_1 & \ddots \\ & \ddots & s_2 \\ & & s_1 \end{matrix}}^{\eta+1} \end{bmatrix}$$

whose size is $(\eta + \varepsilon + 1) \times (\eta + \varepsilon + 1)$, say $m \times m$. We will prove that

$$\delta_{\mathcal{F}}\{G(s)\} = m + 1 = \eta + \varepsilon + 2 \quad \text{if Card } \mathcal{F} \geqq \max\{\varepsilon, \eta\}.$$

From the previous results, it is clear that

$$\delta_{\mathcal{F}}\{G(s)\} \leqq \delta_{\mathcal{F}}\{L_\varepsilon(s)\} + \delta_{\mathcal{F}}\{L_\eta^T(s)\} \leqq \varepsilon + 1 + \eta + 1 = m + 1$$

and because of nondegeneracy, $\delta_{\mathcal{F}}\{G(s)\} \geqq m$. However, $G(s)$ cannot be of degree m since otherwise there exists a triplet (A, B, C) such that $G(s) = CA(s)B$, where C, $A(s)$ and B are $m \times m$ matrices and hence $\det G(s) = (\det C)(\det B) \prod_{i=1}^{m} \langle a_i, s \rangle$ with $\det C \neq 0$, $\det B \neq 0$ and $\det A(s) \neq 0$ [4]; but in our case $\det G(s) \equiv 0$ and therefore $\delta_{\mathcal{F}}\{G(s)\} = m + 1$.

We collect all the previous facts in the following theorem.

THEOREM 2.1. *Let $\mathcal{F}$ be any field with large enough cardinality. Then we have the following*

$$\delta_{\mathcal{F}}\{L_\varepsilon(s)\} = \varepsilon + 1, \ \delta_{\mathcal{F}} \begin{bmatrix} L_{\varepsilon_1}(s) & & \\ & \ddots & \\ & & L_{\varepsilon_k}(s) \end{bmatrix} = \sum_{i=1}^{k} \varepsilon_i + k$$

and

$$\delta_{\mathcal{F}} \begin{bmatrix} L_\varepsilon(s) & 0 \\ 0 & L_\eta^T(s) \end{bmatrix} = \varepsilon + \eta + 2.$$

Remark. From now on, "large enough cardinality" means Card $\mathcal{F} \geqq \max\{\varepsilon_i, \eta_j\}$ whenever we are considering singular pencils.

COROLLARY 2.1.1. *Over any field* $\mathcal{F}$ *with* Card $\mathcal{F} \geqq \varepsilon$, *the minimal number of multiplications needed to compute the pair of bilinear forms*

$$B_1 = x_1y_1 + x_2y_2 + \cdots + x_\varepsilon y_\varepsilon,$$

$$B_2 = x_1y_2 + x_2y_3 + \cdots + x_\varepsilon y_{\varepsilon+1}$$

is precisely $\varepsilon + 1$ *compared to* 2ε *multiplications needed by the ordinary algorithm.*

Similar statements can be made which correspond to the other parts of Theorem 2.1. An example of the actual algorithms will be presented now. Consider the computation of the pair

$$B_1 = x_1y_1 + \cdots + x_ny_n,$$

$$B_2 = x_1y_2 + x_2y_3 + \cdots + x_ny_{n+1}$$

over a field $\mathcal{F}$ with Card $\mathcal{F} \geqq n$. The corresponding characteristic matrix is given by $L_n(s) = s_1[I_n 0] + s_2 J$. Recall that we used a rank one matrix of the form

$$D = \begin{bmatrix} 0 & 0 & & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 \\ \alpha_0 & \alpha_1 & & \alpha_{n-1} & 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} [\alpha_0 \quad \alpha_1 \quad \cdots \quad \alpha_{n-1} \quad 1]$$

so as to make the polynomial $p(t) = t^n + \alpha_{n-1}t^{n-1} + \cdots + \alpha_1 t + \alpha_0$ have n distinct roots, say $\lambda_1, \lambda_2, \cdots, \lambda_n$.

Let $J - D = [H0]$, then it is easy to check that

$$V^{-1}HV = \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix},$$

where V is a Vandermonde matrix given by

$$V = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \lambda_1 & \lambda_2 & \cdots & \lambda_n \\ \lambda_1^2 & \lambda_2^2 & \cdots & \lambda_n^2 \\ \lambda_1^{n-1} & \lambda_2^{n-1} & \cdots & \lambda_n^{n-1} \end{bmatrix}.$$

Note that $(-1)^i\alpha_i$'s are the elementary symmetric functions of the λ_i's. The corresponding optimal algorithm is the following:

Decompose y as $\begin{pmatrix} \bar{y} \\ y_{m+1} \end{pmatrix}$.

1. Compute the Vandermonde Transform of x i.e. $Z_1 = V^T x$.
2. Compute $Z_2 = V^{-1}\bar{y}$.
3. Compute each term of the inner product

$$\langle Z_1, Z_2 \rangle, \text{ say } \langle Z_1, Z_2 \rangle = \sum_{i=1}^{n} \xi_i, \text{ and } \xi_{n+1} = x_n \cdot \left(\sum_{i=1}^{n} \alpha_{i-1}y_i + y_{n+1} \right).$$

4.

$$B_1 = \sum_{i=1}^{n} \xi_i, \quad B_2 = \sum_{i=1}^{n} \lambda_i \xi_i + \xi_{n+1}.$$

The only step which involves nonscalar multiplications is Step 3 and therefore the above bilinear algorithm uses $n+1$ nonscalar multiplications. Note that the choice of the λ_i's is arbitrary as long as they are distinct. Actually, two distinct sequences of λ_i's give rise to two "nonequivalent" optimal algorithms [17].

Recall that bilinear algorithms are noncommutative and hence the above algorithm works if the indeterminates $\{x_i\}$ and $\{y_i\}$ represent arbitrary matrices.

Before proceeding we will need the following result true for any set of bilinear forms.

THEOREM 2.2. *Let* $\{G_i(s) \mid i = 1, 2, \cdots, k\}$ *be a set of characteristic matrices over any commutative ring* $\mathcal{A}$, *not necessarily of the same dimensions, with* $\dim s = m$ *and let* $\pi \in S_k$. *Then*

$$\delta_{\mathcal{A}}\begin{bmatrix} G_1(s) & & & \\ & G_2(s) & & \\ & & \ddots & \\ & & & G_k(s) \end{bmatrix} = \delta_{\mathcal{A}}\begin{bmatrix} G_{\pi(1)}(s) & & & \\ & G_{\pi(2)}(s) & & \\ & & \ddots & \\ & & & G_{\pi(k)}(s) \end{bmatrix}.$$

Proof. The proof is straightforward.

Using this theorem we can prove the following interesting lemma.

LEMMA 2.3. *Let* $G(s)$ *be any characteristic matrix over a field* $\mathcal{F}$ *with* Card $\mathcal{F} \geqq \max\{\varepsilon, \eta\}$. *Then*

$$\delta_{\mathcal{F}}\begin{bmatrix} G(s) & 0 \\ 0 & L_\eta^T(s) \end{bmatrix} = \delta_{\mathcal{F}}\{G(s)\} + \delta_{\mathcal{F}}\{L_\eta^T(s)\} = \delta_{\mathcal{F}}\{G(s)\} + \eta + 1$$

and

$$\delta_{\mathcal{F}}\begin{bmatrix} L_\varepsilon(s) & 0 \\ 0 & G(s) \end{bmatrix} = \delta_{\mathcal{F}}\{L_\varepsilon(s)\} + \delta_{\mathcal{F}}\{G(s)\} = \delta_{\mathcal{F}}\{G(s)\} + \varepsilon + 1.$$

Proof. From the previous theorem we know that

$$\delta_{\mathcal{F}}\begin{bmatrix} G(s) & 0 \\ 0 & L_\eta^T(s) \end{bmatrix} = \delta_{\mathcal{F}}\begin{bmatrix} L_\eta^T(s) & 0 \\ 0 & G(s) \end{bmatrix}.$$

Now applying Theorem 10 of Brockett and Dobkin [4, p. 26] we get

$$\delta_{\mathcal{F}}\begin{bmatrix} L_\eta^T(s) & 0 \\ 0 & G(s) \end{bmatrix} \geqq \delta_{\mathcal{F}}\{G(s)\} + \text{row rank } \{L_\eta^T(s)\} = \delta_{\mathcal{F}}\{G(s)\} + \delta_{\mathcal{F}}\{L_\eta^T(s)\}$$

and therefore

$$\delta_{\mathcal{F}}\begin{bmatrix} L_\eta^T(s) & 0 \\ 0 & G(s) \end{bmatrix} = \delta_{\mathcal{F}}\{L_\eta^T(s)\} + \delta_{\mathcal{F}}\{G(s)\}.$$

The same argument holds for the other case.

We now state the main theorem of this section.

THEOREM 2.4. *Let $\mathcal{F}$ by any field with* Card $\mathcal{F} \geqq \max_{i,j}\{\varepsilon_i, \eta_j\}$. *Then*

$$\delta_{\mathcal{F}}\begin{bmatrix} L_{\varepsilon_1}(s) & & & & & \\ & \ddots & & & & \\ & & L_{\varepsilon_r}(s) & & & \\ & & & L^T_{\eta_1}(s) & & \\ & & & & \ddots & \\ & & & & & L^T_{\eta_k}(s) \\ & & & & & & \tilde{G}(s) \end{bmatrix}$$

$$= \sum_{i=1}^{r} \delta_{\mathcal{F}}\{L_{\varepsilon_i}(s)\} + \sum_{j=1}^{k} \delta_{\mathcal{F}}\{L^T_{\eta_j}(s)\} + \delta_{\mathcal{F}}\{\tilde{G}(s)\}$$

$$= \sum_{i=1}^{r} \varepsilon_i + r + \sum_{j=1}^{k} \eta_j + k + \delta_{\mathcal{F}}\{\tilde{G}(s)\}.$$

Proof. Note that

$$\delta_{\mathcal{F}}\begin{bmatrix} L_{\varepsilon_1}(s) & & & & & \\ & \ddots & & & & \\ & & L_{\varepsilon_r}(s) & & & \\ & & & L^T_{\eta_1}(s) & & \\ & & & & \ddots & \\ & & & & & L^T_{\eta_k}(s) \\ & & & & & & \tilde{G}(s) \end{bmatrix} = \delta_{\mathcal{F}}\{L_{\varepsilon_1}(s)\} + \delta_{\mathcal{F}}\begin{bmatrix} L_{\varepsilon_2}(s) & & & & & \\ & \ddots & & & & \\ & & L_{\varepsilon_r}(s) & & & \\ & & & L^T_{\eta_1}(s) & & \\ & & & & \ddots & \\ & & & & & L^T_{\eta_k}(s) \\ & & & & & & \tilde{G}(s) \end{bmatrix}.$$

Proceeding inductively we get the results.

The regular pencil $\tilde{G}_1 + \lambda\tilde{G}_2$ obtained from the original pencil $G_1 + \lambda G_2$ is called the *regular kernel* of $G_1 + \lambda G_2$.

COROLLARY 2.4.1. *Suppose that the regular kernel of a $p \times q$ pencil $G_1 + \lambda G_2$ is of size $n_1 \times n_1$ and suppose that r (respectively k) is the number of minimal indices for the columns (resp. rows). Then*

$$\delta_{\mathcal{F}}\{G(s)\} = q - n_1 + k + \delta_{\mathcal{F}}\{\tilde{G}(s)\} = p - n_1 + r + \delta_{\mathcal{F}}\{\tilde{G}(s)\}.$$

Recall that we are always considering nondegenerate characteristic matrices which implies here that ($r = 0$ or $\varepsilon_1 > 0$) and ($k = 0$ or $\eta_1 > 0$).

Let us define rank $(G_1 + \lambda G_2)$ to be the dimension of the largest minor of $G_1 + \lambda G_2$ which is not identically zero in λ. We now have an interesting corollary.

COROLLARY 2.4.2. *Let the regular kernel $\tilde{G}_1 + \lambda\tilde{G}_2$ of a $p \times q$ pencil $G_1 + \lambda G_2$ be of size $n_1 \times n_1$. Then*

$$\delta_{\mathcal{F}}\{G(s)\} = p + q - \{n_1 + \text{rank}\,(G_1 + \lambda G_2)\} + \delta_{\mathcal{F}}\{\tilde{G}(s)\}.$$

Proof Note that

$$\sum_{i=1}^{r} \varepsilon_i + \sum_{l=1}^{k} \eta_j + n_1 = \text{rank}\,(G_1 + \lambda G_2) = p - k = q - r.$$

Thus $k = p - \text{rank}\,(G_1 + \lambda G_2)$, $r = q - \text{rank}\,(G_1 + \lambda G_2)$. Therefore

$$\delta_{\mathcal{F}}\{G(s)\} = q - n_1 + k + \delta_{\mathcal{F}}\{\tilde{G}(s)\} = q - n_1 + \{p - \text{rank}\,(G_1 + \lambda G_2)\} + \delta_{\mathcal{F}}\{\tilde{G}(s)\},$$

$$\delta_{\mathcal{F}}\{G(s)\} = p + q - \{n_1 + \text{rank}\,(G_1 + \lambda G_2)\} + \delta_{\mathcal{F}}\{\tilde{G}(s)\}.$$

Let us note here that the dimension n_1 of the regular kernel $\tilde{G}(s)$ can be determined directly from the elementary divisors or the invariant polynomials of the original pencil $G_1 + \lambda G_2$ [9]. However, we will later see that the degree of $\tilde{G}(s)$ will be expressed as $n_1 + \mu$, where μ will be defined later, so that the complexity of a general pair of bilinear forms is free from n_1.

COROLLARY 2.4.3. *Over any field $\mathcal{F}$ with large enough cardinality, the optimal number of multiplications needed to compute $B_1 = x^T G_1 y$ and $B_2 = x^T G_2 y$ is equal to*

$$\sum_{i=1}^{r} \varepsilon_i + r + \sum_{j=1}^{k} \eta_j + k + \theta,$$

where the ε_i's and the η_j's are the minimal indices of the pencil $G_1 + \lambda G_2$ and where θ is the optimal number of multiplications needed to compute the regular kernel of the given pair of bilinear forms.

As an example, consider the computation of

$$B_1 = x_1 y_1 + x_2 y_2 + x_3 y_4 + x_4 y_5,$$

$$B_2 = x_1 y_2 + x_2 y_3 + x_4 y_4 + x_5 y_5.$$

The corresponding characteristic matrix is given by

$$G(s) = \begin{bmatrix} s_1 & s_2 & 0 & 0 & 0 \\ 0 & s_1 & s_2 & 0 & 0 \\ 0 & 0 & 0 & s_1 & 0 \\ 0 & 0 & 0 & s_2 & s_1 \\ 0 & 0 & 0 & 0 & s_2 \end{bmatrix}$$

whose degree is 6 over any field $\mathcal{F}$ with Card $\mathcal{F} \geqq 2$.

Therefore 6 multiplications are necessary and sufficient to evaluate B_1 and B_2 over any nontrivial field compared to 8 multiplications needed by the ordinary algorithm.

It follows from the above that the real problem comes down to evaluating the complexity of a regular pair of bilinear forms, a topic which will be discussed in the next section.

3. The complexity of a regular pair of bilinear forms. Let us recall that a pair of bilinear forms $B_1 = x^T G_1 y$ and $B_2 = x^T G_2 y$ is regular if both G_1 and G_2 are square matrices such that $\det(G_1 + \lambda G_2)$ does not vanish identically in λ. In this case, if Card $\mathcal{F} \geqq n$, we can pick a 2×2 nonsingular matrix T such that $G(Ts)$ has one nonsingular matrix; so, without loss of generality (because of the equivalence group action [4]), we can restrict ourselves to the pencil $I_n + \lambda G$, where G is any $n \times n$ matrix over a field $\mathcal{F}$.

LEMMA 3.1. *Given any $n \times n$ matrix G, then $\delta_{\mathcal{F}}\{I_n, G\} = n$ if and only if, G is similar to a diagonal matrix*, i.e., *G has n distinct eigenvectors.*

Proof. Sufficiency is obvious.

Note that any set of n rank one matrices which include I_n in their span has to be of the form $\{PE_iP^{-1} \mid P \in Gl(\mathcal{F}, n)\}$, E_i is the $n \times n$ matrix with 1 on the (i, i) position and zero elsewhere. Now, if $\delta_{\mathcal{F}}\{I_n, G\} = n$, then $G = \sum_{i=1}^{n} \lambda_i PE_iP^{-1}$ for some $\{\lambda_i\}_{i=1}^{n}$ and $P \in Gl(\mathcal{F}, n)$ from which the result follows.

Let us recall from linear algebra that any $n \times n$ matrix B, over a field $\mathcal{F}$, is similar to one and only one matrix which is in rational canonical form (also called, first normal form), i.e., there exists $P \in Gl(\mathcal{F}, n)$ such that

$$\tilde{B} = PBP^{-1} = \begin{bmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_r \end{bmatrix},$$

A_i is a $k_i \times k_i$ companion matrix corresponding to the invariant polynomial $p_i(t)$ of degree k_i and $p_{i+1}(t) \mid p_i(t)$, $i = 1, 2, \cdots, r-1$. We can get the invariant polynomials of B as follows:

$$p_k(t) = \frac{d_{n-k+1}(B)}{d_{n-k}(B)}, \qquad 1 \leqq k \leqq r, \quad p_{r+1}(t) = \cdots = p_n(t) = 1,$$

where $d_k(B)$ is the greatest common divisor of all minors of $I_n t - B$ of order $k[d_0(B) = 1]$.

To compute the degree of the pair $\{I_n, B\}$, we will consider the equivalent pair $\{I_n, \tilde{B}\}$, where $\tilde{B}$ is the rational canonical form of B.

We first handle the special case where $r = 1$, i.e., $\tilde{B}$ is a companion matrix, say

$$\tilde{B} = C = \begin{bmatrix} 0 & 1 & 0 & & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & 1 \\ \alpha_0 & \alpha_1 & & & \cdots & \alpha_{n-1} \end{bmatrix}, \qquad \alpha_i \in \mathcal{F} \text{ with Card } \mathcal{F} \geqq n.$$

As before, consider the $n \times n$ rank one matrix

$$D = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 0 \\ \beta_0 & \beta_1 & \cdots & \beta_{n-1} \end{bmatrix}, \qquad \beta_i \in \mathcal{F}, \quad i = 0, 1, \cdots, n-1,$$

the β_i's will be specified later. Then

$$D - C = \begin{bmatrix} 0 & 1 & \cdots & 0 \\ \vdots & 0 & & \\ \vdots & \vdots & & \\ 0 & 0 & \cdots & 1 \\ \beta_0 - \alpha_0 & \beta_1 - \alpha_1 & & \beta_{n-1} - \alpha_{n-1} \end{bmatrix}$$

whose characteristic (and minimal) polynomial is

$$p(t) = (\alpha_0 - \beta_0) + (\alpha_1 - \beta_1)t + \cdots + (\alpha_{n-1} - \beta_{n-1})t^{n-1} + t^n.$$

It is easy to see that we can pick the β_i's so that $p(t)$ has n distinct roots in $\mathcal{F}$ and thus so that $D - C$ is diagonalizable, i.e., there exists $P \in Gl(\mathcal{F}, n)$ such that

$$P^{-1}(C - D)P = \sum_{i=1}^{n} \lambda_i E_i,$$

$$C = \sum_{i=1}^{n} \lambda_i P E_i P^{-1} + D.$$

Moreover, note that $I_n = \sum_{i=1}^{n} PE_iP^{-1}$ and therefore $\delta_{\mathcal{F}}\{I_n, C\} \leqq n+1$ for all $\alpha_i \in \mathcal{F}$, $i = 0, 1, \cdots, n-1$ and Card $\mathcal{F} \geqq n$. Now from Lemma 3.1, $\delta_{\mathcal{F}}\{I_n, C\} = n$ if, and only if, C is diagonalizable, i.e., if and only if, the polynomial $q(t) = -\alpha_0 - \alpha_1 t - \cdots - \alpha_{n-1}t^{n-1} + t^n$ has *n distinct* roots in $\mathcal{F}$ because the minimal polynomial of C is the same as the characteristic polynomial $q(t)$. Therefore we have the following theorem.

THEOREM 3.2. *Let $\mathcal{F}$ be any field with* Card $\mathcal{F} \geqq n$ *and let C be an $n \times n$ companion matrix. Then $\delta_{\mathcal{F}}\{I_n, C\} = n$ if and only if, the characteristic polynomial of C factors into n distinct linear factors over $\mathcal{F}$; otherwise, $\delta_{\mathcal{F}}\{I_n, C\} = n+1$.*

COROLLARY 3.2.1. *The optimal number of multiplication needed to compute*

$$B_1 = x_1y_1 + x_2y_2 + \cdots + x_ny_n,$$

$$B_2 = x_1y_2 + x_2y_3 + \cdots + x_{n-1}y_n + \alpha_0x_ny_1 + \cdots + \alpha_{n-1}x_ny_n,$$

$$\alpha_i \in \mathcal{F}, \quad i = 0, 2, \cdots, n-1,$$

is less than or equal to $n+1$; it is equal to n if and only if $q(t) = -\alpha_0 - \alpha_1 t - \cdots - \alpha_{n-1}t^{n-1} + t^n$ factors into n distinct linear factors over $\mathcal{F}$.

As an application, we consider the multiplication of an $n \times n$ matrix C of the following form

$$C = \begin{bmatrix} x_1 & x_2 & & & \\ & x_1 & \ddots & x_2 & \\ & & \ddots & & \ddots \\ & & & x_1 & & x_2 \\ & & & & x_1 & \end{bmatrix}$$

by an arbitrary vector y:

$$Cy = \begin{bmatrix} x_1y_1 & + & x_2y_2 \\ x_1y_2 & + & x_2y_3 \\ & \vdots & \\ x_1y_{n-1} & + & x_2y_n \\ & x_1y_n & \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_{n-1} \\ B_n \end{bmatrix}$$

This is a dual of the problem of the above corollary, where all of the α_i's are zero so its complexity is $n+1$ compared to $2n-1$ mult. used by the ordinary algorithm. The actual optimal algorithm goes as follows:

Let $\lambda_1, \lambda_2, \cdots, \lambda_n$ be n distinct elements of $\mathcal{F}$ and let $(-1)^i\alpha_i$ be the ith elementary symmetric function of the λ_i's; V will denote the following Vandermonde matrix

$$V = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \lambda_1 & \lambda_2 & & \lambda_n \\ \vdots & \vdots & & \vdots \\ \lambda_1^{n-1} & \lambda_2^{n-1} & & \lambda_n^{n-1} \end{bmatrix}.$$

1. Compute $\xi_i = x_1 + \lambda_i x_2$, $i = 1, 2, \cdots, n$.
2. Compute $\eta = V^{-1}y$ and $\eta_{n+1} = \sum_{i=1}^{n} \alpha_i y_i$.
3. Compute the nonscalar multiplications,

$$f_i = \xi_i \times \eta_i, \qquad i = 1, 2, \cdots, n$$

$$f_{n+1} = -x_2 \times \eta_{n+1}.$$

4. Then $B_i = \sum_{j=1}^{n} \lambda_j^{i-1} f_j + \delta_{in} f_{n+1}$, where $\delta_{in} = \begin{cases} 0 & \text{if } i \neq n, \\ 1 & \text{if } i = n. \end{cases}$

The above algorithm works for any n distinct λ_i's in $\mathcal{F}$.

We can pick the λ_i's, depending on the field $\mathcal{F}$ and on the particular machine, so as to make the scalar multiplications "easy".

We now state the main theorem of this section.

THEOREM 3.3. *Let G be any $n \times n$ matrix over a field $\mathcal{F}$ whose invariant polynomials are $p_i(t)$, $i = 1, 2, \cdots, r$, $p_{r+1}(t) = \cdots = p_n(t) = 1$ with* Card $\mathcal{F} \geqq \max \{\deg p_i(t)\}$. *Let k be the number of those $p_i(t)$'s, $i = 1, 2, \cdots, r$, which cannot be factored into distinct linear factors over $\mathcal{F}$. Then $\delta_{\mathcal{F}}\{I_n, G\} \leqq n + k$.*

Note that if $p_k(t)$ factors into distinct linear factors over $\mathcal{F}$, then so do all $p_l(t)$, $l \geqq k$.

Let's now consider some examples.

1. Consider the computation of the following set of pairs of bilinear forms

$$a_i c_i + b_i d_i, \qquad \alpha\beta b_i c_i + (\alpha + \beta) b_i d_i + a_i d_i, \qquad i = 1, 2, \cdots, k,$$

where $\{a_i, b_i, c_i, d_i\}_{i=1}^{k}$ are indeterminates and $\alpha \neq \beta$ are constants in $\mathcal{F}$.

The corresponding characteristic matrix is given by

$$G(s) = s_1 I_{2k} + s_2 (I_k \otimes L) \quad \text{where } L = \begin{bmatrix} 0 & 1 \\ -\alpha\beta & \alpha + \beta \end{bmatrix}.$$

It is clear that $\delta_{\mathcal{F}}\{G(s)\} = 2k$ since L is diagonalizable and therefore $2k$ multiplications are necessary and sufficient to compute the above bilinear forms over any nontrivial field $\mathcal{F}$.

2. Consider the computation of

$$B_1 = x_1 y_1 + x_2 y_2 + x_3 y_3,$$
$$B_2 = x_1 y_2 + x_2 y_3 + x_3 y_1$$

over $\mathcal{F} = Z_5$, Z_7 and Z_{11}. The corresponding characteristic matrix is given by

$$G(s) = \begin{bmatrix} s_1 & s_2 & 0 \\ 0 & s_1 & s_2 \\ s_2 & 0 & s_1 \end{bmatrix} = s_1 I_3 + s_2 \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

This pencil has one invariant polynomial, namely

$$p(t) = t^3 - 1.$$

Over Z_5, we have $p(t) = (t-1)(t^2 + t + 1)$, where $t^2 + t + 1$ is irreducible over Z_5; hence

$$\delta_{Z_5}\{G(s)\} = 3 + 1 = 4.$$

Similarly, it is easy to see that $\delta_{Z_{11}}\{G(s)\} = 4$ because $p(t)$ does not split into a product of distinct linear factors over Z_{11}. However, over Z_7 we have

$$p(t) = (t-1)(t-2)(t-4)$$

and thus $p(t)$ factors into distinct linear factors over Z_7, i.e.,

$$\delta_{\mathcal{F}}\{G(s)\} = 3.$$

It follows that the optimal computation of B_1 and B_2 takes 4 multiplications over Z_5 or Z_{11} and 3 multiplications over Z_7.

We will now state several corollaries to Theorem 3.3. Before, we note that a field with "large enough cardinality" is understood here to mean Card $\mathcal{F} \geqq \max_{i,j,k} \{\deg p_i(t), \varepsilon_j, \eta_k\}$, where $\{\varepsilon_j, \eta_k\}$ are the minimal indices and $p_i(t)$'s are the corresponding invariant polynomials.

COROLLARY 3.3.1. *Let $\mathcal{F}$ be a field with large enough cardinality and let $B_1 = x^T G_1 y$ and $B_2 = x^T G_2 y$ be two $p \times q$ bilinear forms. Then*

$$\delta_{\mathcal{F}}\{B_1, B_2\} \leqq p + q + k - \operatorname{rank}(G_1 + \lambda G_2),$$

where k is as defined in Theorem 3.3 and which corresponds to the regular Kernel $\tilde{G}_1 + \lambda \tilde{G}_2$.

Proof. From Corollary 2.4.2 we have

$$\delta_{\mathcal{F}}\{G(s)\} = p + q - \{n_1 + \operatorname{rank}(G_1 + \lambda G_2)\} + \delta_{\mathcal{F}}\{G(s)\}$$

and from Theorem 3.3, $\delta_{\mathcal{F}}\{\tilde{G}(s)\} \leq n_1 + k$ and therefore

$$\delta_{\mathcal{F}}\{G(s)\} \leqq p + q + k - \operatorname{rank}(G_1 + \lambda G_2).$$

Let us now hasten to make the following remark: To compute k, we do not have to transform $G_1 + \lambda G_2$ into Kronecker's canonical form and then put the regular kernel in the form $\{I, H\}$; we only have to compute the invariant polynomials of the *original* pencil $\mu G_1 + \lambda G_2$ and determine the number of those which do not factor into distinct linear factors over $\mathcal{F}$. This fact together with several similar facts are established in [17].

COROLLARY 3.3.2.[1] *Let $\mathcal{F}$ be an algebraically closed field, then*

$$\delta_{\mathcal{F}}\{I_n, G\} \leqq n + \max\{\textit{number of nontrivial Jordan chains associated with a given eigenvalue of } G\}.$$

Proof. Before giving the proof, let us recall some basic facts about Jordan canonical forms.

Suppose that $p_1(t), p_2(t), \cdots, p_m(t)$ are the invariant polynomials of G as defined above. Factor these polynomials into irreducible factors, we have

$$\begin{aligned} p_1(t) &= (t-\alpha_1)^{\tau_{11}}(t-\alpha_2)^{\tau_{12}} \cdots (t-\alpha_l)^{\tau_{1l}} \\ p_2(t) &= (t-\alpha_1)^{\tau_{21}}(t-\alpha_2)^{\tau_{22}} \cdots (t-\alpha_l)^{\tau_{2l}} \\ &\vdots \\ p_m(t) &= (t-\alpha_1)^{\tau_{m1}}(t-\alpha_2)^{\tau_{m2}} \cdots (t-\alpha_l)^{\tau_{ml}}, \end{aligned}$$

where

$$\begin{aligned} &\alpha_i \in \mathcal{F}, \qquad i = 1, 2, \cdots, l, \\ &\tau_{1i} > 0, \qquad i = 1, 2, \cdots, l, \end{aligned}$$

and

$$\tau_{rs} \leqq \tau_{ts} \qquad \text{for } r \leqq t.$$

Recall that the α_i's are the eigenvalues and that the nontrivial factors $(t-\alpha_j)^{\tau_{ij}}$ constitute the elementary divisors of G. With each elementary divisor $(t-\alpha_j)^{\tau_{ij}}$ is

[1] This was suggested by Professor R. W. Brockett.

associated a Jordan block of the following form

$$\begin{bmatrix} \alpha_j & 1 & & \\ & \alpha_j & 1 & \\ & & \ddots & \ddots \\ & & & \alpha_j \end{bmatrix} \quad \text{of size } \tau_{lj} \times \tau_{lj}.$$

Since $\delta_{\mathscr{F}}\{I_n, G\} \leqq n + k$, where k is the number of $p_i(t)$'s which do not factor into distinct linear factors, it is easy that this is the same as the largest ν such that

$$\tau_{\nu i} > 1 \quad \text{for some } i.$$

The result follows easily from this observation.

COROLLARY 3.3.3. *Let $G(s)$ be a 2×2 nondegenerate characteristic matrix with* dim $s = 2$. *Then*

$$\delta_{\mathscr{F}}\{G(s)\} = 2 + \mu,$$

where

$$\mu = \begin{cases} 0 & \text{if } \det G(s) \text{ factors into two distinct linear factors over } \mathscr{F}, \\ 1 & \text{otherwise.} \end{cases}$$

Proof. Note, from Kronecker's canonical form, that if $G(s)$ is nondegenerate, then it is also regular in this case. Moreover, the only nontrivial invariant polynomial of $G(s)$ is det $G(s)$. The result follows if we apply Theorem 3.3.

As an application to this corollary, consider the multiplication of two complex numbers $(x_1 + x_2 i)$ and $(y_1 + y_2 i)$. The corresponding characteristic matrix is given by

$$G(s) = \begin{bmatrix} s_1 & s_2 \\ s_2 & -s_1 \end{bmatrix}.$$

Since $\det G(s) = -s_1^2 - s_2^2$ does not factor into two distinct linear factors over $\mathscr{R}$ or $\mathscr{Q}$, we have

$$\delta_{\mathscr{R}}\{G(s)\} = \delta_{\mathscr{Q}}\{G(s)\} = 3$$

and the optimal number of multiplications is 3 (see also [4]).

We now have a lemma which is really a corollary of Theorems 2.4 and 3.3.

LEMMA 3.4. *Let B_1 and B_2 be two $p \times q$ bilinear forms over a field $\mathscr{F}$ with* Card $\mathscr{F}$ *large enough. Then the complexity of B_1 and B_2 satisfies*

$$\delta \leqq \min\left(p + \frac{q}{2}, q + \frac{p}{2}\right).$$

Proof. By corollary 2.4.1, δ satisfies

$$\delta = p - n_1 + r + \delta_{\mathscr{F}}\{\tilde{G}(s)\},$$

where $\tilde{G}(s)$ is the regular kernel, n_1 is its size and r is the number of minimal indices for the rows.

We know, from Theorem 3.3 that

$$\delta_{\mathscr{F}}\{\tilde{G}(s)\} \leqq n_1 + \frac{n_1}{2} = \frac{3n_1}{2}$$

and it is obvious that $r \leqq (q - n_1)/2$. Therefore

$$\delta \leqq p - n_1 + \frac{q - n_1}{2} + \frac{3n_1}{2} = p + \frac{q}{2}.$$

Similarly, we have $\delta \leqq q + p/2$ and the lemma follows.

COROLLARY 3.4.1. *If B_1 and B_2 are any two $n \times n$ bilinear forms, then*

$$\delta\{B_1, B_2\} \leqq \left\lfloor \frac{3n}{2} \right\rfloor$$

i.e., *no pair of $n \times n$ bilinear forms requires more than $\lfloor 3n/2 \rfloor$ multiplications.*

COROLLARY 3.4.2. *If $\{B_i\}_{i=1}^m$ is a set of $p \times q$ bilinear forms over a field $\mathcal{F}$ with large enough cardinality, then*

$$\delta_{\mathcal{F}}\{B_i\}_{i=1}^m \leqq \left\lceil \frac{m}{2} \right\rceil \left(\min\left(p + \frac{q}{2}, q + \frac{p}{2} \right) \right).$$

For $p = q$, the above inequality implies

$$\delta_{\mathcal{F}}\{B_i\}_{i=1}^m \leqq \frac{3mn}{4}.$$

Remarks. (a) Results similar to Lemma 3.4 and Corollary 3.4.1 have been established in [2] and [6] in the case where the field $\mathcal{F}$ is algebraically closed. Our results are true over any field and, in particular, over finite fields (see also [18]).

(b) Bounds similar to those of Corollary 3.4.2 have been found by Howell [15] in the case where the constant set is a principal ideal domain.

One may still ask whether the upper bound of Theorem 3.3 is tight or it can be improved. We first prove that, in general, this bound cannot be improved and we later show that it is also a lower bound in the case where the first invariant polynomial $p_1(t)$ splits over $\mathcal{F}$.

Consider the $m \times m$ J_m-matrix defined as follows:

$$J_m = \begin{bmatrix} 0 & 1 & 0 & & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ & & & \ddots & & \vdots \\ & & & & \ddots & 0 \\ & & & & & 1 \\ 0 & \cdot & & \cdot & \cdot & 0 \end{bmatrix}.$$

We are ready for the next theorem.

THEOREM 3.5. *For each k in Theorem 3.3, we can display a regular pair of bilinear forms such that $\delta_{\mathcal{F}}\{I_n, G\} = n + k$, where $\mathcal{F}$ is any field.*

Proof. We first exhibit the pair with the highest k, i.e., $k = \lfloor n/2 \rfloor$,

$$B_1 = \sum_{i=1}^{n} x_i y_i, \qquad B_2 = \sum_{i=0}^{\lfloor n/2 \rfloor - 1} x_{2i+1} y_{2i+2}.$$

The corresponding characteristic matrix is generated by I_n and A, where A is the $n \times n$ matrix with $\lfloor n/2 \rfloor$ invariant polynomials all of which are equal to $p(t) = t^2$; thus, $k = \lfloor n/2 \rfloor$ and $\delta_{\mathcal{F}}\{I_n, A\} \leqq \lfloor 3n/2 \rfloor$. We now prove that it is also a lower bound. For the sake of clarity, let us assume that n is even. Then we have to compute the degree of the

following characteristic matrix:

$$G(s)=\begin{bmatrix} s_1 & s_2 & & & & & \\ 0 & s_1 & & & & & \\ & & s_1 & s_2 & & & \\ & & 0 & s_1 & & & \\ & & & & \ddots & & \\ & & & & & s_1 & s_2 \\ & & & & & 0 & s_1 \end{bmatrix}, \qquad \frac{n}{2}\text{ blocks.}$$

Rearranging rows, we obtain

$$\tilde{G}(s)=\begin{bmatrix} s_1 & s_2 & & & & & \\ & & s_1 & s_2 & & & \\ & & & & \ddots & & \\ & & & & & s_1 & s_2 \\ 0 & s_1 & 0 & 0\cdots 0 & & 0 & \\ 0 & 0 & 0 & s_1\cdots 0 & & 0 & \\ & & & \ddots & & & \\ 0 & \cdots & & 0 & 0 & s_1 & \end{bmatrix}\begin{matrix} \frac{n}{2} \\ \\ \\ \\ \\ \\ \frac{n}{2} \end{matrix}=\begin{bmatrix} G_1(s) \\ G_2(s) \end{bmatrix}.$$

Using Theorem 9 of Brockett and Dobkin [4] we have

$$\delta_{\mathscr{F}}\{\tilde{G}(s)\}\geqq\frac{n}{2}+\min_N \delta_{\mathscr{F}}\{G_1(s)+NG_2(s)\}.$$

But it is easy to see that $G_1(s)+NG_2(s)$ is always nondegenerate and thus $\delta_{\mathscr{F}}\{G_1(s)+NG_1(s)\}\geqq n$ and hence

$$\delta_{\mathscr{F}}\{G(s)\}=\delta_{\mathscr{F}}\{\tilde{G}(s)\}\geqq\frac{n}{2}+n=\frac{3n}{2},$$

and it follows that

$$\delta_{\mathscr{F}}\{G(s)\}=\frac{3n}{2}.$$

Similarly, one can prove, using precisely the same type of arguments, that if A is an $n\times n$ matrix which has r J_{l_j}-blocks, $j=1,2,\cdots,r$, $l_j>1$, on the main diagonal and zero elsewhere, then $\delta_{\mathscr{F}}\{I_n, A\}=n+r$ with Card $\mathscr{F}\geqq\max_j\{l_j\}$, and where r is also the number of invariant polynomials of A which don't factor into distinct linear factors.

We now prove that the bounds of Theorem 3.3 are always optimal in the case where the field $\mathscr{F}$ contains the roots of the first invariant polynomial $p_1(t)$ (and hence the roots of all $p_i(t)$, $i=1,2,\cdots,k$).

THEOREM 3.6. *Let G be any $n\times n$ matrix whose invariant polynomials are $p_i(t)$, $i=1,2,\cdots,r$, $p_{r+1}(t)=\cdots=p_n(t)=1$, over a field $\mathscr{F}$ with* Card $\mathscr{F}\geqq\max_i\{\deg p_i(t)\}$. *Suppose that $\mathscr{F}$ contains the roots of $p_1(t)$. Then $\delta_{\mathscr{F}}\{I_n, G\}\geqq n+k$, where k is the number of $p_i(t)$'s which don't factor into distinct linear factors over $\mathscr{F}$.*

Proof. Let $p_1(t)$, $p_2(t),\cdots,p_k(t)$ be the polynomials which don't factor into distinct linear factors over $\mathscr{F}$. By the properties of invariant polynomials, we have

$$p_k(t)\,|\,p_{k-1}(t),\ p_{k-1}(t)\,|\,p_{k-2}(t),\cdots,p_2(t)\,|\,p_1(t)$$

and therefore $p_k(t)$ has a root λ of multiplicity greater than one. Assume that

$$\begin{aligned} p_k(t) &= (t-\lambda)^{\tau_k} q_k(t), \\ p_{k-1}(t) &= (t-\lambda)^{\tau_{k-1}} q_{k-1}(t), \\ &\vdots \\ p_1(t) &= (t-\lambda)^{\tau_1} q_1(t), \end{aligned}$$

where $\tau_k > 1$ and $\tau_i \geqq \tau_j$ for all $j \geqq i$.

By the theory of invariant polynomials [9], G is similar to a matrix of the following form

$$G' = \begin{bmatrix} B_1 & & & & & & \\ & B_2 & & & & & \\ & & \ddots & & & & \\ & & & B_k & & & \\ & & & & C_1 & & \\ & & & & & \ddots & \\ & & & & & & C_k \end{bmatrix},$$

where B_i is a Jordan block of the form

$$B_i = \begin{bmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ & & & & \lambda \end{bmatrix}, \qquad \tau_i \times \tau_i \quad \text{for } i = 1, 2, \cdots, k,$$

and C_j is a companion matrix corresponding to the polynomial $q_j(t)$, $j = 1, 2, \cdots, k$.

Let $H(s) = s_1 I + s_2 G'$ be the corresponding characteristic matrix. Make the following change of variables,

$$s_1 + \lambda s_2 \overset{T}{\mapsto} s_1, \qquad s_2 \overset{T}{\mapsto} s_2.$$

Since this is a nonsingular transformation, the transformed $H(Ts)$ has the same degree. Note that, $H(Ts)$ is given by

$$H(Ts) = \begin{bmatrix} J_{\tau_1}(s) & & & & & \\ & \ddots & & & & \\ & & J_{\tau_k}(s) & & & \\ & & & C_1(Ts) & & \\ & & & & \ddots & \\ & & & & & C_k(Ts) \end{bmatrix},$$

where, as before, $J_{\tau_i}(s)$ is given by

$$J_{\tau_i}(s) = \begin{bmatrix} s_1 & s_2 & & & \\ & s_1 & s_2 & & \\ & & \ddots & \ddots & \\ & & & s_1 & s_2 \\ & & & & s_1 \end{bmatrix}, \qquad \tau_i \times \tau_i, \quad i = 1, 2, \cdots, k.$$

Using Theorem 3.5, we obtain

$$\delta_{\mathcal{F}}\begin{bmatrix} J_{\tau_1}(s) & & \\ & \ddots & \\ & & J_{\tau_k}(s) \end{bmatrix} = \sum_{i=1}^{k} \tau_i + k.$$

Moreover, note that the matrix $\operatorname{diag}\{C_1(Ts), \cdots, C_k(Ts)\}$ is nondegenerate. Apply now Theorem 10 of Brockett and Dobkin [4] to get

$$\delta_{\mathcal{F}}\{H(s)\} = \delta_{\mathcal{F}}\{H(Ts)\} \geqq n + k.$$

Therefore

$$\delta_{\mathcal{F}}\{I_n, G\} = n + k$$

and the proof of the theorem is complete.

COROLLARY 3.6.1. *Let $B_1 = x^T G_1 y$ and $B_2 = x^T G_2 y$ be a pair of $p \times q$ bilinear forms over a field $\mathcal{F}$ with large enough cardinality. Let $p_1(t), p_2(t), \cdots, p_r(t), p_{r+1}(t) = \cdots = p_l(t) = 1$ be the invariant polynomials of the pencil $G_1 + tG_2$. If $\mathcal{F}$ contains the roots of $p_1(t)$, then*

$$\delta_{\mathcal{F}}\{B_1, B_2\} = p + q + k - \operatorname{rank}(G_1 + \lambda G_2),$$

where k is the number of $p_i(t)$'s which don't factor into distinct linear factors over $\mathcal{F}$.

COROLLARY 3.6.2. *Let B_1 and B_2 be as defined in Corollary* 3.6.1 *where $\mathcal{F}$ is algebraically closed. Then*

$$\delta_{\mathcal{F}}\{B_1, B_2\} = p + q + k - \operatorname{rank}(G_1 + \lambda G_2).$$

Note that, in the proof of Theorem 3.6, we only used the fact that $p_k(t)$ has a multiple root. Thus, the result is true in the case where $p_k(t)$ has a multiple root in $\mathcal{F}$ even if it does not split completely over $\mathcal{F}$. On the other hand, if $p_k(t)$ has a multiple root in any field $\mathcal{F}'$ containing $\mathcal{F}$, then

$$\delta_{\mathcal{F}}\{I_n, G\} \geqq \delta_{\mathcal{F}'}\{I_n, G\} \geqq n + k,$$

and the result holds true in this case too.

The only case which we could not settle is when some $p_j(t)$'s split into products of distinct linear factors over a field extension of $\mathcal{F}$.

We close this section by conjecturing that our bounds are always optimal.

4. Conclusion and acknowledgments. In this paper, a general class of bilinear problems has been solved with the aid of deep results from linear algebra. Even though many important problems have been excluded, these results present a precise analysis of the complexity of a general pair of bilinear forms and its dependence on the algebraic structure of the set of constants used. Several ideas and techniques have been developed which can be used to handle more general problems. However, the general problem remains obscure and it seems that completely new techniques are needed; the main issue remains to be the development of new techniques for proving nontrivial lower bounds.

I would like to express my gratitude to Professor R. W. Brockett for his constant help, encouragement and continual guidance during his supervision of this research while the author was at Harvard University. Many thanks to the referees who read the manuscript carefully and whose comments were helpful.

One of the referees has brought reference [11] to our attention where Gastinel has obtained upper bounds similar to ours. Recently, Atkinson and one of his students [3] have established Corollary 3.3.2 independently.

REFERENCES

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

[2] M. D. ATKINSON AND N. M. STEPHENS, *The multiplicative complexity of two bilinear forms*, manuscript communicated by Atkinson.

[3] M. D. ATKINSON, private communication.

[4] R. W. BROCKETT AND D. DOBKIN, On the Optimal Evaluation of a Set of Bilinear Forms, Linear Algebra and Appl., 19 (1978), pp. 207–235.

[5] ———, *On the number of multiplications required for matrix multiplication*, this Journal, 5 (1976), pp. 624–628.

[6] D. DOBKIN, *On the arithmetic complexity of a class of arithmetic computations*, Harvard University Thesis, Cambridge, MA, September 1973.

[7] C. M. FIDUCCIA, *On obtaining upper bounds on the complexity of matrix multiplication*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, NY, 1972.

[8] C. M. FIDUCCIA AND Y. ZALCSTEIN, *Algebras having linear multiplicative complexities*, Department of Computer Science, State University of New York at Stony Brook, Technical Report no. 46, August 1975.

[9] F. R. GANTMACHER, *The Theory of Matrices*, vols. 1 and 2, Chelsea Publishing Company, New York, NY, 1959.

[10] N. GASTINEL, *Sur le Calcul des Produits de Matrices*, Numer. Math., 17 (1971), pp. 222–229.

[11] N. GASTINEL, *Le Problème De L'Extension Minimale Diagonale D'un Operateur Linéaire*, no. 235, 1975, manuscript communicated by one of the referees.

[12] D. JU GRIGOR'EV, *On the algebraic complexity of computing a pair of bilinear forms*, Investigations on Linear Operators, N. K. Nikol'sku, ed., Izdat. "Nauka" Leningrad Otdel., Leningrad, 1974, pp. 159–163.

[13] J. HOPCROFT AND L. KERR, *On minimizing the number of multiplications necessary for matrix multiplication*, SIAM J. Appl. Math., 20 (1971), pp. 30–36.

[14] J. HOPCROFT AND J. MUSINSKI, *Duality applied to the complexity of matrix multiplication and other bilinear forms*, this Journal, 2 (1973), pp. 159–173.

[15] T. D. HOWELL, *Tensor rank and the complexity of bilinear forms*, Ph.D. Thesis, Cornell University, Sept. 1976.

[16] T. D. HOWELL AND J. C. LAFON, *The complexity of the quaternion product*, TR 75-245, Department of Computer Science, Cornell University, June 1975.

[17] J. JA' JA', *On the algebraic complexity of classes of bilinear forms*, Ph.D. Thesis, Harvard University, Sept. 1977.

[18] ———, *Computation of bilinear forms over finite fields*, Technical Report CS-78-03, Department of Computer Science, Pennsylvania State University, Jan. 1978.

[19] J. LADERMAN, *A noncommutative algorithm for multiplying* 3×3 *matrices using* 23 *multiplications*, Bulletin of the American Mathematical Society, vol. 82, no. 1, January 1976.

[20] J-C LAFON, *Optimum Computation of p Bilinear Forms*, Linear Algebra and Appl., 10 (1975), pp. 225–260.

[21] Y. MATIJASEVIC, *Enumerable sets are diophantine*, Dokl. Acad. Nauk, SSSR., 191 (1970), pp. 279–282. (In Russian.)

[22] I. MUNRO, *Problems related to matrix multiplication*, Proceedings Courant Institute Symposium on Computational Complexity, New York, Oct. 1971.

[23] M. NEWMAN, *Integral Matrices*, Academic Press, New York, 1972.

[24] R. PROBERT, *On the complexity of symmetric computations*, University of Waterloo Computer Science Technical Report CS-73-02, Waterloo, Ontario, Jan. 1973.

[25] V. STRASSEN, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), pp. 354–356.

[26] ———, *Evaluation of rational functions*, Complexity of Computer Computations, R. Miller and J. Thatcher, eds., Plenum Press, 1972.

[27] ———, *Vermeidung von Divisioven*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

[28] S. WINOGRAD, *On multiplication of* 2×2 *matrices*, Linear Algebra Appl., 4 (1971), pp. 381–388.

EFFICIENT ORDERING OF HASH TABLES*

GASTON H. GONNET and J. IAN MUNRO†

Abstract. We discuss the problem of hashing in a full or nearly full table using open addressing. A scheme for reordering the table as new elements are added is presented. Under the assumption of having a reasonable hash function sequence, it is shown that, even with a full table, only about 2.13 probes will be required, on the average, to access an element. This scheme has the advantage that the expected time for adding a new element is proportional to that required to determine that an element is not in the table. Attention is then turned to the optimal reordering scheme and the minimax problem of ordering the table so as to minimize the length of the longest probe sequence to find any element. Both arranging problems can be translated to assignment problems. A unified algorithm is presented for these, together with the first method suggested. A number of simulation results are reported, the most interesting being an indication that the optimal reordering scheme will lead to an average of about 1.83 probes per search in a full table.

Key words. hashing; open addressing; maximum flow; table searching; assignment problem; analysis of algorithms; asymptotic analysis; simulation

1. Introduction. Hash coding techniques are commonly used to quickly enter and retrieve information from tables. Indeed, they provide the possibility of retrieving data from an n entry table in a number of probes bounded (on the average) by a constant, rather than log log n (all logarithms are to base 2 unless otherwise noted) (Gonnet [4], Yao and Yao [13]) for interpolation search, or log n for binary search. Recently, Guibas, Knuth and Szemeredi [7,8,9] performed very sophisticated analysis of the behavior of hashing techniques. The thrust of this work has, however, not been to provide new and better techniques, but as noted, a more sophisticated analysis of fairly standard methods. The state of the art of hashing remains essentially as follows:

(i) If chaining (i.e., the additional storage of a pointer as part of each record) is permitted, then the search for an element which has been hashed to a full table can be conducted in an average of 1.5 probes. The permanent retention of pointers in the table is very often unacceptable. We will be concerned with the situation in which no such auxiliary pointers are allowed, but extra storage may be used to determine the appropriate insertions to be made. For many applications this is precisely what is required.

(ii) The usual technique (when chaining is not allowed) of entering an element by rehashing until an empty location is found (simple open addressing) is quite acceptable until the table begins to fill. The average search time in a full table is, however, $\ln(n)+O(1)$, and the expected worst case is $O(n)$ (i.e., it will probably take $O(n)$ probes to find some element, in particular $n/2$ for the last one inserted).

(iii) Brent [1] has suggested a method of reordering the table slightly as new elements are inserted. This leads to about 2.49 probes on the average for a retrieval from a full table, and an expected worst case of $O(n^{1/2})$.

* Received by the editors September 6, 1977.

† Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1. This research was supported in part by NSERC under grant A8237 and by the University of Waterloo under operating grant 126-7029. A preliminary version of this paper was presented at the 9th Annual ACM Symposium on the Theory of Computing (May 1977).

The contribution of this paper is a new reordering scheme which is still practical and leads to an average of roughly 2.13 probes for retrieval from a full table and, apparently, an expected longest search of $O(\log n)$ probes. Furthermore we examine the problem of finding the arrangements to minimize the average retrieval time and to minimize the worst probe sequence. We find a close correspondence between organizing hashing tables and the assignment problem. For example, some results reported by Kurtzberg [10] correspond to open addressing hashing. The improvements and bounds obtained by Donath [2] correspond to Brent's reorganizing scheme. The Edmonds and Karp [3] algorithm for assignment problems corresponds to our optimal hashing algorithms.

Several simulation results are presented.

2. A reordering scheme. The essence of our algorithm is that when a key to be added to the table hashes to a location already occupied, it is essentially irrelevant which of the two colliding keys is located there, and which is moved to its next choice. Hence, if only one of them hashes next to a free location, it is placed there, while the other retains the original spot. Extending this idea another step, if both of these secondary locations are also occupied, there are (in general) 4 locations at the next level to check. Carrying the idea to its logical conclusion, we perform a breadth first search of the *binary tree* generated by these locations and subsequent rehashes of the keys encountered until an empty location is found. In the example of Figure 1, the element to be inserted, a, hashes to a location currently occupied by b (b may or may not be in its primary location). The secondary location for a is occupied by c, and the next location for b by d. At the third level, however, we see that b hashes into an empty spot, and so b is moved there and a is placed in its primary location. The effect of adding a on the average retrieval time for all the elements in the system is equivalent to that of being able to insert a in its third location.

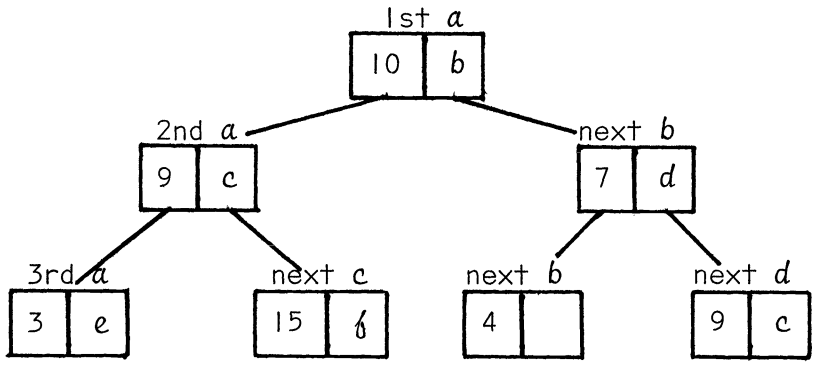


FIG. 1. *The search conducted in adding a to the table and the relevant segment of the hashing function.*

Note that rather than searching for an empty location by sampling without replacement (simply rehashing on a) at a cost of 1 more probe per sample whenever a search is performed for a, we are essentially sampling with replacement (note the probing of location 9 on two paths), but at an effective cost of the logarithm of the

number of locations sampled when searches are performed. The fact that an element hashes into a permutation of the table locations (i.e., a path from any node in the tree of Figure 1 which always takes the left branch has no repetitions) does not significantly help us. We note that an elegant implementation of the search is achieved by representing the search tree as an array with the 2i, 2i+1 - heap style technique of determining the left and right sons of a node. The binary representation of the heap position of the first empty location indicates the way in which the table is to be rearranged.

3. Analyses of the average number of probes required. For purposes of the following preliminary analysis, we assume, that the sequence of probe positions is random and independent. Under this assumption, the number of probes, j, needed to find the first empty position in a table with m locations containing n elements has a geometric distribution with parameter α, that is $(1-\alpha)\alpha^{j-1}$, where $\alpha = n/m$ is the load factor.

This gives, during the search, an expected number of probes, $1/(1-\alpha)$, and an overall average for the first n insertions of

$$E(\textit{accesses}) = \alpha^{-1}[H_m - H_{m-n+1}] \sim -\alpha^{-1}\ln(1-\alpha),$$

where H_n is the n^{th} harmonic number.

Counting the root of the search tree to be a depth 1, the average depth at which the first empty slot is found is

$$\sum_{j=1}^{\infty}(1-\alpha)\alpha^{j-1}(\lfloor \log_2 j \rfloor + 1) = \sum_{k=0}^{\infty}\alpha^{2^k-1} = D(\alpha).$$

There is no known closed form for $D(\alpha)$, although the series, being doubly exponential, converges very rapidly for $\alpha<1$. Furthermore an asymptotic analysis [5] shows that when $\alpha \to 1^-$ then

$$\begin{aligned} \alpha D(\alpha) &= -\log_2(-\log_2\alpha) + \frac{1}{2} - \frac{\gamma}{\ln 2} + P(\log_2(-\log_2\alpha)) \\ &\quad + (1-\alpha) + \frac{1}{3}(1-\alpha)^2 + \frac{4}{21}(1-\alpha)^3 + O((1-\alpha)^4) \end{aligned}$$

where $P(x)$ is periodic with period 1, and can be disregarded for practical purposes since

$$\left|P(x)\right| < 0.0000032.$$

This means that the last element inserted in a complete table increases the total path length by:

$$D\left(\frac{m-1}{m}\right) = \log_2 m + O(1).$$

$D(\alpha)$, then, represents the expected length of the path to locate the new element, plus the increase in length of paths to previously located elements. From the point of view of determining the average path length, it is the effective contribution of adding the new element. We conclude, then, that the expected average path length when n elements have been inserted is

$$E(\textit{path length}) = \frac{1}{n}\sum_{k=0}^{n-1} D(k/m) < \alpha^{-1}\int_0^{\alpha} D(p)\,dp$$
$$= \sum_{k=0}^{\infty} 2^{-k}\alpha^{2^k-1} = \overline{D}(\alpha) \leqslant \overline{D}(1) = 2.$$

Another quantity which may be of interest is the expected number of moves required during insertion. Let $v(j)$ denote the number of 1's in the binary representation of j. Referring back to Figure 1, we see that the number of elements which are moved from their previous locations, while making an insertion, is precisely $v(j)-1$, where the j^{th} location inspected is the first empty one found. An expression for the expected number of moves may be derived as

$$E(\textit{moves}) = \sum_{j=1}^{\infty} (v(j)-1)\,(1-\alpha)\alpha^{j-1}.$$

Decomposing $v(j)-1$ for each of its bit components we have

$$\begin{aligned} E(\textit{moves}) &= \alpha^{-1}(1-\alpha)[(\alpha^3+\alpha^5+\alpha^7+\alpha^9+\ldots) \\ &\qquad + (\alpha^6+\alpha^7+\alpha^{10}+\alpha^{11}+\alpha^{14}+\ldots) \\ &\qquad + (\alpha^{12}+\alpha^{13}+\alpha^{14}+\alpha^{15}+\alpha^{20}+\alpha^{21}+\ldots)\ldots] \\ &= \alpha^{-1}[(\alpha^3-\alpha^4+\alpha^5-\alpha^6+\ldots) \\ &\qquad + (\alpha^6-\alpha^8+\alpha^{10}-\alpha^{12}+\ldots) \\ &\qquad + (\alpha^{12}-\alpha^{16}+\alpha^{20}-\alpha^{24}+\ldots) + \ldots] \\ &= \alpha^{-1}\sum_{k=0}^{\infty} \frac{\alpha^{3\times 2^k}}{(1+\alpha^{2^k})} = M(\alpha). \end{aligned}$$

Again, we know of no closed form for $M(\alpha)$, but it converges rapidly for $\alpha<1$. The expected number of moves of elements already in the table per insertion to fill a table up to a load factor of α is

$$\begin{aligned} E(\text{moves}) &= \overline{M}(\alpha) = \frac{1}{n}\sum_{k=0}^{n-1} M(k/m) \\ &= \alpha^{-1}\int_0^{\alpha} M(p)\,dp + O(m^{-1}) \\ &= \alpha^{-1}\sum_{k=0}^{\infty} 2^{-k}\ln(1+\alpha^{2^k}) - 1 + O(m^{-1}) \\ &\leqslant \overline{M}(1) = \ln(4)-1 = 0.386294\ldots . \end{aligned}$$

This indicates, of course, that complicated sequences of moves happen very rarely.

The approximation of the distribution of the number of probes needed to make an insertion as geometric is rather good for a load factor of .8 or less. Indeed if it held for $\alpha=1$ we could expect to be able to access information from a full table in an average of 2 probes. Unfortunately this approximation leads to an error of a few percent as the table becomes very full. A flaw in the model is that it does not take into account the fact that short chains of probe positions tend to grow more quickly than at random. Following an approach similar to Brent [1], we define $p_i(\alpha)$ to be the probability that given that a key, K, is in h_s, the s^{th} position of its hash sequence, that the next i probe positions, $h_{s+1}, h_{s+2}, \cdots h_{s+i}$, are occupied. This

last sequence of occupied positions will be called the *chain* of K. An equivalent way of defining $p_i(\alpha)$ (or p_i for short) is by

$$p_i(\alpha) = \frac{E(\text{number of chains of length} \geqslant i)}{n}.$$

We will now study the behavior of the quantity $\alpha p_i(\alpha)$ which represents the probability of finding a chain of length i starting at any random location.

Inserting one key and studying the growth of chains, we derive the following system of difference equations:

$$\begin{aligned}(\alpha+1/m)\times p_i(\alpha+1/m) - \alpha p_i(\alpha) &= \frac{\alpha^i}{m} \quad \text{(creation of a new chain)} \\ &+ \frac{1}{m(1-\alpha)}\sum_{j=0}^{i-1}\alpha^{i-j}[p_j(\alpha)-p_{j+1}(\alpha)] \\ &\quad \text{(extension of a chain by the random placement of the new key)} \\ &+ \frac{1}{m}\sum_{j=0}^{i-1}\alpha^{i-j-1}Q_j \\ &\quad \text{(extension of a chain caused by the binary tree insertion).}\end{aligned}$$

Here Q_j denotes the sum of the probabilities of all binary trees for which a breadth first search for a free location ends in a chain of length j. For example we have

$$\begin{aligned}Q_0 &= \alpha^2(1-p_1)\{1+\alpha p_1+\alpha p_1 p_2+\alpha^2 p_1^2 p_2+\alpha^2 p_1^2 p_2^2+\ldots\} \\ Q_1 &= \alpha^3(p_1-p_2)p_1\{1+\alpha p_1 p_2+\alpha p_1^2 p_2 p_3+\alpha^2 p_1^3 p_2^2 p_3+\ldots\} \\ &\ldots\ldots\end{aligned}$$

We will now explain each of the summands of the right hand side of the difference equations in terms of Figure 1.

The first summand comes from the creation of a new chain, in Figure 1, the chain in positions 10, 9, 3, ... Note that since a is not yet in the table, the locations composing this chain are still independent.

The second summand appears from the extension of unrelated chains by the filling of an empty table position. In the example, location 4 will be filled, and consequently any previous chain (not necessarily related to the present construction) that was terminated by location 4, will now be extended. Observe that if a chain is extended (one location) by such an insertion, it may be extended several more positions by the random location of other keys. Note that the contributions of the first two summands are a consequence of any open-addressing scheme.

The last summand represents the extension of a chain originating in the binary tree search. In our example location 4 (and consequently the chain starting at b) has a higher than random probability of being filled by belonging to the full binary tree 10, 9, 7, 3, 15. The locations in the chain following the empty position are independent.

The crux of our use of the $p_i(\alpha)$ is that they carry information concerning the expected length of a chain which, intuitively, we expect to be larger than for uniform or random probing.

This model is not exact; there are several approximations. The most significant is that after insertion, some keys may be moved forward. This will reduce the length of a particular chain, (b in the example will now be located in position 4) while it increases the length of the new one (a will be guaranteed to have a chain of length 2). The total effect on the average length of a chain cancels out exactly, but it may alter the distribution of the $p_i(\alpha)$ slightly.

Straightforward manipulation of the above expressions shows that the total increment in the number of accesses is given by

$$D^*(\alpha) = 1+\alpha+\alpha^2 p_1+\alpha^3 p_1^2 p_2+\alpha^4 p_1^4 p_2^2 p_3+\alpha^5 p_1^8 p_2^4 p_3^2 p_4+\ldots$$

and the average number of accesses is then

$$\overline{D}^*(\alpha) = \alpha^{-1}\int_0^{\alpha} D^*(t)\,dt$$

Taking the limit as $m\to 0$ in the above equations we derive an infinite system of differential equations. We can find the solution in terms of a power series in α, obtaining

$$\overline{D}^*(\alpha)=1+\alpha/2+\alpha^3/4+\alpha^4/15-\alpha^5/18+17\alpha^6/105+53\alpha^7/720-\ldots.$$

This series does not provide a reasonable method of determining $\overline{D}^*(\alpha)$ as α approaches to 1, but we can obtain reasonably good numerical approximations by numerically integrating the system of differential equations.

A similar analysis on the expected number of moves can also be performed. Using the $p_i(\alpha)$, we define

$$\begin{aligned} M^*(\alpha) &= \alpha^2(1-p_1)\{1+\alpha p_1+2\alpha p_1 p_2+\alpha^2 p_1^2 p_2+\cdots\} \\ &\quad + \alpha^3 p_1(p_1-p_2)\{1+\alpha p_1 p_2+\ldots\}+\ldots \end{aligned}$$

and

$$\overline{M}^*(\alpha) = \alpha^{-1}\int_0^{\alpha} M^*(t)\,dt.$$

Table 1 shows $\overline{D}^*(\alpha)$ and $\overline{M}^*(\alpha)$ obtained by numerical integration.

We observe that the numerically computed $\overline{M}^*(\alpha)$ is, in each case, slightly smaller than $\overline{M}(\alpha)$. This may appear inconsistent, but is explained by the fact that long chains do not require more moves.

A number of simulations were performed in order to test the accuracy of our analysis. These, and all our other hashing experiments, use the double hashing scheme noted in the appendix to generate the hash probe sequences. This was done in order to make extensive testing feasible. We claim that for all the insertion schemes that we use, there will be no noticeable difference between this scheme and that of random probe sequences. The appendix contains a comparison of the two methods of probe sequence generation for fairly small tables.

Table 2 shows a typical experiment on a table of size 997 with various load factors (the $\pm$ terms indicate 95% central confidence limits). It is tedious, but not difficult, to rework our predictions of average behavior for the non-asymptotic case and see that for all intents and purposes the limiting behavior is achieved with tables of a few hundred elements. For this reason we are able to compare our experimental results with predicted asymptotic behavior. Note that in all cases our theoretical average is well within the confidence interval of the experimental, and furthermore it

TABLE 1
Average number of accesses and moves for Binary Tree hashing

α	$\overline{D}^*(\alpha)$	$\overline{M}^*(\alpha)$
0.20	1.10209	0.01159
0.40	1.21746	0.04192
0.60	1.36362	0.09124
0.80	1.57886	0.17255
0.85	1.65554	0.20264
0.90	1.75084	0.24042
0.95	1.88038	0.29200
0.96	1.91376	0.30526
0.97	1.95143	0.32015
0.98	1.99525	0.33733
0.99	2.04938	0.35819
1.00	2.13414	0.38521

is neither consistently higher nor lower. The average p.q.o. (priority queue operations in the implementation) column is a good measure of the cumulative time required for all the insertions. Another point of interest is that our preliminary analysis predicts an average of about 1.56 probes for a large table with $\alpha = .8$. We note this is not far off our improved and experimental results. Above this load, however, the difference becomes more significant reaching roughly .05 at 90% (the estimated average is roughly 1.70) and .13 at 100%, since our preliminary analysis predicts an average of 2.

TABLE 2
Simulation of Binary tree hashing
Size of table = 997 number of files (sample size) = 250

occup. factor	number of records	theor. average	average accesses	average max. acc.	average p.q.o.
80%	798	1.5789	1.58061±0.00302	6.184±0.114	2563.1±15.0
90%	897	1.7508	1.74778±0.00381	7.272±0.128	4206.3±31.6
95%	947	1.8804	1.87867±0.00433	8.316±0.152	6365.1±68.4
99%	987	2.0494	2.04991±0.00431	9.692±0.161	14250.±242.

Table 3 indicates the behavior of our scheme on full tables.

TABLE 3
Binary Tree hashing with a full table

file size	sample files	average accesses	average max. acc.	average p.q.o
19	1000	1.8903±0.0138	5.083±0.0914	106.11±2.26
41	1000	2.0053±0.0105	6.438±0.0984	331.25±6.41
101	400	2.0758±0.0107	7.855±0.156	1229.8±33.1
499	100	2.1358±0.0104	10.78±0.357	12612.±462.
997	50	2.13466±0.00958	11.02±0.443	31587.±1487.

Another interesting point is the behavior of the average of the maximum number of probes needed to access any element in a full table. From the analyses of the insertion scheme we see that as the table becomes full, the depth of search required for an insertion will become $\sim\log n$ on the average. Based on this we can expect the length of the longest probe sequence required to access an element to be $O(\log n)$ as well. Our experimental results in Table 3 suggest that may well be very close to $\log_2(n)+c$ (where c is roughly 1).

An efficient implementation of this algorithm, which was used to obtain the simulation results, is described with the optimal allocation algorithm.

4. The optimal arrangement. It is not difficult to construct examples in which our ordering scheme does not provide the best possible arrangement of a set of keys, given their hash sequences. This is a result of the fact that a key tentatively assigned to the i^{th} location in its probe sequence can never be moved to an earlier one, regardless of the new keys added to the table. However, one might wonder how far from the cost of the optimal arrangement the one outlined above tends to be. Before making a comparison we briefly discuss the problem of determining the optimal arrangement.

The problem of optimal allocation is, as Rivest [12] has also observed, a special case of an assignment or minimum cost network flow problem. The solution to the assignment problem which we outline is essentially that of Edmonds and Karp [3]. and Karp [3]). In the terminology of network flows, we can construct a directed network with nodes

(i) a source, s, and terminal node t

(ii) the keys K_i

(iii) the locations l_i

and arcs with cost Δ at a particular time

(s,K_i);	$\Delta(s,K_i) = 0$	for all K_i not assigned
(l_i,t);	$\Delta(l_i,t) = 0$	for all l_i empty
(K_i,l_j);	$\Delta(K_i,l_j) = p$	if K_i is not assigned to l_j and K_i probes to l_j in its p^{th} probe
$(l_j,K_i)=-p$		if K_i is assigned to l_j in its p^{th} probe.

The assignment of a new key is translated to an augmentation of the flow from s to t. This is done by finding a minimum cost path from s to t.

In hashing terms this is equivalent to finding a minimum cost path (way of rearranging) from an unassigned key to an empty table location. For example consider the probe sequences for the keys K_1 to K_4 indicated below:

probe positions

$K_1 \rightarrow 1, 4, 3, 2$
$K_2 \rightarrow 2, 3, 4, 1$
$K_3 \rightarrow 2, 4, 1, 3$
$K_4 \rightarrow 4, 2, 1, 3$

After we assign $K_1 \rightarrow 1$; $K_2 \rightarrow 2$ and $K_4 \rightarrow 4$ (that is an optimal partial assignment) the resulting network is given by Figure 2.

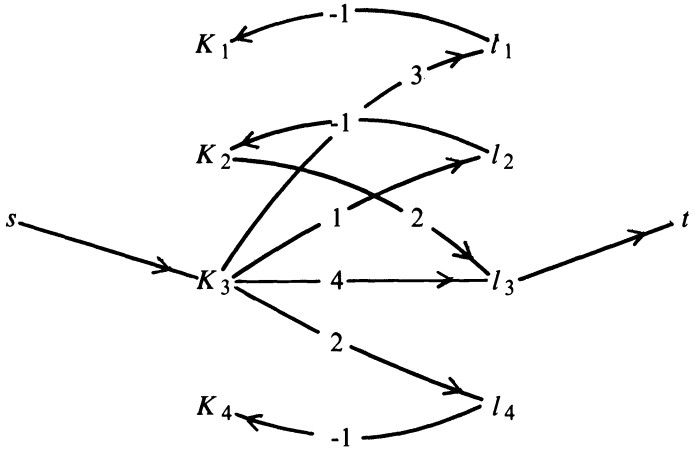


FIG. 2. *The network resulting from an optimal partial assignment (some arcs are omitted for clarity).*

Now if we are to insert K_3, we discover that a minimum cost path is $s \rightarrow K_3 \rightarrow l_2 \rightarrow K_2 \rightarrow l_3 \rightarrow t$. The cost of this path is 2 and the final assignment is

$K_1 \rightarrow l_1$
$K_2 \rightarrow l_3$
$K_3 \rightarrow l_2$
$K_4 \rightarrow l_4$

which is optimal.

To implement the optimal arrangement we use Dijkstra's algorithm to find the minimum cost path from s to t. Although the network contains negative arcs it can be demonstrated that in our case, this causes no problems. With these considerations, the algorithm can be coded with some redundancies in pseudo Algol 68 as follows

```
int n; # is the number of keys to locate in the table #
int m; # is the number of table entries #
[1:m+1] int
key, # contains the key number in location i; 0 if not occupied #
cost, # contains number of probes used to locate key in location #
sigma; # used to find a minimum cost path #
[1:m] int path; # used to record a minimum cost path #

for i to m+1 do key[i] := 0; cost[i] := 0; sigma[i] := 0 od;
zero := -m-1;
for p to n do
        sigma[m+1] := zero;
        key[m+1] := source key(p);
        clear heap;
        j := m+1;  ppos := 1;
        while true do
                heap ← {j,ppos+1,sigma[j]-zero-cost[j]+ppos+1};
                k := probe(key[j],ppos);
                if sigma[j]-cost[j]+ppos < sigma[k] then
                        sigma[k] := sigma[j]-cost[j]+ppos;
                        path[k] := j;
                        if key[k] = 0 then break while fi;
                        heap ← {k,1,sigma[k]-zero-cost[k]+1} fi;
                {j,ppos,} ← heap
                od;
        while k < m+1 do
                j := path[k];
                key[k] := key[j];
                cost[k] := cost[j]+sigma[k]-sigma[j];
                k := j
                od;
        zero := zero-m-1
        od;
```

Program Notes:

Probe (Key,p) = l gives the p^{th} probe position of Key. The vector, cost, can be avoided if we are able to compute probe^{-1}(Key,l) = p easily. The vector, path, is needed only to perform a simple and efficient trace-back through the minimal path. Note that the hashing function should not be linear probing, since for that scheme any ordering produces the same average number of accesses (Peterson [11]).

To implement a priority queue we use a heap which stores records of three components. Each record represents a node in the network. The first component identifies the associated key, the second, its next probe position, and the third, the path cost up to the node in question. The third element is the ordering value for the priority queue. The use of the variable, zero, is to avoid the initialization of the partial cost vector, sigma, for each key. It is worth noting that if we change the statement

```
heap ←{k,1,sigma(k)-zero-cost(k)+1}
```

to

heap ←{k,cost(k)+1,sigma(k)−zero+1},

in the code above, we obtain an algorithm that only searches for an optimum by moving keys forward in their probe sequence. This is, except for the order in which a level of the binary tree is inspected, our previous algorithm.

Tables 4 and 5 summarize simulation results performed with the optimal algorithm.

TABLE 4
Simulation results for Optimal Hashing
Size of table = 997 Number of sample files = 75

occup. factor	number of records	average accesses	average max. acc.	average p.q.o.
80%	798	1.48902±0.00416	4.4±0.112	7456.±230.
90%	897	1.61039±0.00432	5.1467±0.0888	27973.±1431.
95%	947	1.68918±0.00586	5.68±0.118	79757.±4052.
99%	987	1.78514±0.00583	6.773±0.126	223262.±6931.

TABLE 5
Simulation of Optimal Hashing for full tables.

file size	sample files	average accesses	average max. acc.	average p.q.o
19	1000	1.72895±0.0107	4.385±0.0710	224.13±5.46
41	500	1.78283±0.0111	5.296±0.105	888.3±26.2
101	200	1.79837±0.0105	6.3±0.175	4611.±157.
499	50	1.82381±0.0110	7.92±0.358	89937.±4334.
997	50	1.82794±0.00639	8.98±0.382	332365.±12373.

5. Minimax arrangements. Another natural problem is that of arranging a set of keys in a table such that the length of the longest probe sequence to access any element is minimized. Among all possible minimax configurations we would, of course, like to find the one which produces the minimum average number of accesses. The simulations reported in Section 1 indicate that our original scheme produces an average worst case of about $\log n$ in a full table. Gonnet [6] has demonstrated that for the minimax allocation the average length of the longest probe sequence is bounded below by $\ln(n)+O(1)$.

With a small variation in the optimal algorithm of the preceeding section we can derive a minimax allocation. The change is, simply, not to insert a record in the

heap when its probe position exceeds the current minimax. Since, in the creation phase, we do not know the value of the minimax, we try the procedure for minimax values of 1, 2, ... until it does not fail (i.e. the heap never empties before finding an empty table position). The bound noted above suggests that the run time will be multiplied by $\ln(n)$. As a practical approach, we can improve this by finding the smallest value for the minimax such that at least n different table locations appear in the first minimax probes of the n keys.

The following algorithm constructs a minimax optimal hashing table based upon the above remarks.

```
int n; # is the number of keys to locate in the table #
int m; # is the number of table entries #
[1:m+1] int
key, # contains the key number in location i; 0 if not occupied #
cost, # contains number of probes used to locate key in location #
sigma; # used to find a minimum cost path #
[1:m] int path; # used to record a minimum cost path #

uniq := 0;
for i to m do key[i] := 0 od;
for col to m while uniq<n do
        minmax := col;
        for p to n do
                k := probe(source key[p],col);
                if key[k] = 0 then
                        key[k] := 1;  uniq := uniq+1 fi
                od
        od;
start:
for i to m+1 do key[i] := 0; cost[i] := 0; sigma[i] := 0 od;
zero := -m-1;
for p to n do
        sigma[m+1] := zero;
        key[m+1] := source key(p);
        clear heap;
        j := m+1;  ppos := 1;
        while true do
                if ppos<minmax then
                        heap ← {j,ppos+1,sigma[j]-zero-cost[j]+ppos+1} fi;
                k := probe(key[j],ppos);
                if sigma[j]-cost[j]+ppos < sigma[k] then
                        sigma[k] := sigma[j]-cost[j]+ppos;
                        path[k] := j;
                        if key[k] = 0 then break while fi;
                        heap ← {k,1,sigma[k]-zero-cost[k]+1} fi;
                if empty heap then minmax := minmax+1; goto start fi;
                {j,ppos,} ← heap
                od;
```

```
while k < m+1 do
        j := path[k];
        key[k] := key[j];
        cost[k] := cost[j]+sigma[k]−sigma[j];
        k := j
        od;
zero := zero−m−1
od;
```

Tables 6 and 7 report our simulations of minimax hashing.

TABLE 6
Simulation results for Minimax Optimal Hashing
Size of table = 499 Sample size = 100

occup. factor	number of records	average accesses	average max. acc.	average p.q.o.
80%	399	1.49378±0.00670	3±0	4464.±198.
90%	449	1.64829±0.00785	3.05±0.0429	22120.±1744.
95%	474	1.69945±0.00704	3.99±0.0196	41644.±2787.
99%	494	1.78824±0.00774	5.12±0.0893	77304.±4815.

TABLE 7
Simulation of Minimax Optimal Hashing for full tables.

file size	sample files	average accesses	average max. acc.	average p.q.o
19	1000	1.74858±0.0111	3.929±0.0622	241.04±7.35
41	600	1.79638±0.0102	4.665±0.0877	938.2±31.2
101	250	1.80737±0.0102	5.528±0.140	4851.±231.
499	100	1.82998±0.00807	7.38±0.287	91915.±3396.

6. Conclusion. We have examined the problem of arranging elements in a hash table to reduce the average and also the maximum number of probes required to access an element. The main results are summarized in Figure 3.

The thrust of our work is toward the thesis that rather full hash tables using open addressing can still be extremely efficient structures and competitive with chaining techniques. The principal method discussed has the advantages of fast retrieval and insertion (on the average) even when the table is almost completely full. In

terms of both the expected number of probes to access an element and the potential overhead in making an insertion, it lies halfway between Brent's limited search for an insertion route and the optimal assignment. In practice, it seems quite a reasonable scheme. If, however, the table is more than 80% full and to be referenced an extremely large number of times, it is probably worthwhile finding the optimal assignment

There are a number of interesting problems still open. Clearly the most interesting would be a proof that 1.83 or so probes are required, on the average for retrieval from a full but optimally arranged table. Tight analyses of the expected maximum probe sequence for an access under our scheme or the optimal average strategy are also of interest.

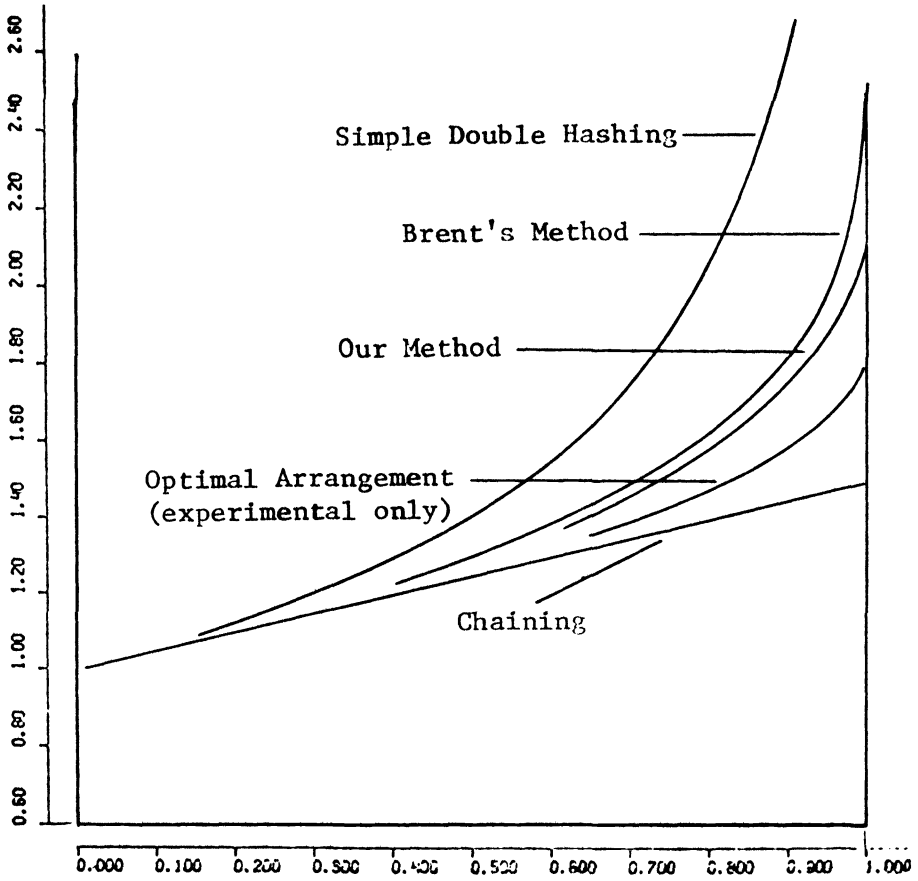


FIG. 3.

7. **Appendix.** Guibas and Szemeredi [8] show that double hashing is equivalent to uniform probing up to a certain load factor. However, all techniques discussed in this paper tend to yield short probe sequences to access elements, even when the table is full. Therefore we claim that for our purposes there is no significant difference between random permutations and double hashing, except from the point

of view of overhead. In actually using a hash table, the cost of generating (and retaining) random probe positions is prohibitive for large tables. Hence all experiments noted in the body of the paper were performed using a double hashing scheme, suggested by Brent [1], which is very useful in practice. The table size, m, is chosen to be prime, the table running from position 0 up to $m-1$. The primary hash location of a key is obtained by taking (the binary number represented by the bit pattern of) the key modulo m, subsequent locations are determined repeatedly by adding (modulo m) the key modulo $(m-2)+1$. The table below shows the results of simulations performed with rather small tables using double hashing (d.h.) and random permutations (r.p.). Note that not only do the average number of probes and average maximum number of probes agree to within their confidence limits in all cases, but also that in some cases the average for double hashing just happened to be lower than for random permutations.

TABLE 8
Comparison of Binary tree hashing using double hashing vs. random permutations.

model	file size	sample files	average accesses	average max. acc.	average p.q.o
d.h.	19	1000	1.8903±0.0137	5.083±0.0914	106.11±2.26
r.p.	19	1000	1.9006±0.0142	5.06±0.0904	107.2±2.26
d.h.	41	1000	2.0053±0.0105	6.438±0.0984	331.25±6.41
r.p.	41	1000	1.9997±0.0101	6.406±0.0942	333.21±6.23
d.h.	101	400	2.0758±0.0107	7.855±0.156	1229.8±33.1
r.p.	101	400	2.0861±0.0108	7.978±0.179	1292.7±36.7

Acknowledgment. The authors thank Richard Lipton and Stanley Eisenstat for many fruitful discussions on the subject of optimal hash assignments and the referee for his/her very precise comments on an earlier manuscript.

REFERENCES

[1] R. P. Brent, *Reducing the retrieval time of scatter storage techniques*, Comm. ACM, 16(1973), pp. 105-109.

[2] W. E. Donath, *Algorithm and average-value bounds for assignment problems*, IBM J. Res. Develop., 13(1969), pp.380-386.

[3] J. Edmonds and R. M. Karp, *Theoretical improvements in algorithmic efficiency for network flow problems*, J. Assoc. Comput. Mach., 19(1972), pp. 248-264.

[4] G. H. Gonnet, *Interpolation and Interpolation Hash Searching*, Doctoral Dissertation, University of Waterloo, Computer Science Dept. Research Report 77-02, Waterloo, Ontario.

[5] ———, *Notes on the derivation of asymptotic expressions from summations*, Information Processing Letters, 7(1978), pp. 165-169.

[6] ———, *Average lower bounds for open-addressing hash coding*, Proceedings of the Conference on Theoretical Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 1977, pp. 159-162.

[7] L. J. Guibas, *The analysis of hashing techniques that exhibit k-ary clustering*, J. Assoc. Comput. Mach., 25(1978), pp. 544-555.

[8] L. J. Guibas and E. Szemeredi, *The analysis of double hashing*, J. Comput. System Sci., 16(1978), pp. 226-274.

[9] D. E. Knuth, *The Art of Computer Programming, Vol III, Sorting and Searching*. Addison-Wesley, Don Mills, Ont (1973).

[10] J. M. Kurtzberg, *On approximation Methods for the assignment problem*, J. Assoc. Comput. Mach., 9(1962), pp. 419-439.

[11] W. W. Peterson, *Addressing for random-access storage*, IBM J. Res. Develop., 1(1957), pp. 130-146.

[12] R. L. Rivest, *Optimal arrangement of keys in a hash table*, J. Assoc. Comput. Mach., 25(1978), pp. 200-209.

[13] A. C. Yao and F. F. Yao, *The complexity of searching an ordered random table*, Proc. 17th Annual IEEE-FOCS Symp., Houston, Texas, Oct. 1976, pp. 173-177.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0001 $01.00/0

OPTIMAL AND NEAR-OPTIMAL SCHEDULING ALGORITHMS FOR BATCHED PROCESSING IN LINEAR STORAGE*

J. R. BITNER† AND C. K. WONG‡

Abstract. In this paper, we consider the accessing of batched requests in a linear storage medium. The batch size is assumed fixed and the access probabilities of individual records known. For a given arrangement of records in the storage, we consider the problem of read/write head scheduling to minimize the expected access time for a batch measured in terms of the distance traveled by the head. In the first part of the paper, several simple algorithms are proposed, analyzed and compared. The effect of different record arrangements is also discussed. In the second part of the paper, a family of algorithms called B-optimal rules are described. When B is ∞, an ∞-optimal rule is indeed optimal in the sense of minimizing expected distance traveled by the head per batch when accessing an arbitrarily large number of batches. A procedure to calculate an ∞-optimal rule for any given record arrangement is described, which is based on the idea of "discrete dynamic programming".

Key words. Batched processing, expected access time, read/write head scheduling, minimization of disk seek time, linear storage medium, near-optimal algorithms, optimal algorithms, scheduling algorithms, B-optimal algorithms, ∞-optimal algorithms, organ-pipe arrangement of records, discrete dynamic programming

1. Introduction.

The problem of positioning a set of records in a linear storage medium in such a way that the expected access time is minimized has been thoroughly studied [1]–[4] when consecutive accesses are independent and the frequencies are all known prior to the placement of the first record. Tape is the prototypical linear storage medium but, when minimization of disk seek time is of interest, it is useful to view the cylinders as forming a linear store. (See, for example, Chapter 5 of [5].) It is sufficient to know the relative frequency of access to the individual records and the optimal solution is obtained by placing the most frequently accessed record and then repetitively placing the next most frequently accessed record alternating between the position immediately to the right of those already placed and the position immediately to the left (the so-called organ-pipe arrangement).

In [6]–[7] the 2-dimensional version of this problem is studied while in [8] the problem of placing records when relative frequencies are not known in advance is discussed. However, in all these works it is assumed that requests to records are processed sequentially, i.e. one request at a time on the first-come-first-served basis.

In the present paper, we shall consider the accessing of batched requests, i.e. we process a fixed number (a batch) of requests at a time. The advantage of batched processing has been discussed thoroughly in [9]. We make the same assumption as before, namely, consecutive accesses are independent and the frequencies are known. Our objective is to minimize expected access time for a batch. Here access time is measured by the distance traveled by the read/write head.

In this model, two problems arise immediately. First, for a given arrangement of the records, what is a good scheduling algorithm (or rule for short) for the head movement? Second, what is a good arrangement of the records?

In this paper we address ourselves primarily to the first problem since we believe that the organ-pipe arrangement is the best arrangement for most reasonable rules. In fact, we prove this for the two simple rules discussed in § 3.

* Received by the editors March 29, 1978, and in final revised form September 15, 1978.

† Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712. The research of this author was supported in part by the University Research Institute at the University of Texas at Austin and NFS Grant MCS 77-02705.

‡ IBM T. J. Watson Research Center, Yorktown Heights, New York 19598.

In § 4, we study a more complicated but intuitively appealing rule, which can be regarded as a member of the family of rules called B-optimal rules studied in § 5. The case when $B = \infty$ is the subject of § 6. An ∞-optimal rule is in fact optimal (in the original sense of minimizing expected access time per batch while accessing an arbitrarily large number of batches). For general B-optimal rules (including $B = \infty$), a finite number of computational steps suffices to determine them completely (for a given record arrangement). Therefore, one can in practice compute an optimal rule and use it.

2. Formulation of the problem. Let the locations of a linear storage be labeled $1, 2, \cdots, n$ from left to right. We use a row n-vector to represent the arrangement of records, e.g. $(R_1, \cdots, R_n)$ means record R_i is at location i for all i. Let p_i be the access probability of record R_i for $i = 1, 2, \cdots, n$. Then $\sum_{i=1}^{n} p_i = 1$. At time t, b requests are generated (with repetitions allowed) according to these probabilities. Let $L_1 \leqq L_2 \cdots \leqq L_b$, $1 \leqq L_i \leqq n$, be the b locations where the requests are made. From now on, L_1, L_b will be referred to as the *left* and *right extremes* respectively and will be written as L and R. Suppose the current head location is x, $1 \leqq x \leqq n$. A *rule* specifies the order in which the head goes through $L_1, L_2, \cdots, L_b$, starting from x. The head stops when all locations have been visited. Let the stopping location be y. The location y will be the starting location for the next batch, i.e. for time $t+1$. Let $d(x, y)$ denote the distance traveled by the head from x to y, whose expected value is our *cost*. In the preent paper, we are only interested in rules whose cost has a definite value. For example, rules whose behavior can be modeled by a Markov chain. A more formal definition of rules will be given in § 6. Our objective is to find rules with as small a cost as possible. In general,we denote the cost by C or $C(R_1, \cdots, R_n)$ if the accord arrangement $(R_1, \cdots, R_n)$ needs emphasis.

3. Two simple rules. In this section, we propose and analyze two simple rules which are called the *Leftist* and *Alternating* rules.

Leftist rule. From the current head location x, move to the extreme L, then sweep across to the right extreme R, and stop.

A *Rightist* rule can be similarly defined. That is, the head moves to the right first and then sweep left.

Alternating rule. When the time t is odd, use Leftist rule, otherwise use Rightist rule.

Let COST_L, COST_R, COST_A be the cost functions of the Leftist, Rightist and Alternating rules respectively. Also, define

$$\lambda_i = \mathrm{Prob}\,(L \geqq i) = \left(\sum_{j=i}^{n} p_j\right)^b,$$

$$\rho_i = \mathrm{Prob}\,(R \leqq i) = \left(\sum_{j=1}^{i} p_j\right)^b,$$

By symmetry, $\mathrm{COST}_L = \mathrm{COST}_R$ (see Corollary 1). We shall derive closed form expressions for COST_L and COST_A.

LEMMA 1. $E(\mathrm{RANGE}_b) = E(R - L) = \sum_{i=1}^{n-1} (1 - \lambda_{i+1} - \rho_i)$.

Proof. Define

$$I_j = \begin{cases} 1 & \text{if record } R_j \text{ is at } L \text{ or } R, \text{ or is in between,} \\ 0 & \text{otherwise.} \end{cases}$$

Then $E(\text{RANGE}_b) = [\sum_{j=1}^{n} E(I_j)] - 1 = [\sum_{j=1}^{n} \text{Prob}\,(I_j = 1)] - 1$. But

$$\text{Prob}\,(I_j = 1) = 1 - \Big(\sum_{i=1}^{j-1} p_i\Big)^b - \Big(\sum_{i=j+1}^{n} p_i\Big)^b;$$

the result follows from substitution. □

THEOREM 1.

$$\text{(a)} \qquad \text{COST}_L = E(\text{RANGE}_b) + \sum_{i=1}^{n-1} (1+\lambda_{i+1})(1-\rho_i) + \sum_{i=1}^{n-1} \lambda_{i+1}\rho_i$$

$$\text{(b)} \qquad = 2E(\text{RANGE}_b) + 2\sum_{i=1}^{n-1} \lambda_{i+1}\rho_i$$

$$\text{(c)} \qquad = 2\sum_{i=1}^{n-1} (1-\lambda_{i+1})(1-\rho_i).$$

Proof. Let L and R be the left and right extremes for the current batch and R_0 the right extreme for the previous batch. COST_L consists of moving from R_0 to L, then from L to R, hence

$$\text{(1)} \qquad \text{COST}_L = E(\text{RANGE}_b) + E(|R_0 - L|).$$

Note that

$$\text{(2)} \qquad E(|R_0 - L|) = \sum_{r=1}^{n} \sum_{s=1}^{n} |r-s| \cdot \text{Prob}\,(R_0 = r, L = s),$$

$$\text{(3)} \qquad \text{Prob}\,(R_0 = r) = \text{Prob}\,(R_0 \leqq r) - \text{Prob}\,(R_0 \leqq r-1) = \rho_r - \rho_{r-1},$$

$$\text{(4)} \qquad \text{Prob}\,(L = s) = \lambda_s - \lambda_{s+1}.$$

Since R_0 and L are independent,

$$E(|R_0 - L|) = \sum_{r=1}^{n} \sum_{s=1}^{n} |r-s| \cdot (\rho_r - \rho_{r-1})(\lambda_s - \lambda_{s+1})$$

$$\text{(5)} \qquad = \sum_{r=1}^{n} (\rho_r - \rho_{r-1}) \sum_{s=1}^{n} |r-s|(\lambda_s - \lambda_{s+1}).$$

The inner sum $= |r-1|\lambda_1 + \sum_{s=1}^{n-1} (|r-s-1| - |r-s|)\lambda_{s+1} + |r-n|\lambda_{n+1} = |r-1| - \sum_{s=1}^{r-1} \lambda_{s+1} + \sum_{s=r}^{n-1} \lambda_{s+1}$ since $\lambda_1 = 1$, $\lambda_{n+1} = 0$ and

$$|r-s-1| - |r-s| = \begin{cases} -1 & \text{if } r > s, \\ 1 & \text{if } r \leqq s. \end{cases}$$

After substituting into (5), we further simplify the expression by splitting the sum, changing an index and recombining. Noting that $\rho_0 = 0$ and $\rho_n = 1$, we obtain the expression for $E(|R_0 - L|)$ as specified in (a). Equations (b) and (c) follow from straightforward manipulation and Lemma 1. □

COROLLARY 1. COST_L *is symmetric*, i.e. COST_L *remains the same if we replace* p_i *by* p_{n-i+1} *for* $i = 1, \cdots, n$. *Consequently*, $\text{COST}_L = \text{COST}_R$.

Proof. Only note that ρ_i and λ_{i+1} are now replaced by λ_{n-i+1} and ρ_{n-i} respectively. The formula in Theorem 1(b) becomes $2\sum_{i=1}^{n-1} \rho_{n-i}\lambda_{n-i+1}$. By replacing i by $n-i$ and reversing the order of summation, the original formula is obtained. □

THEOREM 2.

$$\text{(a)} \qquad \text{COST}_A = E(\text{RANGE}_b) + \sum_{i=1}^{n-1} \rho_i(1-\rho_i) + \sum_{i=1}^{n-1} \lambda_{i+1}(1-\lambda_{i+1})$$

$$\text{(b)} \qquad = \sum_{i=1}^{n-1} (1-\rho_i^2-\lambda_{i+1}^2)$$

$$\text{(c)} \qquad = E(\text{RANGE}_{2b}).$$

Proof. The cost of accessing a batch using this rule consists of: First, either going from the left extreme of the previous batch (L_0) to the left extreme of the current batch (L) or going from the previous right extreme (R_0) to the current right extreme (R). Each possibility has probability $\frac{1}{2}$. Second, a sweep across the batch is needed, requiring a move of $E(\text{RANGE}_b)$. Therefore

$$\text{(6)} \qquad \text{COST}_A = E(\text{RANGE}_b) + \tfrac{1}{2}(E(|L-L_0|) + E(|R-R_0|)).$$

Following the reasoning in Theorem 1, we have

$$E(|L-L_0|) = \sum_{s=1}^{n} (\lambda_s - \lambda_{s+1}) \sum_{m=1}^{n} |s-m|(\lambda_m - \lambda_{m+1}),$$

$$E(|R-R_0|) = \sum_{r=1}^{n} (\rho_r - \rho_{r-1}) \sum_{u=1}^{n} |r-u|(\rho_u - \rho_{u-1}).$$

These are simplified as in Theorem 1, giving

$$E(|L-L_0|) = 2\sum_{s=1}^{n-1} \lambda_{s+1}(1-\lambda_{s+1}), \qquad E(|R-R_0|) = 2\sum_{r=1}^{n-1} \rho_r(1-\rho_r).$$

Substituting into (6) proves part (a). Part (b) is obtained by straightforward manipulation. Part (c) is derived by noting $\rho_r^2 = (\sum_{i=1}^{r} p_i)^{2b}$ and $\lambda_s^2 = (\sum_{i=s}^{n} p_i)^{2b}$ so (b) is exactly the formula for $E(\text{RANGE}_{2b})$. □

Next, we shall show that COST_L (COST_R), COST_A are all minimized when the record arrangement is the organ-pipe arrangement.

Define an *interchange operation*, which, given an arrangement $(R_1, \cdots, R_n)$, creates a new arrangement as follows: For every $i \leqq \lfloor n/2 \rfloor$, R_i and R_{n-i+1} are interchanged iff $p_i > p_{n-i+1}$. Similarly, a *reverse interchange operation* creates a new arrangement by interchanging R_{i+1} and R_{n-i+1} iff $p_{i+1} < p_{n-i+1}$. Record R_1 is ignored.

LEMMA 2. *The organ-pipe arrangement minimizes the cost function, C, of a rule if C satisfies the following conditions for every* $n \geqq 1$:

(1) *C is symmetric, i.e.,* $C(R_1, \cdots, R_n) = C(R_n, \cdots, R_1)$.

(2) *If* $p_n = 0$, *then* $C(R_1, \cdots, R_{n-1}, R_n) = C(R_1, \cdots, R_{n-1})$.

(3) *Given any arrangement* $(R_1, \cdots, R_n)$, *let* $(R_1', \cdots, R_n')$ *be the arrangement created by an interchange operation. Then* $C(R_1, \cdots, R_n) \geqq C(R_1', \cdots, R_n')$.

Proof. We first show that a reverse interchange operation will not increase the cost function. Consider any arrangement $(R_1, \cdots, R_n)$ and let $(R_1', \cdots, R_n')$ be the arrangement created by a reverse interchange operation. Consider the arrangement $(R_1, \cdots, R_n, R_{n-1})$, where R_{n+1} has zero probability. Using conditions (1) and (2), we have $C(R_1, \cdots, R_n) = C(R_1, \cdots, R_n, R_{n+1}) = C(R_{n+1}, R_n, \cdots, R_1)$. Performing an interchange operation on $(R_{n+1}, R_n, \cdots, R_1)$ produces $(R_{n+1}, R_n', \cdots, R_1')$ since R_1 is

compared with a record of zero probability and will not be moved. Therefore

$$C(R_{n+1}, R_n, \cdots, R_1) \geqq C(R_{n+1}, R'_n, \cdots, R'_1)$$
$$= C(R'_1, \cdots, R'_n, R_{n+1}) = C(R'_1, \cdots, R'_n),$$

proving that a reverse interchange operation will not increase the cost function.

It is clear that we can "sort" any initial arrangement into the organ-pipe arrangement by using a sufficiently long alternating sequence of interchange and reverse interchange operations. Since each step will not increase the cost, the cost of the organ-pipe arrangement must be less than or equal to any other arrangement, proving the lemma. □

LEMMA 3. *If* $f(x)$ *is concave* $(f''(x) \leqq 0)$ *on an interval* $[a, b]$, *then for any* $a \leqq x \leqq Y \leqq b$ *and* $\varepsilon > 0$, $f(x)+f(y) \geqq f(x-\varepsilon)+f(y+\varepsilon)$. (Of course, we must also have $x-\varepsilon$, $y+\varepsilon$ in $[a, b]$.)

Proof. Since $f''(x) \leqq 0$, $f'(x)$ is decreasing on $[0, 1]$. By the mean value theorem, for some u such that $x-\varepsilon \leqq u \leqq x$,

$$f(x)-f(x-\varepsilon) = \varepsilon \cdot f'(u) \geqq \varepsilon \cdot f'(x)$$

and for some v such that $y \leqq v \leqq y+\varepsilon$,

$$f(y+\varepsilon)-f(y) = \varepsilon \cdot f'(v) \leqq \varepsilon \cdot f'(y).$$

Then

$$f(x)-f(x-\varepsilon) \geqq \varepsilon \cdot f'(x) \geqq \varepsilon \cdot f'(y) \geqq f(y+\varepsilon)-f(y). \qquad \square$$

THEOREM 3. *The organ-pipe arrangement minimizes* COST_L.

Proof. We show that Lemma 2 holds. Clearly, condition (2) is satisfied. Corollary 1 verifies condition (1). To verify condition (3), let $k = \lfloor (n-1)/2 \rfloor$ and rewrite the formula in Theorem 1(b) as

$$\mathrm{COST}_L = 2\sum_{i=1}^{k}\left[\left(1-\left(\sum_{j=1}^{i} p_j\right)^b\right)\left(1-\left(\sum_{j=i+1}^{n} p_j\right)^b\right)+\left(1-\left(\sum_{j=1}^{n-i} p_j\right)^b\right)\left(1-\left(\sum_{j=n-i+1}^{n} p_j\right)^b\right)\right]$$
$$+2\left(1-\left(\sum_{j=1}^{n/2} p_j\right)^b\right)\left(1-\left(\sum_{j=n/2+1}^{n} p_j\right)^b\right)$$

where the last term is included *only if* n is even. This last term can be written as $x+x$ and then is in the same form as the others with $i = n/2$.

Now consider the term for any $i \leqq n/2$. Let

s = sum of the probabilities of the records in $\{R_1, \cdots, R_i\}$ that are *not* moved by an interchange operation.

t = sum of the probabilities of the records in $\{R_1, \cdots, R_i\}$ that *are* moved.

u = sum of the probabilities of the records in $\{R_{n-i+1}, \cdots, R_n\}$ that are *not* moved.

v = sum of the probabilities of the records in $\{R_{n-i+1}, \cdots, R_n\}$ that *are* moved.

w = sum of the probabilities of the records in $\{R_{i+1}, \cdots, R_{n-i}\}$.

An interchange operation interchanges the t-records with the v-records. Let T be this term before the interchange operation and T' be it after. Then

$$T = (1-(s+t)^b)(1-(u+v+w)^b)+(1-(s+t+w)^b)(1-(u+v)^b,$$
$$T' = (1-(s+v)^b)(1-(u+t+w)^b)+(1-(s+v+w)^b)(1-(u+t)^b).$$

Since $s+t+u+v+w=1$ these can be rewritten as:

$$T=(1-(s+t)^b)(1-(1-s-t)^b)+(1-(s+t+w)^b)(1-(1-s-t-w)^b),$$
$$T'=(1-(s+v)^b)(1-(1-s-v)^b)+(1-(s+v+w)^b)(1-(1-s-v-w)^b).$$

It is easy to verify that the function $f(x)=(1-x^b)(1-(1-x^b))$ is concave over $[0, 1]$. Lemma 3 is used with $x=s+t$, $y=s+t+w$ and $\varepsilon=t-v$. (Note that by the definition of the interchange operation, $t>v$, so $\varepsilon>0$.) Thus, each term is decreased by an interchange operation, therefore COST_L is decreased, verifying condition (3) of Lemma 2 and proving the theorem. □

THEOREM 4. *The organ-pipe arrangement minimizes* COST_A.

Proof. The proof proceeds as in Theorem 3. Conditions (1) and (2) of Lemma 2 are easily verified. To verify condition (3), the formula in Theorem 2(*b*) is rewritten as

$$\mathrm{COST}_A=\sum_{i=1}^{k}\left[1-\left(\sum_{j=1}^{i}p_j\right)^{2b}-\left(\sum_{j=i+1}^{n}p_j\right)^{2b}+1-\left(\sum_{j=1}^{n-i}p_j\right)^{2b}-\left(\sum_{j=n-i+1}^{n}p_j\right)^{2b}\right]$$
$$+1-\left(\sum_{j=1}^{n/2}p_j\right)^{2b}-\left(\sum_{j=n/2+1}^{n}p_j\right)^{2b},$$

where $k=\lfloor(n-1)/2\rfloor$ and the last term is included *only if* n is even.

Now consider any term for $i\leqq n/2$ and define s, t, u, v and w as in Theorem 3. Let T be the term before the interchange operation and T' after. Then

$$T=1-(s+t)^{2b}-(1-s-t)^{2b}+1-(s+t+w)^{2b}-(1-s-t-w)^{2b},$$
$$T'=1-(s+v)^{2b}-(1-s-v)^{2b}+1-(s+v+w)^{2b}-(1-s-v-w)^{2b}.$$

The function $1-x^{2b}-(1-x)^{2b}$ is concave over $[0, 1]$ and Lemma 3 is used with $x=s+t$, $y=s+t+w$ and $\varepsilon=t-v$ to prove the theorem. □

COROLLARY 2. $E(\mathrm{RANGE}_b)$ *is minimized by the organ-pipe arrangement.*

Proof. Theorem 4 shows $E(\mathrm{RANGE}_{2b})$ is minimized by the organ-pipe arrangement. The proof also works if an odd number is used instead of $2b$. □

Next we prove a simple result which compares COST_A, COST_L and the minimum cost.

THEOREM 5. $E(\mathrm{RANGE}_{2b})\leqq 2\cdot E(\mathrm{RANGE}_b)$.

Proof. Using Lemma 1, we have

$$E(\mathrm{RANGE}_{2b})-2\cdot E(\mathrm{RANGE}_b)=\sum_{i=1}^{n-1}1-(1-\lambda_{i+1})^2-(1-\rho_i)^2.$$

Consider any term in the sum and let $x=\sum_{j=i+1}^{n}p_j$ then this term equals $1-(1-x^b)^2-(1-(1-x)^b)^2$ which is nonpositive for x in $[0, 1]$. Since all terms are nonpositive, we have $E(\mathrm{RANGE}_{2b})-2\cdot E(\mathrm{RANGE}_b)\leqq 0$. □

COROLLARY 3.

(1) $\mathrm{COST}_A\leqq\mathrm{COST}_L=\mathrm{COST}_R$

for any record arrangement.

(2) $C_{\min}\leqq C_A\leqq 2\cdot C_{\min}$,

where $C_{\min}$ *is the minimum cost for any rule and any record arrangement;* C_A *is* COST_A *when the organ-pipe arrangement is used.*

Proof. From Theorems 1(b), 2(c), and 5, $\mathrm{COST}_L\geqq 2\cdot E(\mathrm{RANGE}_b)\geqq E(\mathrm{RANGE}_{2b})=\mathrm{COST}_A$.

From Corollary 2, $E(\text{RANGE}_b)$ is minimized by the organ-pipe arrangement. Let this minimum value be E_0. Then clearly, $E_0 \leqq C_{\min}$. Therefore by Theorem 2(c) and 5, $C_A \leqq 2 \cdot E_0 \leqq 2 \cdot C_{\min}$. □

Therefore the Alternating rule is better than the Leftist or Rightist rule and is never more than twice larger than the minimum.

4. The nearest rule. In this section, we study a slightly more complicated rule. In the next section, we shall see that this belongs to a family of rules, called B-optimal rules. In fact, this rule is a 1-optimal rule.

Nearest rule. From the current head location, move to the closer of the two extremes, then sweep across to the other extreme and stop. In case of tie, choose either extreme.

Analysis of this rule is harder than for previous rules because it is difficult to calculate the probability that the head will be at a given position after accessing a batch. For the previous rules, this was simple. It was either the probability that the given position was the right extreme, or one-half times the probability that it was either a left or a right extreme. In the Nearest rule, we know that the head position must be an extreme, but which one depends on the previous head position.

The head position can be described by a Markov chain where the ith state ($i = 1, \cdots, n$) corresponds to having the head at location i. Such a chain is irreducible and aperiodic[1] and hence approaches a unique steady state distribution, $\bar{s} = (s_1, \cdots, s_n)$. This can be found by solving the equation $\bar{s}P = \bar{s}$, or equivalently, $\bar{s}(P - I) = \bar{0}$, where P is the transition matrix of the chain, I is the identity matrix and $\bar{0}$ is a row vector of zeros.

Let $p_{xy} = \text{Prob (head ends up at } y \mid \text{head starts at } x)$. Then the transition matrix is $P = [p_{xy}]$. To compute p_{xy}, recall that record R_i is at location i, and L, R are the left and right extremes respectively. Then we have

$$\text{For } x = y, \qquad p_{xy} = (p_x)^b. \tag{7}$$

$$\text{For } x < y, \qquad p_{xy} = \text{Prob}\,(R = y, L \geqq 2x - y) - \tfrac{1}{2}\text{Prob}\,(R = y, L = 2x - y). \tag{8}$$

$$\text{For } x > y, \qquad p_{xy} = \text{Prob}\,(L = y, R \leqq 2x - y) - \tfrac{1}{2}\text{Prob}\,(L = y, R = 2x - y). \tag{9}$$

Note that

$$\text{Prob}\,(L = u, R \leqq r) = \Big(\sum_{i=u}^{r} p_i\Big)^b - \Big(\sum_{i=u+1}^{r} p_i\Big)^b, \tag{10}$$

$$\text{Prob}\,(L \geqq u, R = r) = \Big(\sum_{i=u}^{r} p_i\Big)^b - \Big(\sum_{i=u}^{r-1} p_i\Big)^b \tag{11}$$

and

$$\text{Prob}\,(L = u, R = r) = \text{Prob}\,(L = u, R \leqq r) - \text{Prob}\,(L = u, R \leqq r - 1) \tag{12}$$

where we consider $p_i = 0$ if $i < 1$ or $i > n$. Substituting (10), (11), (12), into (7), (8), (9) yields p_{xy} and hence the transition matrix P.

Note that if L and R are equidistant from x, the rule randomly chooses L or R with probability $\frac{1}{2}$. This is why the second terms in (8), (9) have to be subtracted; otherwise it is counted twice.

Although P assumes such a simple form, to solve for $\bar{s}$ symbolically appears intractable even for simple cases, such as a uniform distribution (i.e. $p_i = 1/n$, for all i).

[1] For definitions of terms pertaining to Markov chains, see, for example [10]. Note that $\bar{s}$ is a row vector.

However, given a distribution, n, and b, the resulting system of equations is easily solved by computer using numerical techniques, giving $\bar{s}$. Let COST_N be the cost of the Nearest rule. Then

$$\text{COST}_N = E(\text{RANGE}_b) + \sum_{i=1}^{n} \sum_{u \leqq r} s_i \cdot \text{Prob}\,(L = u, R = r) \cdot \min\,(|x-u|, |x-r|).$$

Such a calculation was done for several distributions, and the results are shown in Table 3. To simplify the computation, symmetric versions of Zipf's distribution:

$$p_i = p_{n-i+1} = 1/(2((n/2)-i+1)H), \quad \text{where } H = \sum_{i=1}^{n/2} \frac{1}{i},$$

and the exponential distribution:

$$p_i = p_{n-i+1} = r^{(n/2)-i}/(1-r^{n/2}) \quad \text{with } r = 0.9$$

were used. (Note that the records are in the organ-pipe arrangement.)

For comparison, we did the same calculation for the Leftist and Alternating rules (Tables 1 and 2).

TABLE 1
Asymptotic cost for the Leftist rule.

	b \ n	100	200	300
Uniform		133.389	266.803	400.210
Zipf	5	73.584	134.738	192.346
Expo		58.114	60.303	60.327
Uniform		163.603	327.256	490.897
Zipf	10	107.014	200.347	289.465
Expo		80.964	84.836	84.882
Uniform		174.950	349.976	524.982
Zipf	15	126.223	239.489	348.565
Expo		94.276	99.594	99.660

TABLE 2
Asymptotic cost for the Alternating rule.

	b \ n	100	200	300
Uniform		81.802	163.628	245.449
Zipf	5	53.507	100.174	144.733
Expo		40.482	42.418	42.441
Uniform		90.443	180.936	271.418
Zipf	10	69.393	132.831	194.279
Expo		51.784	55.093	55.136
Uniform		93.498	187.072	280.629
Zipf	15	77.139	149.290	219.704
Expo		58.156	62.615	62.677

TABLE 3
Asymptotic cost for the Neasrest rule.

	b \ n	100	200	300
Uniform		80.753	161.529	242.299
Zipf	5	52.135	97.409	140.588
Expo		39.977	41.972	41.999
Uniform		90.408	180.869	271.317
Zipf	10	68.728	131.211	191.617
Expo		51.698	55.121	55.168
Uniform		93.498	187.070	280.626
Zipf	15	76.870	148.515	218.327
Expo		58.136	62.726	62.795

From these results and intuition, one would expect $\text{COST}_N \leqq \text{COST}_A$ for all distributions, n, and b. Surprisingly, this is not the case. In the appendix, we show that there exists a distribution, n, and b such that $\text{COST}_N > \text{COST}_A$.

Whether or not the organ-pipe arrangement minimizes COST_N is still an open question, but we strongly believe the answer is affirmative.

5. *B*-optimal rules. In this section, we propose a family of rules called B-optimal rules. When $B = 1$, we have the Nearest rule. When $B = \infty$, we have the optimal rule. The latter case will be discussed separately in § 6 since it requires a completely different approach.

Before proceeding further, it is worth mentioning why the Nearest rule is not optimal and what exactly the difficulty is in determining an optimal rule. Although successive batches are assumed to be independent, the final head position for accessing the current batch is the initial head position for the next batch. Some positions, namely, those near where we expect the extremes to be, are more "advantageous" as final head positions, as the expected cost for accessing the next batch will be smaller. An optimal rule must consider this effect in deciding how to access the current batch and must go to the extreme that is farther from the initial head position (instead of the closer one) if the advantage of ending at the closer extreme outweighs the additional cost of moving to the farther extreme instead of the closer.

Consider the situation where the initial head location is x and exactly B batches of b records are to be accessed. A *B-optimal rule* is a rule which minimizes the expected total distance traveled by the head starting from x going through the B batches of requests. We define the cost of a B-optimal rule as the expected total distance divided by B.

Obviously, when $B = 1$, if L and R are the left and right extremes respectively of the batch and x is the head location, then a 1-optimal rule would be to move the head from x to the closer of L and R and then sweep across to the other extreme if necessary. But this is exactly the Nearest rule.

Before we can show rules to be optimal, we have to define what is meant by "rule". The most general definition is that a rule gives a sequence of positions for the head to pass through, given the current head position and the extremes of the current and all previous batches. A rule may be "nondeterministic", giving probabilities that different

sequences should be followed. The following lemmas restrict the form a B-optimal rule can take to a more manageable class.

LEMMA 4. *If a rule is B-optimal, then immediately after accessing all the records in a given batch, the head must be at an extreme.*

Proof. If the last record access is not an extreme, clearly records on both sides of this record (the two extremes) must have already been accessed and we must have already passed through this record to get from one side to the other. The move into this position can then be deleted, giving a rule of lower cost, a contradiction. □

In the following lemma, the total cost for accessing the B batches is divided among the batches: The cost for accessing a batch begins immediately after the last record in the previous batch is accessed and ends immediately after the last record in this batch is accessed.

LEMMA 5. *If a rule is B-optimal, it must access a batch by moving directly to an extreme then sweeping across to the other extreme (if necessary).*

Proof. Since all the records are accessed, both extremes must be accessed. Call the first extreme to be accessed the first extreme and the other the second extreme. If this rule is B-optimal, the first extreme can be accessed only once. Otherwise, a rule which behaves like this rule but deletes any movement between accesses to the first extreme will still access all the records (since both extremes are accessed) and have lower cost, a contradiction.

Thus, the rule must move to one extreme, then to the other. If it does not do so directly, a rule which does will still access all records but have lower cost, again, a contradiction. □

LEMMA 6. *For every rule whose decisions depend on previous batches, there is a rule with equal cost whose decisions depend only on the current batch.*

Proof. Since the batches are independent, a rule which ignores the previous batches will cost the same as one that does not. □

Note that a rule satisfying the previous lemmas is expressible as a sequence, $d_B(x; s, r)$, $d_{B-1}(x; s, r), \cdots, d_1(x; s, r)$ of *decision functions*, where $d_i(x; s, r)$ is the probability that the rule moves left when its head is currently at x for accessing the ith from the last batch with extremes s and r. A rule is *deterministic* if for every i, x, s, and r, $d_i(x; s, r)$ is either zero or one. We now prove the nondeterministic rules are no better than deterministic rules.

LEMMA 7. *For every nondeterministic rule, there is a deterministic rule with smaller or equal cost.*

Proof. Let M be a nondeterministic rule with cost, COST_M. We show how to eliminate one nondeterministic decision. The process is then repeated until a deterministic rule is obtained. Choose any $d_i(x; s, r)$ which is neither zero nor one and let COST_L be the expected cost assuming a left move is made at $d_i(x; s, r)$ and COST_R be that assuming a right move. Clearly,

$$\text{COST}_M = d_i(x; s, r) \cdot \text{COST}_L + (1 - d_i(x; s, r)) \cdot \text{COST}_R$$

and at least one of COST_L and COST_R must be less than or equal to COST_M. Therefore, there is a rule, making a deterministic choice at $d_i(x; s, r)$ that has cost less than or equal to rule M. Continuing in this manner eventually results in a deterministic rule with cost less than or equal to COST_M. □

Note that now all decision functions can be restricted to 0-1 function, meaning only a finite number of rules might possibly be B-optimal.

A final lemma guarantees that an optimal rule is optimal no matter what the initial head position is. This is important; conceivably one rule might be better than another from one position but poorer from another.

LEMMA 8. *An optimal rule is optimal for every initial head position.*

Proof. (The proof is by induction on B.) For $B = 0$, the proof is trivial, so consider any B and let $r_1, r_2, \cdots, r_n$ be rules that are optimal from positions $1, 2, \cdots, n$. These rules must all have the same cost if the first batch is not counted, because they must all use a $(B-1)$-optimal rule to access the remaining batches, which, by induction, is optimal for all head positions. A new rule which accesses the first batch using r_i when it starts at position $i (1 \leqq i \leqq n)$, then uses a $(B-1)$-optimal rule to access the remaining batches will have the same cost as r_i when it starts at position $i (1 \leqq i \leqq n)$ and hence is optimal for all head positions. □

To define the class of B-optimal rules, we need a sequence of functions, $C_B(x)$ for $B = 0, 1, \cdots$, defined by:

$$C_0(x) = 0 \quad \textit{for } 1 \leqq x \leqq n,$$

$$C_B(x) = \sum_{s \leqq r} \text{Prob}\,(L = s, R = r) \cdot \min\,[|x - s| + C_{B-1}(r),$$

$|x - r| + C_{B-1}(s)] + E(\text{RANGE}_b)$ for $1 \leqq x \leqq n$ and $B = 1, 2, \cdots$.

Recursive rule. Let i be the number of batches remaining to be accessed and x the current head position. To access this batch, we have the following rule:

(1) Move to the left extreme, then sweep across to the right if $|x - L| + C_{i-1}(R) < |x - R| + C_{i-1}(L)$.

(2) Move to the right, then sweep left if $|x - L| + C_{i-1}(R) > |x - R| + C_{i-1}(L)$.

(3) Do either if $|x - L| + C_{i-1}(R) = |x - R| + C_{i-1}(L)$.

The optimality of this rule and $C_B(x)$ is proven in the following theorem.

THEOREM 6. *$C_B(x)$ is the optimal cost for accessing B batches, with the head starting at x. The Recursive rule is indeed B-optimal.* (Thus it has cost $C_B(x)/B$.)

Proof. (The proof is by induction on B). For $B = 0$, the proof is trivial. Assume then for all x that $C_B(x)$ is the optimal cost for accessing B batches from x and consider a rule accessing $B + 1$ batches from some position y. By Lemmas 4 and 5, an optimal rule can only access these batches by either:

(1) moving to the left extreme, sweeping across to the right, and then accessing the remaining B batches in an optimal manner with head starting at R (total cost is $|y - L| + E(\text{RANGE}_b) + C_B(R)$), or

(2) moving to the right initially (total cost is $|y - R| + E(\text{RANGE}_b) + C_B(L)$).

Since the $(B+1)$-optimal rule always chooses the option with the smallest cost it must be optimal, and therefore $C_B(x)$ must be the optimal cost. □

Several comments can be made on $C_B(x)$. First, it indicates how advantageous it is to have the head at a given position. The B-optimal rules take this into consideration when accessing a batch; a rule is willing to spend more than the minimum head movement on the current batch (by moving to the further extreme first) if $C_B(x)$ indicates that the difference will be made up on subsequent batches. The shape of $C_B(x)$ is also quite interesting; at first glance one might expect the best position to be near the center of the storage. This, of course, is not the best as is clearly shown by $C_B(x)$ (see Fig. 1). The best locations are near the edges (if b is not too small) because that is the probable location of the extremes.

A final note is that B-optimal rules are not necessarily good rules for accessing long sequences of batches, especially if B is small. Such rules pay too little attention to the

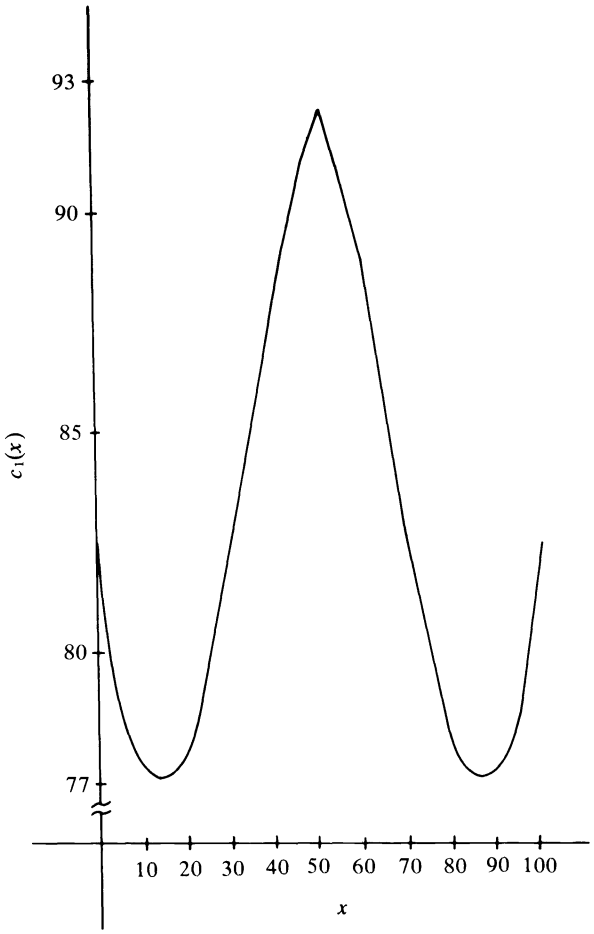


FIG. 1. $C_1(x)$ *for the uniform distribution with* $n = 100$ *and* $b = 5$.

position of the head, concentrating only on minimizing head movement. As an example, the Nearest rule (a 1-optimal rule) can be outperformed by even the Alternating rule (see Appendix) if a large number of batches are to be retrieved.

6. An ∞-optimal rule. In this section, we define and prove correct a procedure to calculate an ∞-optimal rule for any given arrangement $(R_1, \cdots, R_n)$. An ∞-optimal rule is defined in a manner similar to the B-optimal rules of the previous section. If $\text{COST}^r_B(x)$ is the cost of accessing B batches starting from head position x and using rule r, then the *expected asymptotic cost* of rule r (COST(r)) is defined to be $\lim_{B\to\infty} \text{COST}^r_B(x)/B$. An *∞-optimal rule* has expected asymptotic cost which is less than or equal to that of any other rule.

The previous section defined a rule for accessing B batches as a sequence of B decision functions. Though we could define a rule for accessing an infinite number of batches to be an infinite sequence of decision functions, we will restrict our attention to *uniform* rules which consist of one decision function, which is used to access all batches. It can, in fact, be proven that this restriction is not harmful; by only considering uniform rules we do find the ∞-optimal rule (over *all* rules). However, we omit the proof since it is long and technical. Besides, this fact is intuitively quite clear. Since the batches are independent and (unlike the case with a finite number of batches) each batch will have

an infinite number following, there is no intuitive justification for using different decision functions for accessing two different batches.

We note in passing that the sequences of head positions for a uniform rule is described by a Markov chain which is closed and irreducible, and hence it approaches a steady state distribution. If p_i is the steady state probability that the head is at position i, and c_i is the expected cost of accessing a batch from position i, then the expected asymptotic cost is given by $\sum_{i=1}^{n} p_i c_i$. The fact that this is equivalent to the original definition can be shown in a manner similar to that used in Lemma 10. Note also that an ∞-optimal rule is optimal in the original sense as described in § 2.

To develop an algorithm to determine an ∞-optimal rule for a given set of probabilities, we use the idea of "discrete dynamic programming" developed by Blackwell [11].

The situation in discrete dynamic programming is as follows: we are given a system with T states (labeled $1, \cdots, T$). At any time, the system is in exactly one state. At intervals of time, we are required to choose any one of a given set of A *actions* (labeled $1, \cdots, A$) to be performed. The *cost* of performing action a in state t is given by $c(t, a)$. (We will assume all costs are nonnegative.) After an action is performed, the system moves to a new state. The probability that action a will cause the system to move from state t to t' is denoted by $p(t \to t' | a)$. A *decision function*, f is a function from $\{1, \cdots, T\}$ into $\{1, \cdots, A\}$ where $f(t)$ specifies that action to be performed in state t. A *policy* $\pi = (f_1, f_2, \cdots,)$ is a sequence of decision functions where f_i is used to determine the ith decision.

The correspondence between this terminology and the original problem should be clear: a state corresponds to an ordered triple (x, l, r) where x is the current head position and l and r are the extremes of the current batch. There are only two actions: one for moving to the left extreme first, then sweeping across to the right and another for moving right first. (Note that when we apply an action in a given state, only the x of the next state is determined: l and r are chosen probabilistically.) The correspondence for costs and probabilities follow directly.

The procedure for calculating the ∞-optimal rule starts with an arbitrary rule (we choose the Nearest rule) and iteratively improves it. At each iteration we assume the current rule will be used to access all batches except the first and then calculate the best rule for accessing the first batch. A subsequent theorem shows that the asymptotic cost for this rule is less than or equal to that for the original rule. This new rule is then used on the next iteration, and we continue iterating until the new rule and the current rule are identical. The procedure for calculating the ∞-optimal rule is given below.

1. Make the Nearest rule the "current" rule.
2. Calculate $f_B(x)$ (the cost of accessing B batches with the head starting at position x) for the "current" rule. $f_B(x)$ is iteratively calculated for $B = 1, 2, \cdots,$ using the formula

$$f_0(x) = 0$$

$$f_B(x) = \sum_{l,r} \text{Prob}\,(L = l, R = r) \cdot \delta(x, l, r)$$

where

$$\delta(x, l, r) = \begin{cases} |x - l| + (r - l) + f_{B-1}(r) & \text{if the current rule goes to } l \text{ first with the head at } x \text{ and extremes } l \text{ and } r, \\ |x - r| + (r - l) + f_{B-1}(l) & \text{if it goes to } r \text{ first.} \end{cases}$$

The calculation continues until we reach a B such that $f_B(x) - f_{B-1}(x)$ is nearly constant with respect to x.

3. Calculate the "new" rule as follows:
   For each x, l and r, if

$$|x-r|-|x-l|+\lim_{B\to\infty}[f_B(l)-f_B(r)]<0$$

   the "new" rule should move to the right extreme first with head at x and extremes l and r. If this quantity is nonnegative, the rule should move left first. (Note that $\lim_{B\to\infty}[f_B(l)-f_B(r)]$ is closely approximated by $f_B(l)-f_B(r)$ for some large B.)
4. If the new rule is the same as the current rule, stop. This rule is the ∞-optimal rule. Otherwise set the "current rule" to be the "new rule" and go to step 2.

This procedure was used to find the ∞-optimal rule for several n and b. The results are shown in Table 4. The difference between the cost of the ∞-optimal rule and that of the Nearest rule is extremely small (compare Tables 3 and 4); in all cases it is less than

TABLE 4
Table of cost for the ∞-optimal rule assuming a uniform distribution

b \ n	100	200
5	80.713	161.452
10	90.394	180.839
15	93.496	187.067

0.05%. This, however, is to be expected for such large values of b. The extremes and the initial head position will almost always be near 1 and n, and an ∞-optimal rule will rarely have a chance to make a decision differing from that of the Nearest rule (moving to the farther extreme will rarely be "advantageous"). However, even if b is very small, the difference between the costs of the two rules is still very small. For $n = 100$, $b = 2$ and assuming a uniform distribution, the cost for the ∞-optimal rule is 54.971 as opposed to 55.036 for the Nearest rule, a difference of only .1%.

The correctness of this procedure is proven below. In these proofs, we need to distinguish between rules where one initially has lower cost, but asymptotically, both have the same cost. Therefore, we need the following definition:

If c_i is the cost incurred at time i, the *expected cost weighted by* β (for $\beta<1$) is $\sum_{i=1}^{\infty}\beta^{i-1}c_i$.

The following notations are convenient:

DEFINITION 1. If f is a decision function and π is a policy, then

$\bar{c}(f)$ is the column vector whose ith component is $c(i, f(i))$.

$\bar{p}(t, a)$ is the row vector whose ith component is $p(t\to i|a)$.

$Q(f)$ is the matrix whose (i, j)th entry is $p(i\to j|f(i))$ (state transition matrix).

$\bar{w}^{(\beta)}(\pi)$ is a column vector whose ith component is the expected cost weighted by β for using policy π if the system is initially in state i. (When the dependence on β is not important, the superscript will be dropped.)

$\bar{w}(f, \pi)$ will denote the vector of costs for using f followed by π.

We denote the ith component of a vector $\bar{x}$ by $\bar{x}_i$. We say $\bar{x}\leqq\bar{y}$ iff $\bar{x}_i\leqq\bar{y}_i$ for every i and $\bar{x}<\bar{y}$ iff $\bar{x}\leqq\bar{y}$ and there exists an i such that $\bar{x}_i<\bar{y}_i$.

Let $f^{(n)}$ denote the (finite) policy where f is successively applied n times and $f^{(\infty)}$ the policy $(f, f, \cdots)$. Note that in this notation, a policy π is *uniform* iff $\pi = f^{(\infty)}$ for some f.

Finally, if $\pi = (f_1, f_2, \cdots)$ let shift (π) denote the policy $(f_2, f_3, \cdots)$.

LEMMA 9. *Let $\pi = (f_1, f_2, \cdots)$ be a policy, then $\bar{w}(\pi) = \bar{c}(f_1) + \beta Q(f_1)\bar{w}$ (shift (π)).*

Proof. The $(n-1)$-step transition matrix is given by $Q(f_1)Q(f_2)\cdots Q(f_{n-1})$, and multiplying by $\bar{c}(f_n)$ will give the vector of cost for the nth decision. Hence

$$\bar{w}(\pi) = \sum_{n=1}^{\infty} \beta^{n-1} \cdot (Q(f_1)Q(f_2)\cdots Q(f_{n-1}))\bar{c}(f_n).$$

Then

$$\bar{w}(\text{shift }(\pi)) = \sum_{n=1}^{\infty} \beta^{n-1}(Q(f_2)Q(f_3)\cdots Q(f_n))\bar{c}(f_{n+1})$$

and the lemma clearly follows. □

THEOREM 7. *Let f be a decision function and π a policy, then*

(a) *If $\bar{w}(f, \pi) < \bar{w}(\pi)$ then $\bar{w}(f^{(\infty)}) < \bar{w}(\pi)$;*

(b) *If $\bar{w}(f, \pi) > \bar{w}(\pi)$ then $\bar{w}(f^{(\infty)}) > \bar{w}(\pi)$,*

and

(a)′ *and* (b)′: (a) *and* (b) *also hold if $<$ and $>$ are replaced by $\leqq$ and $\geqq$, respectively.*

Proof. We first prove by induction that

Claim: $\bar{w}(f^{(n+1)}, \pi) < \bar{w}(f^{(n)}, \pi)$ *for all* $n \geqq 1$.

For $n = 1$, use Lemma 9 to obtain:

$$\bar{w}(f^{(1)}, \pi) = \bar{c}(f) + \beta Q(f)\bar{w}(\pi), \qquad \bar{w}(f^{(2)}, \pi) = \bar{c}(f) + \beta Q(f)\bar{w}(f, \pi).$$

By hypothesis $\bar{w}(f, \pi) < \bar{w}(\pi)$, and since all entries in $Q(f)$ are nonnegative, we conclude that $Q(f)\bar{w}(f, \pi) < Q(f)\bar{w}(\pi)$ and hence $\bar{w}(f^{(2)}, \pi) < \bar{w}(f^{(1)}, \pi)$. The inductive step is proved in exactly the same manner, establishing the claim.

By applying Lemma 9 n times, we obtain

$$\bar{w}(f^{(n)}, \pi) = \sum_{i=1}^{n} \beta^{i-1} Q^i(f)\bar{c}(f) + \beta^n \bar{w}(\pi).$$

for any n. Since $\beta^{i-1}Q^i(f)\bar{c}(f)$ is nonnegative for all i, the summation is monotonically nondecreasing with respect to n. Hence it obviously has $\bar{w}(f^{(\infty)}) = \sum_{i=1}^{\infty} \beta^{i-1} Q(f)\bar{c}(f)$ as its limit as $n \to \infty$. Since $\beta < 1$, the second term approaches zero, and therefore, $\lim_{n\to\infty} \bar{w}(f^{(n)}, \pi) = \bar{w}(f^{(\infty)})$. From the claim, $\bar{w}(f^{(n)}, \pi)$ is decreasing with n. Hence $\bar{w}(f^{(\infty)}) < \bar{w}(f, \pi) < \bar{w}(\pi)$ proving (a). The proof of (b) is identical except that all inequalities are reversed. Finally (a)′ and (b)′ are proved by noting that if $\bar{w}(f, \pi) = \bar{w}(\pi)$ then $\bar{w}(f^{(\infty)}) = \bar{w}(\pi)$ (again, a simple proof by induction). □

LEMMA 10. *For any two policies $f^{(\infty)}$ and $g^{(\infty)}$ the following holds*:

$$\textit{If } \lim_{\beta\to 1} [\bar{w}_t^{(\beta)}(g^{(\infty)}) - \bar{w}_t^{(\beta)}(f^{(\infty)})] \geqq 0 \quad \textit{for all } t, \textit{ then} \quad \text{COST}(g^{(\infty)}) \geqq \text{COST}(f^{(\infty)}).$$

Proof. Consider any state t and let $c_i(f)$ and $c_i(g)$ be respectively the cost of applying $f^{(\infty)}$ and $g^{(\infty)}$ at time i, starting in state t. We have then $\lim_{\beta\to 1} [\sum_{i=1}^{\infty} \beta^{i-1}(c_i(g) - c_i(f))] \geqq 0$. Letting x_f and x_g be the steady state costs for $f^{(\infty)}$ and $g^{(\infty)}$, we decompose the cost into steady state and transient parts.

$$c_i(f) = x_f - \delta_i(f); \qquad c_i(g) = x_g - \delta_i(g).$$

Using the theory of Markov chains, it can be shown that the δ_i can be written in the following form:

$$\delta_i(f) = p_1(i)\lambda_1^i + p_2(i)\lambda_2^i + \cdots + p_k(i)\lambda_k^i,$$

where each p_j is a polynomial in i and each λ_j is a complex number with modulus strictly less than one. It is easily seen that $\sum_{i=1}^{\infty} \delta_i(f)$ and $\sum_{i=1}^{\infty} \delta_i(g)$ do not diverge to $\pm\infty$. Therefore

$$\lim_{\beta\to 1}\left[\sum_{i=1}^{\infty} \beta^{i-1}(c_i(g)-c_i(f))\right] = \lim_{\beta\to 1}\left[\sum_{i=1}^{\infty} \beta^{i-1}(x_g - x_f) - \sum_{i=1}^{\infty} \beta^{i-1}(\delta_i(g)-\delta_i(f))\right].$$

We now claim that $x_g \geqq x_f$. Otherwise $x_g < x_f$ and the first term in the above approaches $-\infty$, while the second is bounded, violating the assumption that the limit is nonnegative.

Now consider COST $(g^{(\infty)})$ − COST $(f^{(\infty)})$. By definition, this equals

$$\lim_{B\to\infty} \frac{\sum_{i=1}^{B} c_i(g)-c_i(f)}{B} = \lim_{B\to\infty} \frac{(\sum_{i=1}^{B} x_g - x_f) - (\sum_{i=1}^{B} \delta_i(g)-\delta_i(f))}{B}.$$

Since the second term is bounded, this equals $x_g - x_f$ which is nonnegative. Hence COST $(g^{(\infty)}) \geqq$ COST $(f^{(\infty)})$. □

We now define an algorithm for calculating the ∞-optimal rule. The strategy is one of iterative improvement; we begin with any policy $f^{(\infty)}$ and then calculate for each state the cost of performing a given action at the first time instant, assuming that $f^{(\infty)}$ will be used thereafter. This defines for each state t, a set Impr (t, f) of actions that are an improvement upon performing $f(t)$ at the first time instant. This gives rise to a policy $(g, f^{(\infty)})$ which is better than $f^{(\infty)}$. Theorem 6 guarantees that $g^{(\infty)}$ is better than $f^{(\infty)}$, and we therefore use $g^{(\infty)}$ for our next iteration. This process continues until Impr $(t, f) = \phi$ for every t. In this case, $f^{(\infty)}$ is the optimal rule.

The formula for calculating Impr (t, f) is given in the following theorem. To determine if action a is an improvement over $f(t)$, we calculate for every pair of states x and y the probability that a will send the system to state x and $f(t)$ will send it to state y, multiplied by "advantage" of starting in state x over state y when using policy $f^{(\infty)}$. This is summed over pairs and $c(t, a) - c(t, f(t))$ is added, to reflect the difference in cost between performing action a and $f(t)$. If the result is negative, a will be an improvement over $f(t)$.

THEOREM 8. *Let* Impr $(t, f) = \{a | [c(t, a) - c(t, f(t))] + [\sum_{x,y} p(t \to x | f(t)) \cdot p(t \to y | a)$ $\lim_{\beta\to 1} [\bar{w}_x^{(\beta)}(f^{(\infty)}) - \bar{w}_y^{(\beta)}(f^{(\infty)})]] < 0\}$; *then*

(1) *If* Impr $(t, f) = \phi$ *for all* t, *then* f *is optimal.*

(2) *If for some* t, Impr $(t, f) \neq \phi$, *then any* g *such that for all* t *either* $g(t) = f(t)$ *or* $g(t) \in$ Impr (t, f) *will have* COST $(g^{(\infty)}) \leqq$ COST $(f^{(\infty)})$.

Proof. We begin by establishing the following claim: *The condition in the theorem is equivalent to*

$$\lim_{\beta\to 1} [\bar{w}_t^{(\beta)}(g, f^{(\infty)}) - \bar{w}_t^{(\beta)}(f^{(\infty)})] < 0.$$

Proof of claim. Expanding each term by Lemma 9 gives

$$\begin{aligned}
&\lim_{\beta\to 1} [\bar{w}_t^{(\beta)}(g, f^{(\infty)}) - \bar{w}_t^{(\beta)}(f^{(\infty)})] \\
&\quad = \lim_{\beta\to 1} [c(t, a) + \beta\bar{p}(t, a)\bar{w}^{(\beta)}(f^{(\infty)}) \\
&\qquad - [c(t, f(t)) + \beta\bar{p}(t, f(t))\bar{w}^{(\beta)}(f^{(\infty)})]] \\
&\quad = \lim_{\beta\to 1} [[c(t, a) - c(t, f(t))] + \beta[\bar{p}(t, a) - \bar{p}(t, f(t))]\bar{w}^{(\beta)}(f^{(\infty)})].
\end{aligned}$$

Expanding the product of $\bar{p}$ and $\bar{w}$ gives that the above equals

$$\lim_{\beta\to 1}\left\{[c(t,a)-c(t,f(t))]+\beta\sum_x p(t\to x|f(t))\cdot\bar{w}_x^{(\beta)}(f^{(\infty)})+\beta\sum_y p(t\to y|a)\cdot\bar{w}_y^{(\beta)}(f^{(\infty)})\right\}.$$

Multiplying the first summation by $\sum_y p(t\to y|a)$ $(=1)$ and the second by $\sum_x p(t\to x|f(t))$, then combining the two sums gives that the above equals

$$\lim_{\beta\to 1}\left\{[c(t,a)-c(t,f(t))]+\beta\sum_{x,y} p(t\to x|f(t))p(t\to y|a)\cdot\bar{w}_x^{(\beta)}(f^{(\infty)})-\bar{w}_y^{(\beta)}(f^{(\infty)})]\right\}$$

proving the claim.

To prove part (1) of the theorem, consider any decision function, g, and suppose $\text{Impr}(t,f)=\phi$ for all t. This means that

$$\lim_{\beta\to 1}[\bar{w}_t^{(\beta)}(g,f^{(\infty)})-\bar{w}_t^{(\beta)}(f^{(\infty)})]\geqq 0 \quad \text{for all } t.$$

Hence there is a β_0 such that

$$\bar{w}_t^{(\beta)}(g,f^{(\infty)})-\bar{w}_t^{(\beta)}(f^{(\infty)})\geqq 0 \quad \text{for all } t \text{ and all } \beta \text{ such that } \beta_0<\beta<1.$$

Theorem 6 then implies that

$$\bar{w}_t^{(\beta)}(g^{(\infty)})-\bar{w}_t^{(\beta)}(f^{(\infty)})\geqq 0 \quad \text{for all } t \text{ and all } \beta \text{ such that } \beta_0<\beta<1.$$

Hence $\lim_{\beta\to 1}[\bar{w}_t^{(\beta)}(g^{(\infty)})-\bar{w}_t^{(\beta)}(f^{(\infty)})]\geqq 0$ for all t and Lemma 10 implies $\text{COST}(g^{(\infty)})\geqq\text{COST}(f^{(\infty)})$ for any g, and hence $f^{(\infty)}$ is optimal.

To prove part (2) of the theorem, consider the tth component of any g satisfying the restrictions of part (2). Either (a) $g(t)=f(t)$ in which case $\bar{w}_t^{(\beta)}(g,f^{(\infty)})=\bar{w}_t^{(\beta)}(f^{(\infty)})$ for all $\beta<1$ and by Lemma 9, $\bar{w}_t^{(\beta)}(g^{(\infty)})\leqq\bar{w}_t^{(\beta)}(f^{(\infty)})$ or (b) $g(t)\in\text{Impr}(t,f)$ in which case $\lim_{\beta\to 1}[\bar{w}_t^{(\beta)}(g,f^{(\infty)})-\bar{w}_t^{(\beta)}(f^{(\infty)})]<0$, implying in a manner like that in the proof of part (1) that $\bar{w}_t^{(\beta)}(g^{(\infty)})\leqq\bar{w}_t^{(\beta)}(f^{(\infty)})$. In either case, Lemma 10 proves that $\text{COST}(g^{(\infty)})\leqq\text{COST}(f^{(\infty)})$. □

Appendix.

Claim. There exists a distribution, n, and b such that $\text{COST}_N>\text{COST}_A$.

Proof. We consider an arrangement of $2x$ records, where the only records with nonzero probability are at positions 1, x, and $2x$. In addition, we add the restrictions that $p_1=p_x$. Therefore specifying one of the three nonzero probabilities (say p_1) will determine them all. Thus, there are three parameters that can be independently varied: x, b, and p_1. Later we will choose x and b to be large, and p_{2x} to be small.

Let us reason intuitively why the Nearest rule should have a higher cost than the Alternating rule for this arrangement. Since $p_1=p_x\approx\frac{1}{2}$ and b is large, the only batches with significant probability are $\{1,x\}$ (called a *nonfull* batch), and $\{1,x,2x\}$ (a *full* batch), with the former being much more probable. Since $\{1,x\}$ is nearly always the received batch, having the head at position $2x$ is significantly disadvantageous, and the Nearest rule has a much higher probability of being at position $2x$. The Alternating rule will be at $2x$ iff its head is currently at 1 and the batch $\{1,x,2x\}$ is received. The Nearest rule will also move its head to $2x$ in this case. In addition, it will move to $2x$ when its head is currently at x and $\{1,x,2x\}$ is received. (It moves to 1 first since it is closer, then ends at $2x$. The Alternating rule is at x, a *right* extreme, and hence will move to $2x$ first, ending at 1.) This argument is a simplification of the actual situation, but does motivate the chosen arrangement. The following formalizes this argument.

During the initial analysis, we assume the only possible batches are $\{1, x\}$ and $\{1, x, 2x\}$. At the conclusion, we show that other batches can be made so improbable that they can be safely ignored. Let ε be the probability of a full batch. That is,

$$\varepsilon = \text{Prob (batch is } \{1, x, 2x\} | \text{batch is } \{1, x\} \text{ or } \{1, x, 2x\})$$
$$= [1-(2p_1)^b-(1-p_1)^b+p_1^b]/[1-p_1^b-(1-p_1)^b],$$

where $0<p_1<\frac{1}{2}$. For any b, ε ranges continuously over (0, 1) for values of p_1 in $(0, \frac{1}{2})$. We now fix ε at any arbitrary value in (0, 1). Later, when b is chosen, p_1 must also be chosen to give the desired value for ε.

Consider the two rules acting on the given arrangement, with their heads starting at the same position. It is easy to see that their movement will be exactly the same until both heads are at position x, the Alternating rule is at a *right* extreme and a full batch appears. In this case, the Alternating rule moves right first, then left, and the Nearest rule does exactly the opposite. We now analyze the difference in the costs of the two rules for accessing the batches until the heads are again at the same position.

We define a Markov chain of 5 states with the following specifications:

If the heads are at the same position, the chain is in state 0. The other states are defined by:

State	Position for *Alternating Rule*	Position for *Nearest Rule*
1	1	$2x$
2	$2x$	1
3	1	x
4	x	1

Note that any other pair of head positions is impossible and that if the Alternating rule is at x, it is known to be a *right* extreme (resulting from the batch $\{1, x\}$), and hence this rule's actions are solely determined by the current head position.

It is mechanical to calculate the transition probabilities for each state by observing what each rule will do when its head is in a given position and a full/nonfull batch is received. A state-diagram for this chain is shown in Fig. 2. Note each transition has associated "cost" which gives the "advantage" of using the Alternating rule (the second number) in addition to its probability (the first number). More specifically, the second number gives the distance the Alternating rule moves in making the given transition,

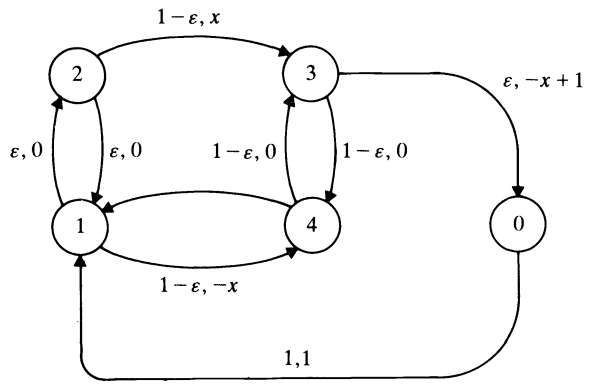


FIG. 2. *The Markov chain describing the head position of the two rules.*

minus that for the Nearest rule. The chain is initially in state 0 and remains there until both heads are at x and a full batch arrives. Let c_i be the expected cost incurred from the time the chain leaves state i until it first enters state 0 (meaning the heads are again together). We, then, are interested in c_0. The c_i satisfy the following system of equations:

$$c_0 = 1 + c_1,$$
$$c_1 = \varepsilon \cdot c_2 + (1-\varepsilon)(c_4 - x),$$
$$c_2 = \varepsilon \cdot c_1 + (1-\varepsilon)(c_3 + x),$$
$$c_3 = (1-\varepsilon) \cdot c_4 + \varepsilon(-x+1),$$
$$c_4 = (1-\varepsilon) \cdot c_3 + \varepsilon(c_1 + x).$$

These equations are easily seen: for example, consider c_1. With probability ε, the chain goes from c_1 to c_2. In this case, the total cost incurred in going from state 1 to state 0 equals c_2. With probability $1-\varepsilon$ the chain goes to state 4. Here the total cost will be $-x$ (the cost of the transition) plus c_4.

Solving the equations gives $c_0 = -2x(1-\varepsilon)^2 + 2$. Since ε has already been fixed, we can now choose x large enough so that c_0 will be negative.

We now consider the possibility of an "improbable" batch. The first necessity is to show that the probability of an improbable batch (p_I) can be made arbitrarily small by choosing b large enough. Since the "probable batches" are those containing records 1 and 3, or records 1 and 2, but not 3, an improbable batch must consist solely of record 1, or not contain record 1 at all. Hence $p_I = p_1^b + (1-p_1)^b$. Since $0 < \varepsilon < 1$, we have $0 < p_1 < 1$ and hence $p_I \to 0$ as $b \to \infty$.

A Markov chain that considers all batches as possible can be defined. (Since we are only concerned with the chain's existence and obvious properties, we will not actually specify it.) This chain would have a state for the heads being together, and one for each ordered pair of different positions. (In fact, there must be two states for each ordered pair where the Alternating rule is at position x: one for x being a left extreme and another for it being a right.) Now consider b as a parameter. As b is varied, p_1 is always chosen so that ε remains constant. Clearly this chain will be aperiodic and irreducible for any value of b, and its "cost" can be solved for as in the case of the simplified chain in Fig. 2, and the limit of the cost as $b \to \infty$ will be exactly the cost of the simplified chain (the transition probability of all the "improbable" arcs approaches zero).

Since the cost of the simplified chain is positive, b can be chosen large enough to make the cost of the actual chain positive, proving the claim. □

REFERENCES

[1] P. P. BERGMANS, *Minimizing expected travel time on geometrical patterns by optimal probability rearrangements*, Information and Control, 20 (1972), pp. 331–350.

[2] D. D. GROSSMAN AND H. F. SILVERMAN, *Placement of records on a secondary storage device to minimize access time*, J. Assoc. Comput. Mach., 20 (1973), pp. 429–438.

[3] V. R. PRATT, *An N log N algorithm to distribute N records optimally in a sequential access file*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 111–118.

[4] P. C. YUE AND C. K. WONG, *On the optimality of the probability ranking scheme in storage applications*, J. Assoc. Comput. Mach., 20 (1973), pp. 624–633.

[5] E. G. COFFMAN, JR. AND P. J. DENNING, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[6] R. M. KARP, A. C. MCKELLAR AND C. K. WONG, *Near-optimal solutions to a 2-dimensional placement problem*, this Journal, 4 (1975), pp. 271–286.

[7] P. C. YUE AND C. K. WONG, *Near-optimal heuristics for an assignment problem in mass storage*, Internat. J. Comput. Information Sci., 4 (1975), pp. 281–294.

[8] A. C. MCKELLAR AND C. K. WONG, *Dynamic placement of records in linear storage*, J. Assoc. Comput. Mach., 25, (1978), pp. 421–434.

[9] B. SHNEIDERMAN AND V. GOODMAN, *Batched searching of sequential and tree structured files*, ACM Trans. Database Systems, 1 (1976), pp. 268–275.

[10] P. B. HOEL, S. C. PORT AND C. J. STONE, *Introduction to Stochastic Processes*, Houghton Mifflin, Boston, 1972.

[11] D. BLACKWELL, *Discrete dynamic programming*, Ann. Math. Statist., 33 (1962), pp. 719–726.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0002 $01.00/0

POLYNOMIAL ALGORITHMS FOR COMPUTING THE SMITH AND HERMITE NORMAL FORMS OF AN INTEGER MATRIX*

RAVINDRAN KANNAN† AND ACHIM BACHEM‡

Abstract. Recently, Frumkin [9] pointed out that none of the well-known algorithms that transform an integer matrix into Smith [16] or Hermite [12] normal form is known to be polynomially bounded in its running time. In fact, Blankinship [3] noticed—as an empirical fact—that intermediate numbers may become quite large during standard calculations of these canonical forms. Here we present new algorithms in which both the number of algebraic operations and the number of (binary) digits of all intermediate numbers are bounded by polynomials in the length of the input data (assumed to be encoded in binary). These algorithms also find the multiplier-matrices K, U' and K' such that AK and $U'AK'$ are the Hermite and Smith normal forms of the given matrix A. This provides the first proof that multipliers with small enough entries exist.

Key words. Smith normal form, Hermite normal form, polynomial algorithm, Greatest Common Divisor, matrix-triangulation, matrix diagonalization, integer matrices, computational complexity

1. Introduction. Every nonsingular integer matrix can be transformed into a lower triangular integer matrix using elementary column operations. This was shown by Hermite ([12], Theorem 1 below). Smith ([16], Theorem 3 below) proved that any integer matrix can be diagonalized using elementary row and column operations. The Smith and Hermite normal forms play an important role in the study of rational matrices (calculating their characteristic equations), polynomial matrices (determining the latent roots), algebraic group theory (Newman [15]), system theory (Heymann and Thorpe [13]) and integer programming (Garfinkel and Nemhauser [10]).

Algorithms that compute Smith and Hermite normal forms of an integer matrix are given (among others) by Barnette and Pace [1], Bodewig [5], Bradley [7], Frumkin [9] and Hu [14]. The methods of Hu, Bodewig and Bradley are based on the explicit calculation of the greatest common divisor (GCD) and a set of multipliers whereas other algorithms ([1]) perform GCD calculations implicitly.

As Frumkin [9] pointed out, none of these algorithms is known to be polynomial. In transforming an integer matrix into Smith or Hermite normal form using known techniques, the number of digits of intermediate numbers does not appear to be bounded by a polynomial in the length of the input data as was pointed out by Blankinship [3], [4] and Frumkin [9].

To alleviate this problem, it has been suggested (Wolsey [17], Gorry, Northup and Shapiro [11], Hu [14], Frumkin [9]) that the Smith normal form of an integer matrix A can be computed modulo d (where d is the determinant of A). However this is not always valid as the following example shows. If we take

$$A = \begin{pmatrix} 5 & 26 \\ 2 & 11 \end{pmatrix}$$

* Received by the editors April 19, 1978 and in revised form September 21, 1978. This research was supported in part by N.S.F. Grant ENG-76-09936 and SFB 21 (DFG), Institut für Operations Research, Universität Bonn, Bonn, West Germany.

† Institute for Operations Research, University of Bonn and School of Operations Research, Cornell University, Ithaca, New York 14850.

‡ Institute for Operations Research, University of Bonn, Nassestrasse 2, D-5300 Bonn 1, West Germany.

then $\det(A) = d = 3$ and

$$\tilde{A} = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} \equiv A \pmod 3.$$

Comparing the GCD of all entries of A (which is 1) with the GCD of all entries of $\tilde{A}$ (which is 2) indicates that A and $\tilde{A}$ have different Smith normal forms.

Here we present polynomial algorithms for computing the Smith and Hermite normal forms. All intermediate numbers produced by these algorithms have at most a polynomial number of digits and the number of algebraic operations (additions and multiplications) performed is also bounded by a polynomial of the length of the input. Moreover the algorithms calculate the left and right multipliers (see description below) and thus prove that their entries are bounded in the number of digits by a polynomial in the length of the input. We must stress however that no exponential lower bounds have been proved on existing algorithms.

2. An algorithm for Hermite normal form. An integer square matrix with a determinant of $+1$ or -1 is called unimodular. Post-(pre-) multiplying an $(m \times n)$ matrix by a $(n \times n)((m \times m))$ unimodular matrix is equivalent to performing a series of column (row) operations consisting of (cf. Newman [15]):

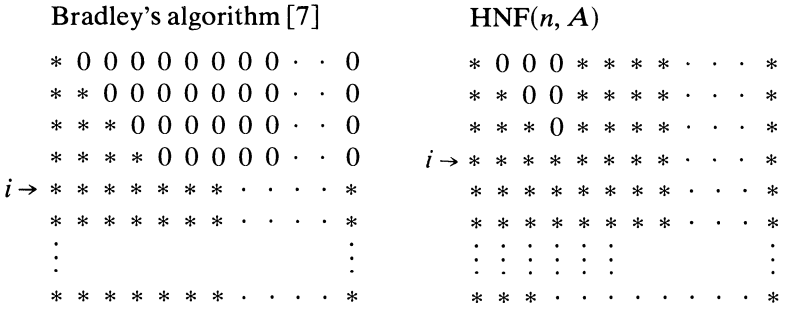
1. adding an integer multiple of one column (row) to another,
2. multiplying a column (row) by -1 and
3. interchanging two columns (rows).

These column (row) operations are termed elementary column (row) operations.

THEOREM 1. (Hermite [12]). *Given a nonsingular $n \times n$ integer matrix A, there exists a $n \times n$ unimodular matrix K such that AK is lower triangular with positive diagonal elements. Further, each off-diagonal element of AK is nonpositive and strictly less in absolute value than the diagonal element in its row. AK is called the Hermite normal form of A (abbreviated HNF).*

We now give an algorithmic procedure for calculating the Hermite normal form AK and the multiplier K. All currently known algorithms build up the Hermite normal form row by row, whereas the new algorithm HNF(n, A) (see description below) successively puts the principal minors of orders $1, \cdots, n$ (the submatrices consisting of the first i rows and i columns $1 \leqq i \leqq n$) into Hermite normal form. So at the ith step we have the following pictures. "0" stands for a zero entry of the matrix and "$*$" for an entry that may not be zero.

e.g.



ALGORITHM HNF(n, A): returns (HNF, K).

1. Initialize the multiplier K:

$$K = I \text{ (the } n \times n \text{ identity matrix).}$$

2. Permute the columns of A so that every principal minor of A is nonsingular; do the corresponding column operations on K:

Use a standard row reduction technique (Edmonds [8]) or see Appendix. If the matrix is found to be singular, the algorithm terminates here.

Note. This step need not be carried out, if suitable modifications are made in the rest of the algorithm. However, in the interest of simplicity, we have inserted this step here.

3. Initialize i which denotes the size of the principal minor that is already in *HNF*:

$$i = 1.$$

4. Put the $(i+1)\times(i+1)$ principal minor into HNF: (For any real matrix R, we denote by R_j the jth column of R and by $R_{i,j}$ (or R_{ij}) the element in the ith row and jth column of R).

If $i = n$ then terminate
else,
for $j = 1$ to i

4.1. *Calculate* $r = \mathrm{GCD}(A_{jj}, A_{j,(i+1)})$ and integers p and q such that $r = pA_{jj} + qA_{j,(i+1)}$ using a modified Euclidean algorithm (see Appendix)

4.2. Perform elementary column operations on A so that $A_{j,i+1}$ becomes zero:

$$D = \begin{pmatrix} p & -A_{j,(i+1)}/r \\ q & A_{j,j}/r \end{pmatrix}$$

Replace column j and $(i+1)$ of A by the two columns of the product

$$(A_j A_{i+1})D$$

Replace column j and $(i+1)$ of K by the two columns of the product

$$(K_j K_{i+1})D$$

4.3. If $j > 1$ then call REDUCE OFF DIAGONAL (j, A);

end

5. Call REDUCE OFF DIAGONAL $(i+1, A)$

6. Set $i = i+1$ and go to 4.

end HNF.

ALGORITHM REDUCE OFF DIAGONAL (k, A). (For any real real number y, $\lfloor y \rfloor$ and $\lceil y \rceil$ denote respectively the floor and ceiling of y.)

1. If $A_{kk} < 0$ set $A_k = -A_k$ and $K_k = -K_k$.

2. For $z = 1$ to $k-1$

$$\text{set } K_z = K_z - \lceil A_{kz}/A_{kk} \rceil K_k$$

$$\text{set } A_z = A_z - \lceil A_{kz}/A_{kk} \rceil A_k$$

end REDUCE OFF DIAGONAL.

We divide the proof that algorithm HNF is polynomial into two parts. First we show that intermediate numbers do not grow "too big". Using this result we prove that the number of algebraic operations (i.e. additions and multiplications) is bounded by a polynomial. The first part of the proof proceeds in 3 stages (Lemma 1–3). The simple fact stated below as a proposition is used many times in the proofs.

PROPOSITION 1. *For any real $n \times n$ matrix R,*

$$|\det R| \leqq (n \cdot \|R\|)^n$$

where $\|R\| =$ the maximum absolute value of any entry of R.

Proof. det R is the sum of $n!$ terms each of which is at most $\|R\|^n$ in absolute value. Since $n! \leqq n^n$, the proposition follows.

LEMMA 1. *For all $i = 1, \cdots, n$*

$$\|A^i\| \leqq n(n\|A^1\|)^{2n+1}, \tag{1}$$

$$\|K^i\| \leqq n(n\|A^1\|)^{2n} \tag{2}$$

$A^i(K^i)$ $(i = 1, \cdots, n)$ denote the matrix $A(K)$ after the $(i \times i)$ principal minor has been put into HNF *by the algorithm (Note that A^1* contains the original data.)

Proof. Clearly, A^i has been obtained from A^1 by elementary column operations on the first i columns of A alone. Thus if M^i and N^i denote the $(i \times i)$ principal minors of A^i and A^1 respectively, there is a unimodular $(i \times i)$ matrix $\tilde{K}^i$ such that

$$M^i = N^i \tilde{K}^i. \tag{3}$$

N^i is nonsingular, hence $\tilde{K}^i$ is unique and is given by $\tilde{K}^i = (N^i)^{-1} M^i$. Since $\|(N^i)^{-1}\|$ is at most the maximum absolute value of a cofactor of N^i we obtain (using Proposition 1):

$$\|\tilde{K}\| \leqq n\|(N^i)^{-1}\|\|M^i\| \leqq n(\|N^i\|n)^n\|M^i\|.$$

Because M^i is in Hermite normal form we obtain

$$\begin{aligned}\|M^i\| &\leqq |M^i_{11} \cdots M^i_{ii}| \\ &= |\det(M^i)| \\ &= |\det(N^i)| \\ &\leqq (i\|N^i\|)^i \leqq (i\|A^1\|)^i \leqq (n\|A^1\|)^n,\end{aligned}$$

hence

$$\|\tilde{K}^i\| \leqq n(n\|A^1\|)^{2n}.$$

Further, we have

$$A^i = A^1 \begin{bmatrix} \tilde{K}_i & 0 \cdots 0 \\ 0 & 1 \\ \vdots & \ddots \\ \vdots & \ddots \\ 0 & 1 \end{bmatrix}.$$

Thus $\|A^i\| \leqq n\|A^1\|\|\tilde{K}^i\| \leqq n(n\|A^1\|)^{2n+1}$. The proof of Lemma 2 below was inspired by a proof of Edmonds [8].

LEMMA 2. *At any stage of the execution of the algorithm,*

$$\|A\| \leqq 2^{3n} n^{(20n^3)} \|A^1\|^{12n^3}.$$

Proof. Let $A^{i,j}$ denote the matrix A after the "do loop" of Step 4 *has been executed* for the values i and j. First, we prove a bound on the entries in the $(i+1)$ st column of $A^{i,j}$ and using this, prove a bound on the rest of A at every stage. Let $d(i, j, k)$ be the determinant of the $(j+1) \times (j+1)$ submatrix of A^i consisting of columns 1 through j and column $(i+1)$ of A^i and rows 1 through j and row k of A^i. We show that there are integers r_j, $j = 1, \cdots, i$ such that

$$A^{i,j}_{k,i+1} = d(i, j, k)/r_j \qquad \text{for all } k \geqq j+1 \text{ and for all } j \leqq i. \tag{4}$$

For $j=1$ it is clear that (4) holds with $r_1=\mathrm{GCD}\,(A^i_{1,1}, A^i_{1,i+1})$. Suppose the statement is valid for $j=1,2,\cdots,p$. Let k be such that $n\geqq k\geqq p+2$. Denote $\alpha=A^{i,p}_{p+1,p+1}$, $\beta=A^{i,p}_{p+1,i+1}$, $\gamma=A^{i,p}_{k,p+1}$ and $\delta=A^{i,p}_{k,i+1}$ and let α', β', γ' and δ' be the corresponding elements of $A^{i,p+1}$.

A^i	$i+1$	$A^{i,p}$	$i+1$	$A^{i,p+1}$	$i+1$
	↓		↓		↓
	* 0 0 0 0 0 *		* 0 0 0 0 0 0		* 0 0 0 0 0 0 0
	* * 0 0 0 0 *		* * 0 0 0 0 0		* * 0 0 0 0 0 0
	* * * 0 0 0 *		* * * 0 0 0 0		* * * 0 0 0 0
$p+1\rightarrow$	* * * α 0 0 *		* * * α 0 0 β		* * * α' 0 0 0
	* * * * * 0 *		* * * * * 0 *		* * * * * 0 *
$k\rightarrow$	* * * γ * * *		* * * γ * * δ		* * * γ' * * δ'
	* * * * * * *		* * * * * * *		* * * * * * *

Then the matrices $\begin{pmatrix}\alpha & \beta\\ \gamma & \delta\end{pmatrix}$ and $\begin{pmatrix}\alpha' & 0\\ \gamma' & \delta'\end{pmatrix}$ have the same determinant and $\alpha'=\mathrm{GCD}(\alpha,\beta)$, hence using the induction hypothesis

$$\delta'=\frac{\alpha'\delta'}{\alpha'}=\frac{\alpha\delta-\beta\gamma}{\mathrm{GCD}(\alpha,\beta)}=\frac{\alpha d(i,p,k)-d(i,p,p+1)\gamma}{r_p\mathrm{GCD}(\alpha,\beta)}.$$

Since the last term above represents the expansion of $d(i,p+1,k)$ by the $(p+1)$th column of A^i we obtain

$$|\delta'|=|A^{i,p+1}_{k,i+1}|=|d(i,p+1,k)/r_{p+1}|,$$

where we set $r_{p+1}=r_p\mathrm{GCD}(\alpha,\beta)$. Thus, (4) holds. Using Lemma 1 we obtain

$$\begin{aligned}A^{i,j}_{k,i+1}&\leqq|d(i,j,k)|\\&\leqq(n\|A^i\|)^n\\&\leqq n^n(n(n\|A^1\|)^{2n+1})^n\\&\leqq n^{2n+2n^2+n}\|A^1\|^{2n^2+n}\\&\leqq n^{(5n^2)}\|A^1\|^{(3n^2)},\end{aligned}$$

which gives a bound on the $(i+1)$th column of $A^{i,j}$. For all other columns $1\leqq j\leqq i$ we conclude

$$|A^{i,j}_{k,j}|=|pA^i_{k,j}+qA^{i,j}_{k,i+1}|\leqq|p|(n^2\|A^1\|)^{2n+1}+|q|(n^{(5n^2)}\|A^1\|^{(3n^2)})$$

(cf. Step 4.2 of the algorithm). Both p and q are bounded by $\max\{|A^{i,j}_{j,j}|, |A^{i,j}_{j,i+1}|\}$ (see Appendix). Thus, $\|A\|\leqq 2n^{(10n^2)}\|A^1\|^{(6n^2)}=f$ (say). This does not still account for Step 4.3. Note that REDUCE OFF DIAGONAL (j, A) increases $\|A_z\|$ (the maximum absolute value of entry of column A_z) by at most a factor of $(1+\|A_j\|)$. Thus $\|A\|\leqq f(1+f)^n\leqq 2^n f^{2n}$. Hence Lemma 2 is proved.

LEMMA 3. *At any stage of the execution of the algorithm,*

$$\|K\|\leqq(2n\|A^1\|)^{0(n^4)}$$

Proof. We have already proved (in Lemma 1) that K^i has small enough entries. Each time the do loop of Step 4 is executed, $(-A_{j,i+1}/r)$ and $(A_{j,j}/r)$ are bounded by $2n^{(10n^2)}\|A^1\|^{(6n^2)}=d$ (cf. Lemma 2). By the modified Euclidean algorithm (see Appendix) p and q are bounded by d. Thus each execution of the do loop multiplies $\|K\|$ by at most 2d. There are at most n such multiplications before we arrive at K^{i+1} from

K^i. Thus $K \leqq (2d)^n n(n\|A^1\|)^{n^i}$ (by Lemma 1). Again to account for Step 4.3, an argument similar to the one in Lemma 2 shows that an exponent of $O(n^4)$ suffices.

THEOREM 2. *Algorithm* HNF *is polynomial.*

Proof of Theorem 2. Clearly the calculation in Steps 4.1, 4.2 and 4.3 of the algorithm are to be done at most n^2- times. The GCD calculation of step 4.1 is polynomial (see Appendix). Also, Step 2 of the algorithm is polynomial as shown again in the Appendix. Hence, the number of arithmetic operations as well as the number of digits of all intermediate numbers are polynomial and Theorem 2 is proved.

The algorithm HNF(n, A) is concerned with square nonsingular integer matrices. However an examination of the procedure will show that with obvious modification the algorithm works on arbitrary (m, n) integer matrix which has full row rank. In this case, the normal form is (H, 0), where 0 is a $(m \times (n-m))$ block of zeros and the $(m \times m)$ matrix H is in the form prescribed by Theorem 1. The algorithm for this (referred to later as $H(m, n, A)$) is as follows:

Use Step 2 of algorithm HNF(n, A) to permute the columns of A so that the first m principal minors are nonsingular. Call the new matrix A'. Use the steps 3–6 of the algorithm to transform the square nonsingular matrix $\begin{pmatrix} A' \\ 0I \end{pmatrix}$ into Hermite normal form (I is an $(n-m)\times(n-m)$ identity and 0 an $(n-m)\times m$ matrix of zeros). By Theorem 2 this is a polynomial algorithm. Return the first m rows of this HNF and all of K. Clearly using these algorithms we can transform any $(m \times n)$ integer matrix A with full column rank into a "left" Hermite normal form (LHNF) using row instead of column operations, i.e. A will be transformed into $\begin{pmatrix} H \\ 0 \end{pmatrix}$ where H is *upper triangular* with positive diagonal elements. Further, each off-diagonal element of H is nonpositive and strictly less in absolute value than the diagonal element in its column. Let us denote this algorithm by LHNF (m, n, A). LHNF (m, n, A) returns UA and U where U is an (m×m) unimodular matrix and UA the left Hermite normal form. Obviously LHNF (n, m, A) is still polynomial and an analogous result to Lemma 3 holds.

3. An algorithm for Smith normal form. THEOREM 3. (Smith [16]). *Given a nonsingular* $(n \times n)$ *integer matrix* A, *there exist* $(n \times n)$ *unimodular matrices* U, K *such that* $S(A) = UAK$ *is a diagonal matrix with positive diagonal elements* $d_1, \cdots, d_n$ *such that* d_i *divides* $d_{i+1} (i = 1, \cdots, n-1)$.

The typical step of the polynomial algorithm for the Smith normal form can be summarized in the following pictures:

$$
\begin{array}{lc}
 & \begin{matrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 \\ 0 & 0 & * & * & * & * \end{matrix} \\
i \rightarrow & \\
 & \begin{matrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \end{matrix}
\end{array}
\quad \text{LHNF on } (i+1)\text{th column} \quad
\begin{matrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \end{matrix}
\quad \text{HNF} \quad
\begin{matrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & * & * & * & 0 \\ 0 & 0 & * & * & * & * \end{matrix}
$$

$$
\begin{matrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * & * \end{matrix}
\quad \text{HNF} \quad \cdots \quad
\begin{matrix} * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & 0 & * & * & * \end{matrix}
\quad \text{LHNF} \quad (i+1)\text{th step} \rightarrow
$$

Note that this algorithm puts the bottom right $(n-i)\times(n-i)$ square matrix into HNF "frequently". Just after this is done each time, the product of the diagonal entries of A equals the absolute value of the determinant of the original matrix and thus each entry of A is at most this determinant in absolute value. Thus, the numbers are prevented from cumulatively building up.

This repeated use of HNF is the crucial difference between the algorithm presented here and the standard algorithms (e.g. [7] and [1]).

In the algorithm below we use the following notation. HNF $(n-i, n-i, A)$ is the procedure which puts the bottom-right-hand minor consisting of the last $(n-i)$ rows and columns into Hermite normal form. LHNF $(n-i, i+1, A)$ is the procedure which puts the submatrix of A consisting of the last $(n-i)$ rows and the column $i+1$ into left Hermite normal form.

ALGORITHM SNF(n, A): returns $(S(A), U, K)$.

1. Set $U = K = I$ the identity matrix of order n.
2. $i = -1$.
3. $i = i+1$. If $i = n$ stop.

At this stage the top-left $(i \times i)$ matrix is already in SNF and, if $i \geqq 1$, $A_{i,i}$ divides $A_{j,k}$ $\forall\, i \leqq j \leqq n$, $i \leqq k \leqq n$.

4. Call LHNF$(n-i, i+1, A)$ (returns A and U^*)

$$U = \begin{pmatrix} I & 0 \\ 0 & U^* \end{pmatrix} U \quad (I \text{ is an } i \times i \text{ identity matrix}).$$

5. Call HNF$(n-i, n-i, A)$ (returns A and K^*)

$$K = K \cdot \begin{pmatrix} I & 0 \\ 0 & K^* \end{pmatrix} \quad (I \text{ is an } i \times i \text{ identity matrix}).$$

6. If $A_{i+1,i+1}$ is not the only nonzero element in column $(i+1)$ go to 4.
7. If $A_{i+1,i+1}$ divides every element $A_{j,k}$ $i+1 \leqq j \leqq k$, $i+1 \leqq k \leqq n$, go to 3, otherwise $A_{i+1,i+1}$ does not divide $A_{j,k}$ (say). Add column k into column $i+1$ in A and K. Go to 4.

THEOREM 4. *The algorithm* SNF *is polynomial.*

Proof. Note that for a fixed i every time Step 4 or 5 is passed $A_{i+1,i+1}$ is replaced by a proper divisor of itself except the last and possibly the first times. Thus, for a fixed i, Steps 4 and 5 are executed at most $\log \|A(i)\| + 2$ times where $A(i)$ denotes the matrix A at the beginning of the ith iteration. Clearly $\|A(i)\| \leqq \max\{|\det(A(0))|, \|A(0)\|\}$, since either $i = 0$ or $A(i)$ is in Hermite normal form. Thus we count at most $2 + \log(\|A(i)\|) \leqq n \log(n\|A(0)\|) + 2$ calls of Step 4 and 5 for each value of i. We have therefore at most $n^2(\log n\|A(0)\|) + 2n$ passes of Steps 4 and 5. But here we use Hermite normal form algorithms and using Theorem 2 this proves Theorem 4.

THEOREM 5. *The unimodular matrices U and K which the algorithm* SNF *returns have entries whose number of digits are bounded by a polynomial.*

Proof. For every U^* and K^* in Steps 4 and 5 of the algorithm we have $\|U^*\|$, $\|K^*\| \leqq (c \cdot n \cdot \|A(0)\|)^{p(n)}$ for some polynomial $p(n)$ and constant c (cf. Lemma 3). By previous arguments, U and K at any stage are the product of at most $n \log((n\|A(0)\|)^n)$ of these matrices. Thus the theorem is proved.

Clearly analogous to LHNF, we can modify the algorithm SNF(n, A) in such a way that it works for arbitrary (n, m) integer matrices and remains polynomial. The details are rather elementary and are left to the reader.

We must remark that the algorithms in this paper are not meant to be efficient. The main concern has been simplicity of presentation. A computer code that includes several efficiency improvements is available from the authors.

Appendix.

LEMMA A.1. *If A is a nonsingular $n \times n$ matrix, then its columns can be permuted so that in the resulting matrix, all principal minors are nonsingular.*

Proof. The first row of A must have at least one nonzero element. Thus after a permutation of columns, if necessary, we can assume that A_{11} is nonzero. Assume for induction that the columns of A have been permuted to give a matrix A' in which the first i principal minors are nonsingular for some i, $1 \leqq i \leqq n-1$. Let A'' be the matrix consisting of all columns of A' and only rows 1 through $(i+1)$. A' is nonsingular implies that rank $(A'') = i+1$. Thus at least one of the columns say k, among $(i+1)$, $(i+2) \cdots, n$ of A'' cannot be expressed as a linear combination of the first i columns of A''. Swapping columns k and $(i+1)$ in A' leads to a matrix whose first $(i+1)$ principal minors are nonsingular. This completes the inductive proof.

The algorithm below uses essentially the idea of the proof.

Step 2 of Algorithm HNF.

for i = 1 to n
det = determinant of the $i \times i$ principal minor of A
$j = 1$
do while $(j \leqq n)$ **and** (det = 0)
$j = j+1$
det = determinant of the $i \times i$ submatrix of A consisting of the first i rows of A and columns 1 through $(i-1)$ and column j of A
end
if $j > n$, **terminate** /* A is singular */
Interchange columns j and i of A and of K.
end.

As remarked earlier, this subroutine is wasteful from the point of view of efficiency. However, it is polynomial, since determinant calculations can be done in polynomial time and at most n^2 determinants are to be computed.

GCD ALGORITHM. We use any standard algorithm (e.g. [2] or [6]) that for any two given numbers a and b, not both zeros, find p, q and r such that $r = GCD(a, b) = pa + qb$. We then execute the following step:

We assume $|a| \geqq |b|$, else swap a and b.

If $|q| > |a|$, then do: $p = p + \left\lfloor \frac{q}{a} \right\rfloor \cdot b$

$$q = q - \left\lfloor \frac{q}{a} \right\rfloor \cdot a$$

end.

Note that we still have $r = pa + qb$. But now, $|q| < |a|$. Thus

$$|qb| < |ab|,$$

$$pa + qb = r \Rightarrow |pa| < |ab| + |r|$$

$$\Rightarrow |p| < |b| + \left|\frac{r}{a}\right| \leqq |b| + 1.$$

Thus $|p|, |q| \leqq |a|$.

The Euclidean algorithm is well-known to be polynomial and certainly the step above is also polynomial.

Acknowledgment. The authors wish to thank Professor Les Trotter for reading the manuscript and suggesting several improvements. Thanks are due to the referee for several helpful suggestions.

REFERENCES

[1] S. BARNETTE AND I. S. PACE, *Efficient algorithms for linear system calculations; Part I—Smith form and common divisor of polynomial matrices*, Internat. J. of Systems Sci., 5 (1974), 403–411.

[2] W. A. BLANKINSHIP, *A new version of the Euclidean algorithm*, Amer. Math. Monthly, 70 (1963), 742–745.

[3] ———. Algorithm 287, *Matrix triangulation with integer arithmetic I* [F1], Comm. ACM, 9 (1966), p. 513.

[4] ———. *Algorithm* 288, *Solution of simultaneous linear diophantine equations* [F4], Comm. ACM, 9 (1966), p. 514.

[5] E. BODEWIG, *Matrix Calculus*, North Holland, Amsterdam, 1956.

[6] G. H. BRADLEY, *Algorithm and bound for the greatest common divisor of n* integers, Comm. ACM, 13 (1970), 433–436.

[7] ———. *Algorithms for Hermite and Smith normal matrices and linear diophantine equations*, Math. Comput., 25 (1971), pp. 897–907.

[8] J. EDMONDS, *Systems of distinct representatives and linear algebra*, J. Res. Nat. Bur. Standards, 71B (1967), pp. 241–245.

[9] M. A. FRUMKIN, *Polynomial time algorithms in the theory of linear diophantine equations*, M. Karpinski, ed., Fundamentals of Computation Theory (Springer, Lecture Notes in Computer Science 56, New York, 1977) pp. 386–392.

[10] R. GARFINKEL AND G. L. NEMHAUSER, *Integer Programming*, J. Wiley & Sons, New York, 1972.

[11] G. A. GORRY, W. D. NORTHUP AND J. F. SHAPIRO, *Computational experience with a group theoretic integer programming algorithm*, Math. Programming, 4 (1973), pp. 171–192.

[12] C. HERMITE, *Sur l'introduction des variables continues dans la théorie des nombres*, J. R. Angew. Math., 41 (1851), pp. 191–216.

[13] M. HEYMANN AND J. A. THORPE, *Transfer equivalence of linear dynamical systems*, SIAM J. Control Optimization, 8 (1970), pp. 19–40.

[14] T. C. HU, *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

[15] M. NEWMAN, *Integral Matrices*, Academic Press, New York, 1972.

[16] H. J. S. SMITH, *On systems of indeterminate equations and congruences*, Philos. Trans., 151 (1861), pp. 293–326.

[17] L. A. WOLSEY, *Group representational theory in integer programming*, Technical Report No. 41, Massachusetts Institute of Technology, Cambridge, MA, 1969.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0003 $01.00/0

TOWARDS A PRECISE CHARACTERIZATION OF THE COMPLEXITY OF UNIVERSAL AND NONUNIVERSAL TURING MACHINES*

LUTZ PRIESE†

Abstract. A computation universal Turing machine, U, with 2 states, 4 letters, 1 head and 1 two-dimensional tape is constructed by a translation of a universal register-machine language into networks over some simple abstract automata and, finally, of such networks into U. As there exists no universal Turing machine with 2 states, 2 letters, 1 head and 1 two-dimensional tape only the 2-state, 3-letter case for such machines remains an open problem. An immediate consequence of the construction of U is the existence of a universal 2-state, 2-letter, 2-head, 1 two-dimensional tape Turing machine, giving a first sharp boundary of the necessary complexity of universal Turing machines.

Key words. Turing-machines, two-dimensional tape, universal, nonuniversal, complexity-measures, register-machines, networks of abstract automata

Introduction. About fifteen years ago several attempts were made to find simple universal machines. Well known are the results of Watanabe (1961)—there exists a universal Turing machine with 8 states and 5 letters—and Minsky (1962)—there exists a universal Turing machine with 7 states and 4 letters. However, the gap in complexity between universal Turing machines and Turing machines which are not candidates for universal computation (for example, 2 states in the case of quadruple instructions cannot be sufficient, Fischer (1965)) was quite large and could not be closed. Some generalizations of Turing machines, such as Turing machines with multi-dimensional tape (or tapes) and multiple heads, led to some progress. Hooper (1963) showed the existence of universal Turing machines with 1 state, 2 letters, but also 4 heads, while Wagner (1973) proved Turing machines with 8 states and 4 letters operating on a two-dimensional tape, to be universal. The last result was improved by Kleine Büning and Ottmann (1977) to a universal two-dimensional Turing machine with 3 states and 6 letters and Kleine Büning (1977) proved in his Ph.D. thesis that 2 states and 5 letters or 10 states and 2 letters are also sufficient.

Kleine Büning and Ottmann proved their results by simulating so-called normed networks. This technique was first used by the author (Priese (1974)) to find some simple undecidable Thue-systems. We follow the principles of this technique in this paper also. Wagner's technique is very closely related to the common methods in 1-dimensional Turing machines.

Table 1 gives a review of the existing results. The mentioned complexity is explained in the final section of this paper.

Wagner (1973) proved also that for no dimension n can a 2-state, 2-letter, n-dimensional Turing machine be universal. This last result may offer a surprising chance to close the gap of complexity between universal and nonuniversal Turing machines: In this paper we prove the existence of a universal two-dimensional Turing machine U with 2 states and 4 letters. Thus only the 2-state, 3-letter case for 1-head two-dimensional Turing machines remains open. Any positive or negative characterization of this case results in a complete characterization of the complexity of universal two-dimensional Turing machines. In addition, U leads at once to a universal 2-state, 2-letter, 2-head, 1 two-dimensional tape Turing machine.

* Received by the editors February 10, 1978, and in revised form October 27, 1978.

† Fachgebiet Systemtheorie und Systemtechnik, Universität Dortmund, 4600 Dortmund 50, Federal Republic of Germany.

TABLE 1

Author	Year	States	Letters	Tape dimension	Heads	Complexity
Watanabe	1961	8	5	1	1	10^{76}
Minsky	1962	7	4	1	1	10^{49}
Hooper	1963	1	2	1	4	$6 \cdot 10^{49}$
Hooper	1963	2	3	1	2	$6 \cdot 10^{36}$
Wagner	1973	8	4	2	1	$3 \cdot 10^{67}$
Kleine Büning, Ottmann	1977	3	6	2	1	$5 \cdot 10^{44}$
Kleine Büning	1977	2	5	2	1	10^{16}
Kleine Büning	1977	10	2	2	1	10^{38}

In the sequel the notation "universal Turing machine" is understood as:

For different types of Turing machines there exist concepts of configurations ("instantaneous descriptions"). A computation $\mathscr{C}$ of a Turing machine is a finite or infinite sequence $\mathscr{C} = C_0, C_1, C_2, \cdots$ of configurations C_i such that C_{i+1} is a successor-configuration of C_i, for all C_i in $\mathscr{C}$. We shall give a precise definition in § 2. A Turing machine, M, is universal if there exists two Goedel-functions $\Phi: F_\mu \times \mathbb{N}_0 \to \mathbf{C}$ (F_μ denotes the class of all partially defined, recursive functions $f: \mathbb{N}_0 \to \mathbb{N}_0$, $\mathbb{N}_0$ is the set of all nonnegative integers, and $\mathbf{C}$ is the set of all configurations), and $\Psi: \mathbf{C} \to \mathbb{N}_0 \cup \{+\}$, such that for any $f \in F_\mu$, $n \in \mathbb{N}_0$ and computation $\mathscr{C} = C_0, C_1, C_2, \cdots$ of M that starts with $C_0 = \Phi(f, n)$, the following holds:

(a) $f(n)$ is undefined $\nleftrightarrow$ for all $m \in \mathbb{N}_0$: $\Psi(C_m) = +$, and

(b) $f(n)$ is defined $\nleftrightarrow$ there exists an index m_0 with

(i) for all $m < m_0$: $\Psi(C_m) = +$,

(ii) for all $m \geqq m_0$: $\Psi(C_m) = f(n)$.

A Goedel-function is a primitive recursive word-function. We have to use word-functions since a configuration is generally a word over some alphabet rather than an integer. A Goedel-function has to be primitive recursive since for the case of a recursive coding, the whole complexity of the computation of M can be hidden in the coding-functions Φ and Ψ; e.g. a trivial, nonoperating machine would become a universal machine.

$\Psi(C_m) = +$ means that C_m contains no result of the computation of M. Conditions (i) and (ii) ensure that, whenever a result has been found, this result remains unchanged if the computation of M continues.

We thus do not require that M stop once a result is computed. As finite computations are allowed, M may stop, but may also compute further configurations that code the same result.

The following construction of our universal Turing machine U implicitly describes both required Goedel-functions.

1. Networks of register machines. The universality of some simple machine, M, is often shown by implementing some universal calculus in M. Thus Minsky's 7-4-machine simulates tag-systems while Kleine Büning and Ottmann implement some networks of abstract automata with only a few simple primitives. Networks of register machines will be convenient for our purpose.

In this section we introduce in four steps the technical apparatus we need to construct the universal Turing machine U. This apparatus is discussed in some depth to simplify research on U in the second section.

Step 1. O_3-language register machines. Assume O_1 and O_2 to be the following register machine languages of possibly labeled instruction:

O_1: $\mathbf{a}_i$ add 1 in register i and **goto** the next instruction

$\mathbf{s}_i$ subtract 1 in register i and **goto** the next instruction

$\mathbf{t}_i(k)$ **if** register i is empty **goto** to instruction k, **else goto** the next instruction

O_2: $\mathbf{a}_i$

$\mathbf{t}'_i(k)$ **if** register i is empty **goto** instruction k, **else** subtract 1 in register i and **goto** the next instruction

It is well known (Sherpherdson–Sturgis (1963)) that both languages are universal programming languages for register machines. In O_1 the instruction $\mathbf{s}_i$ applied to an empty register leads to a stop by error. In addition, register machines involving only three registers are sufficient to compute any recursive function. If some standard codes, like $(x, y) \to 2^x 3^y$, are allowed for a representation of the input numbers two registers are also sufficient.

A programming language, O, is called universal, if for all recursive functions $f \in F_\mu$, $f: \mathbb{N}_0 \to \mathbb{N}_0$, there exists an O-program P_f such that P_f applied to the initial content n in register 1 and 0 elsewhere leads to a final content $f(n)$ in register 1 and 0 elsewhere. If $f(n)$ is undefined P_f, initialized as described, will never stop.

An alternative language is O_3:

O_3: $\mathbf{s}_i^+$ subtract 1 in register i and **goto** the next instruction

$\mathbf{t}_i^+(k)$ **if** register i contains a nonpositive integer add 1 in register i and **goto** instruction k, **else** add 1 in register i and **goto** the next instruction.

O_3 is a programming language that operates on register machines where the register may store arbitrary integers, including negative integers. As a consequence $\mathbf{s}_i^+$ is always applicable. This ability of storing negative integers also adds no more computational abilities, but reflects exactly the storing capacities of the universal Turing machine of § 2.

O_3 is also a universal programming language. Assume P to be a O_1-program. To obtain an equivalent O_3-program P' simply replace all instructions $\mathbf{a}_i$, $\mathbf{s}_i$, $\mathbf{t}_i(k)$ of P by the O_3-subprograms:

$\mathbf{a}_i$ $\quad \leftarrow \mathbf{t}_i^+(k)$ and label the next instruction with k, where k is a label that has not been used in P' and is not used in P.

$\mathbf{s}_i$ $\quad \leftarrow \mathbf{t}_i^+(k); \mathbf{s}_i^+; \mathbf{s}_i^+$ where k is a label of no instruction in P' (this simulates the stop-by-error of s_i in P if s_i is applied to a register i containing 0).

$\mathbf{t}_i(k) \leftarrow \mathbf{t}_i^+(k'); \mathbf{a}_i;$
$k': \mathbf{s}_i^+; \mathbf{s}_i^+; \mathbf{t}_i^+(k); \mathbf{s}_i^+$ where k' is a label that has not been used in P' and is not used in P.

We discuss this last translation: Suppose $\mathbf{t}_i(k)$ is applied in P while register i stores an integer n.

Case 1. $n > 0$:

Instruction in P'	Content of register i	
$\mathbf{t}_i^+(k')$	$n \leftarrow n+1$	(**goto** next instruction)
$\mathbf{a}_i$	$n+1 \leftarrow n+2$	
$k': \mathbf{s}_i^+$	$n+2 \leftarrow n+1$	
$\mathbf{s}_i^+$	$n+1 \leftarrow n$	
$\mathbf{t}_i^+(k)$	$n \leftarrow n+1$	
$\mathbf{s}_i^+$	$n+1 \leftarrow n$	

Case 2. $n = 0$:

$\mathbf{t}_i^+(k')$	$0 \leftarrow 1$ (**goto** instruction k')
$k' : \mathbf{s}_i^+$	$1 \leftarrow 0$
$\mathbf{s}_i^+$	$0 \leftarrow -1$
$\mathbf{t}_i^+(k)$	$-1 \leftarrow 0$ (**goto** instruction k).

Thus, P and P' compute the same functions $f: \mathbb{N}_0 \to \mathbb{N}_0$. Although P and P' are applied only to registers with an initial nonnegative content, P' stores in some intermittent steps negative integers also.

Step 2. *Automata networks.* The infinite automaton REG $= (I, O, S, T)$ with inputs $I = \{test, sub\}$, outputs $O = \{>0, \leqq 0, sub'\}$, all integers $\mathbb{Z}$ as states, is given by the transitions $T \subseteq (I \times S) \times (O \times S)$:

$$\begin{aligned} T: \; & sub, n \to sub', n-1, n \in \mathbb{Z} \\ & test, n \to >0, n+1, n > 0 \\ & test, n \to \leqq 0, n+1, n \leqq 0 \end{aligned}$$

where a transition $((i, s), (o, s'))$ is written as $i, s \to o, s'$. REG will be graphically represented as shown in Fig. 1.

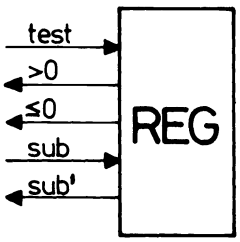


FIG. 1

We have defined the infinite register automaton REG in such a fashion that REG behaves just like a register of a register machine with an O_3-program. By using some standard techniques—like the methods of representing regular events with finite automata—any O_3-program P of a n-register machine can easily be translated into a finite, initial automaton, M_p, such that the automata network N_p of Fig. 2, with n copies of the REG-automaton, simulates P in the following sense:

The program P applied to initial content m_i in register i stops with m_i' in register i if and only if a signal, entering M_P in its initial state via input line (start), while all REGi are in the states m_i, respectively, reaches the output line (stop) of M_p while all REGi are in the states m_i', respectively. P applied to m_i never stops if and only if a signal, applied to N_p as just described, never reaches the output line (stop).

Step 3. *Universal primitives for automata-networks.* The theory of *normed networks* of D. Rödding leads to a further decomposition of M_p and simplification of REG. K and P are the automata

$\mathrm{K} = (\{1, 2\}, \{3\}, \{a\}, T)$, with the transitions
T: $1, a \to 3, a$ and $2, a \to 3, a$.
$\mathrm{P} = (\{t, c\}, \{t^u, t^d, c^u, c^d\}, \{up, down\}, T)$ with the transitions
T: $t, up \to t^u, up$
$t, down \to t^d, down$
$c, up \to c^u, down$
$c, down \to c^d, up$.

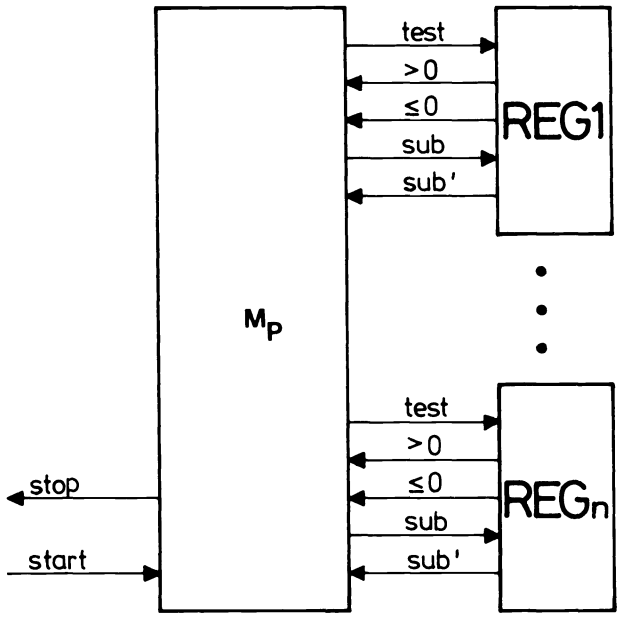


FIG. 2

They are represented by the diagram of Fig. 3.

K acts as a union of wires that is needed for topological reasons. P is a switch (or "flip-flop") with a test-input, t, that tests the state of P, and with a change-input, c, that changes and tests the state of P.

A *normed network* over some automata $A_1, \cdots, A_n$ consists of copies of the automata $A_1, \cdots, A_n$ that are connected according to the composition rule that any output of a copy may be identified with at most one input of a copy and that no input shall be identified with more than one output. The automata $A_1, \cdots, A_n$ are regarded as sequential machines with transitions of the described type $I \times S \to O \times S$. We always refer to the model of a single signal, passing through a normed network and changing the states of the components as described by the transitions. Formal definitions of normed networks with heuristic discussion and results can be found in Ottmann (1973), Priese (1978), Priese and Rödding (1974) and Rödding and Rödding (1978). The automata K and E, where E results from P by identifying both outputs c^u and c^d, are

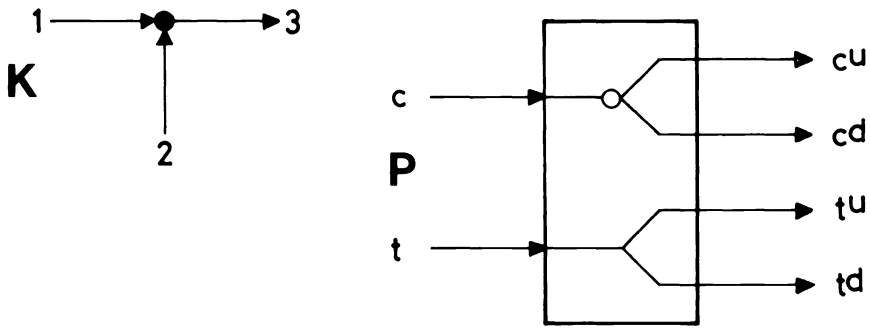


FIG. 3

universal primitives for normed networks: Any finite automaton can be simulated (in the sense of Hartmanis, Stearns (1966), e.g.) by a normed network over the two primitives K and E; see Priese, Rödding (1979). As an immediate consequence K and P also form a universal base of primitives. In particular, there exists a normed network that simulates M_p and consists only of K and P primitives.

We name the automaton that is obtained from REG by identifying both outputs $\leqq 0$ and *sub'* by REG^+. REG^+ is represented by the diagram of Fig. 4.

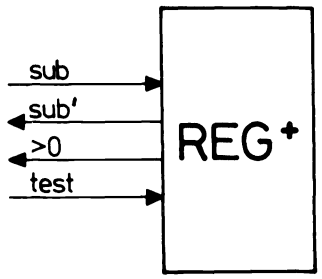


FIG. 4

It is quite a simple exercise to prove how to simulate REG with the help of REG^+. However, as such a proof cannot be found in the literature of normed networks, and in order to give an idea of how to operate with normed networks, we will outline a proof.

REG is simulated by the normed network, N_R, of Fig. 5. N_R shall be in a state n and all automata P_1, P_2 and P_3 are in the same state *up* or *down*. A signal that enters N_R via the input test reaches the input test of REG^+ while all P_i are in the state *down*, as this signal has tested the state of P_1 and eventually changed the states of all P_i to *down*. A signal that enters N_R via its input *sub* reaches the input *sub* of REG^+ while all P_i are in the state *up*. Thus, if a signal leaves REG^+ via its output *sub'* a further test of the state of P_3 tells whether the output was reached as a consequence of an input-signal on the *sub*-input (P_3 is in the state *up*) or of an input-signal on the *test*-input while REG^+ was in a state $\leqq 0$.

Combining the results of Steps 2 and 3 any O_3-program of a n-register machine can be simulated by a normed network with the primitives K and P and n copies of REG^+.

Step 4. *Topological standardizations of normed networks.* We will restrict the representations of normed networks over K and P to some standardized diagrams. Those diagrams consist of some rather complicated patterns of "dominoes." We operate with the elementary standardized diagrams of Table 2. These components involve one cell (components *In*, $r-d$, $u-d$, $u-l$, $l-r$, $r-l$), two cells (components $l-d-l$, $l-u-l$, K), three cells (components $r-d-r$, $r-u-r$), five cells (component *Out*), six cells (component $l-u-r$), fifteen cells (component *CR*), and

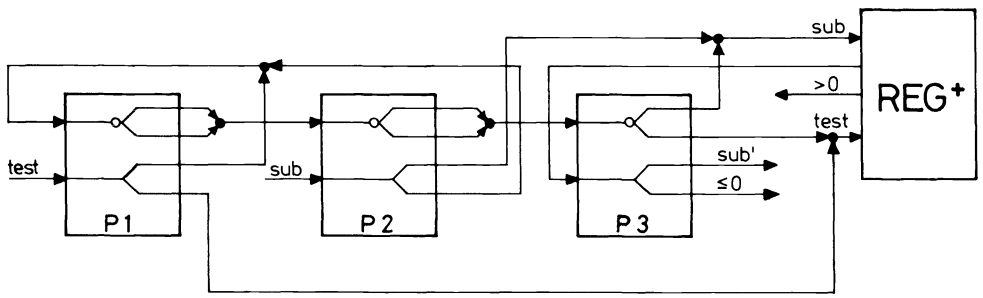
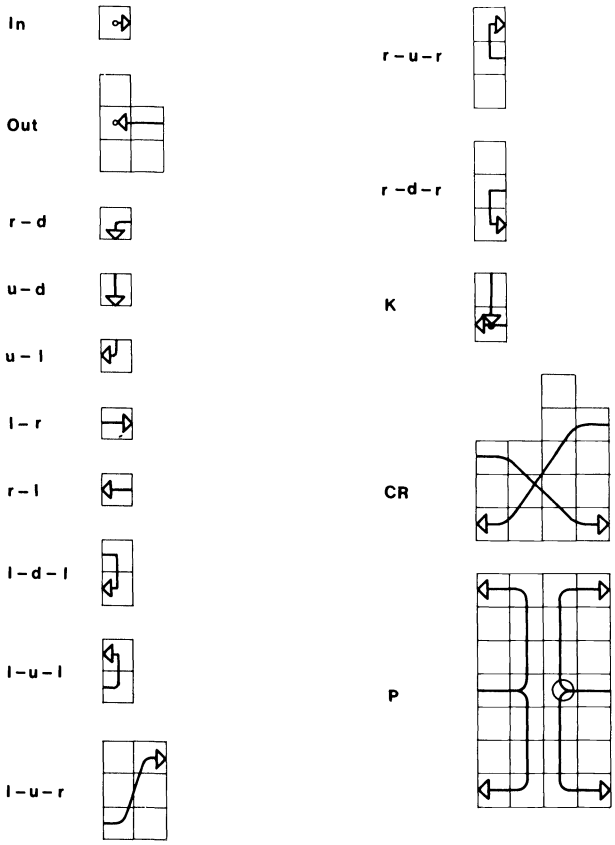


FIG. 5

twenty-eight cells (component P). In the following the term *s-diagram* refers to a (partial) covering of the Gaussian plane $\mathbb{N}_0 \times \mathbb{N}_0$ by standardized diagrams according to the rule that all arrow-inputs and arrow-outputs shall be connected with arrow-outputs and arrow-inputs, respectively, of the neighboring cells. No overlapping of those standardized diagrams is allowed.

TABLE 2



Arrows thus have to start with an *In*-component and end with an *Out*-component. The components K and P are alternative representations of the automata K and P. Any s-diagram defines uniquely a normed network over K and P. The opposite is also true: Any normed network over K and P can be represented by an s-diagram. Figs. 6 and 7 give the idea of how to find a representing s-diagram. One starts with a (nonstandardized) diagram representation of a given normed network and applies some "topological transformations" to obtain an s-diagram.

The first two lines of Fig. 6 show how to get horizontal arcs in an s-diagram. The third line gives an idea of how to implement a vertical "*up-to-down*" arc. It should be noted that we implement some "bented" vertical arcs, due to the restrictions in the elementary diagrams of Table 2, that start from a horizontal line and lead to a horizontal line again. We have shown an implementation of a "*from left-(go down)*-go right*" arc. All remaining versions of line 3 are easily implemented in analogy.

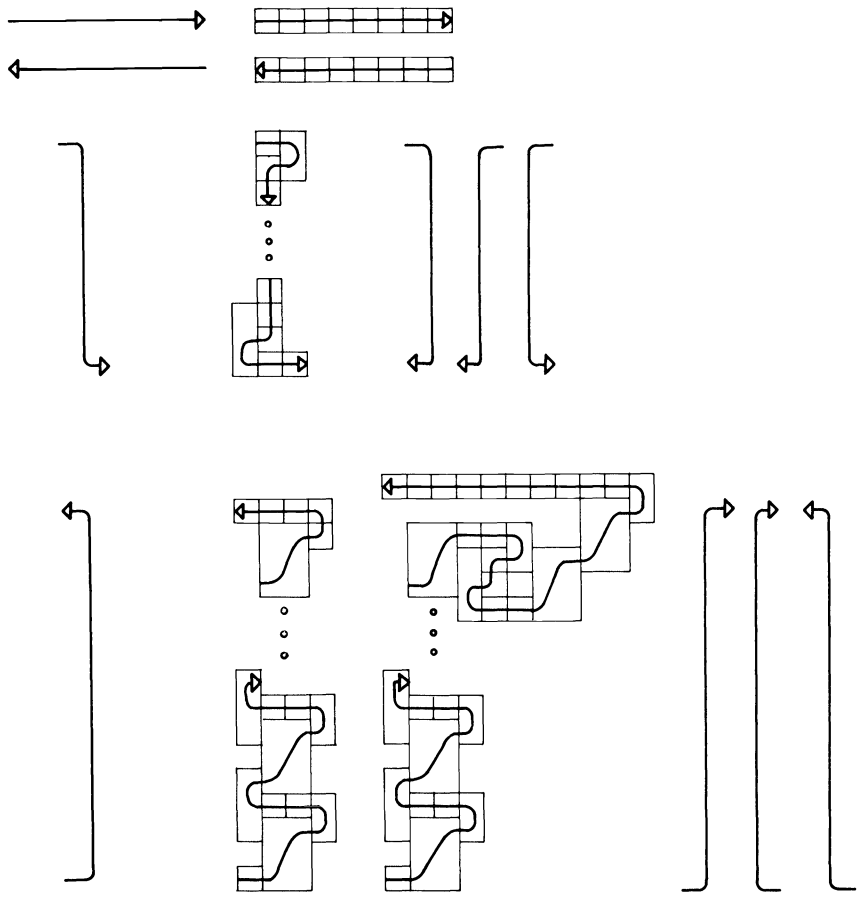


FIG. 6

Line 4 shows an implementation of a vertical "*down-to-up*" arc with a second implementation of a different length modulo four.

Figure 7 shows how to implement different versions of crosses with the help of the cross *CR* and bented arcs. The same holds for different versions of the *K*-element. The last line of Fig. 7 "separates" the inputs and outputs of the *K*- and *P*-diagrams in such a way that further connections with arcs can be done. These examples should suffice to give a clear idea of how to implement any normed network over K and P into an *s*-diagram.

An *s*-diagram with exactly one occurring *In*- and *Out*-diagram is called a 1-1-diagram, and a normed network with exactly one input and output is called a 1-1-net. By Step 2, 1-1-nets over K, P and REG^+ form a universal logic. This allows us to turn our attention to 1-1-diagrams.

2. Two-dimensional Turing machines. A (two-dimensional) Turing machine, M, is a tuple $M = (S, L, I)$ of a finite set S of states together with a distinguished (initial) state, s_1, a finite set L of letters together with a distinguished (blank) letter, B, and a functional relation $I \subseteq (S \times L) \times (L \times S \times \{u, d, r, l\})$. An instruction $(s, l \to l', s', m) \in I$, $m \in$

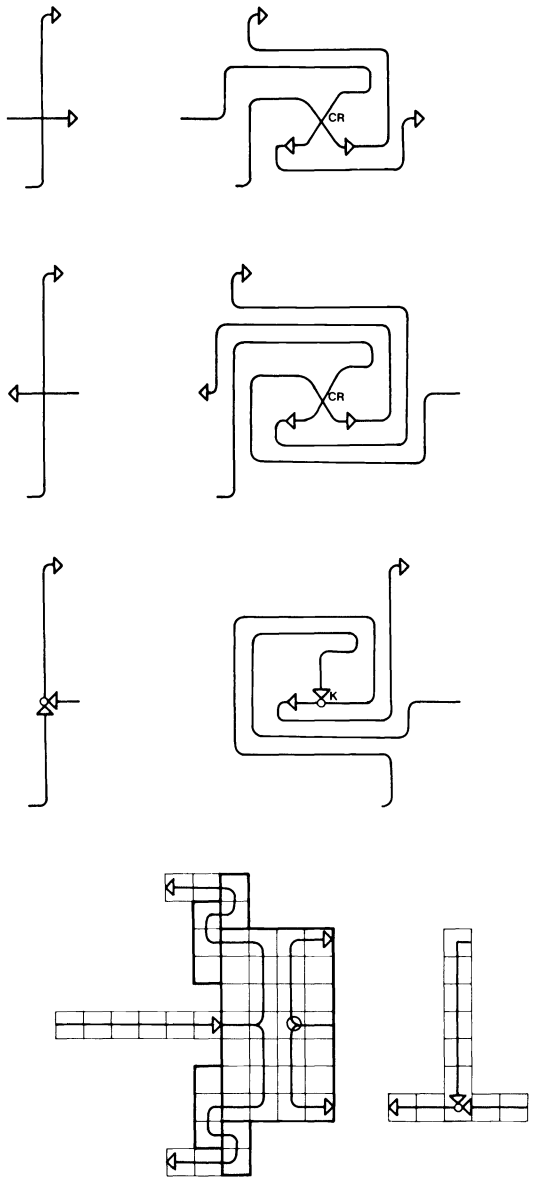


FIG. 7

{u, d, r, l}, has the interpretation: If M is in the state s and the head of M reads a letter l, M replaces this letter l by l', moves the head one cell to m, where r abbreviates right, l left, u up, and d down, and switches to state s'. M may operate on the plane $\mathbb{Z} \times \mathbb{Z}$ or the first quadrant $\mathbb{N}_0 \times \mathbb{N}_0$; here $\mathbb{Z}$ denotes the set of all integers (and $\mathbb{N}_0$ of all nonnegative integers). An application of an instruction that forces the head of M to leave the first quadrant would lead to a stop by error in the second case. Our main theorem will hold for both cases with some minor differences.

We have to state some definitions on two-dimensional Turing machines.

Let us regard the plane F, $F = \mathbb{N}_0^2$ or $\mathbb{Z}^2$. A (plane-)configuration C is a mapping $C: F \to (S \cup \{*\}) \times L$ with the properties:

(i) $\#\{c \in F;\ C(c) \notin (S \cup \{*\}) \times \{B\}\} < \infty$,

(ii) $\#\{c \in F;\ C(c) \in S \times L\} = 1$.

C denotes the set of all configurations. By condition (i) we regard only finite configurations, where only finitely many cells carry a letter different from the blank letter B. By condition (ii) exactly one cell carries also a state of S. This cell denotes the position of the Turing machine head and the state of the Turing machine. A configuration thus describes uniquely a finite pattern of letters on the plane, the position of the head and the state of the Turing machine. A computation $\mathscr{C}$ of a Turing machine, M, is a finite or infinite sequence $\mathscr{C} = C_0, C_1, C_2, \cdots$ of configurations C_i where C_0 is an initial configuration and all C_{i+1}-configurations are successor-configurations of the C_i-configurations. An initial configuration is a configuration with the head positioned on the origin of the plane in the initial state s_1, i.e.: $C_0((0, 0)) \in \{s_1\} \times L$. A configuration C' is a successor-configuration of a configuration C if the following holds:

Assume (x_0, y_0) to be the position of the head, i.e.: $C(x_0, y_0)) = (s, l)$ for some $s \in S$ and $l \in L$ $(s \neq *)$.

If there exists no instruction of M with the left-hand side (s, l), M stops on C, i.e., $C' = C$ holds.

If there exists an instruction $s, l \to l', s', m$ of M, then there holds for C':

(i) $C'((x_0, y_0)) = (*, l')$,

(ii) $[C((x_0 + \delta_1, y_0 + \delta_2)) = (*, l'')] > C'((x_0 + \delta_1, y_0 + \delta_2)) = (s', l'')$ holds for

$$\begin{aligned}
\delta_1 &= 0, & \delta_2 &= 1 && \text{if } m = u,\\
\delta_1 &= 0, & \delta_2 &= -1 && \text{if } m = d,\\
\delta_1 &= 1, & \delta_2 &= 0 && \text{if } m = r,\\
\delta_1 &= -1, & \delta_2 &= 0 && \text{if } m = l,
\end{aligned}$$

(iii) $C'(c) = C(c)$ for all remaining cells c.

This is a straightforward generalization of configurations of one-dimensional Turing machines to the case of a two-dimensional tape.

A Turing machine, M, simulates a 1-1-net, N, (with the input i and output o), if there exists two Goedel-functions $\Phi: \{i\} \times S_N \to \mathbf{C}$, $\Psi: \mathbf{C} \to (\{o\} \times S_N) \cup \{+\}$, such that for any $s, s' \in S_N$:

(a) $i, s \to_N o, s'$ holds if and only if a computation $\mathscr{C} = C_0, C_1, C_2, \cdots$ of M, starting with $C_0 = \Phi(i, s)$, possess an index m_0 with

(i) for all $m < m_0$: $\Psi(C_m) = +$,

(ii) for all $m \geqq m_0$: $\Psi(C_m) = (o, s')$.

(b) there exists no o, s' such that $i, s \to_N o, s'$ holds if and only if for a computation $\mathscr{C} = C_0, C_1, C_2, \cdots$ of M, starting with $C_0 = \Phi(i, s)$, $\Psi(C_m) = +$ holds for all $m \in \mathbb{N}_0$.

It is well known that there exist universal register machines (in exactly the same sense of universality as we defined for Turing machines. A register machine configuration describes the number of the applied instruction and the content of all registers). If we simulate such a universal register machine, R, by a 1-1-net, N_R, over K, P and REG^+, and, finally, N_R by a Turing machine, M_R, according to the above definition, M_R is also a universal machine.

The universal Turing machine U. Define U to be the 2-state, 4-letter, two-dimensional tape Turing machine $U = (\{1, 2\}, \{B, C, D, U\}, I)$ with the instructions:

$$
\begin{aligned}
I: \quad & 1, B \to C, 2, u \\
& 2, B \to B, 2, d \\
& 1, C \to C, 1, r \\
& 2, C \to C, 2, l \\
& 1, D \to D, 2, d \\
& 2, D \to U, 1, d \\
& 1, U \to U, 2, u \\
& 2, U \to D, 1, u.
\end{aligned}
$$

B is the blank letter and 1 the initial state. To ease an understanding of U the letters should be read as: Blank (B), continue (C), go Down (D), go Up (U), and the states 1 and 2 should be read as "move right" and "move left." Figure 8 gives a kind of letter-diagram for U, using *right* and *left* for the states 1 and 2. These mnemonic names refer to the activities of U.

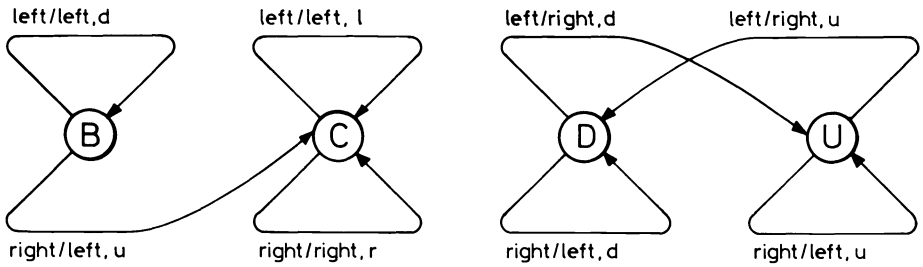


FIG. 8

THEOREM. U *simulates any* 1-1*-net over* K, P *and* REG$^+$.

Proof. Let N be a 1-1-net over K, P and REG$^+$. The following construction attaches to any state of N a finite sub-configuration for U. We will not state the Goedel-functions explicitly but the constructions will elucidate them with all the required properties.

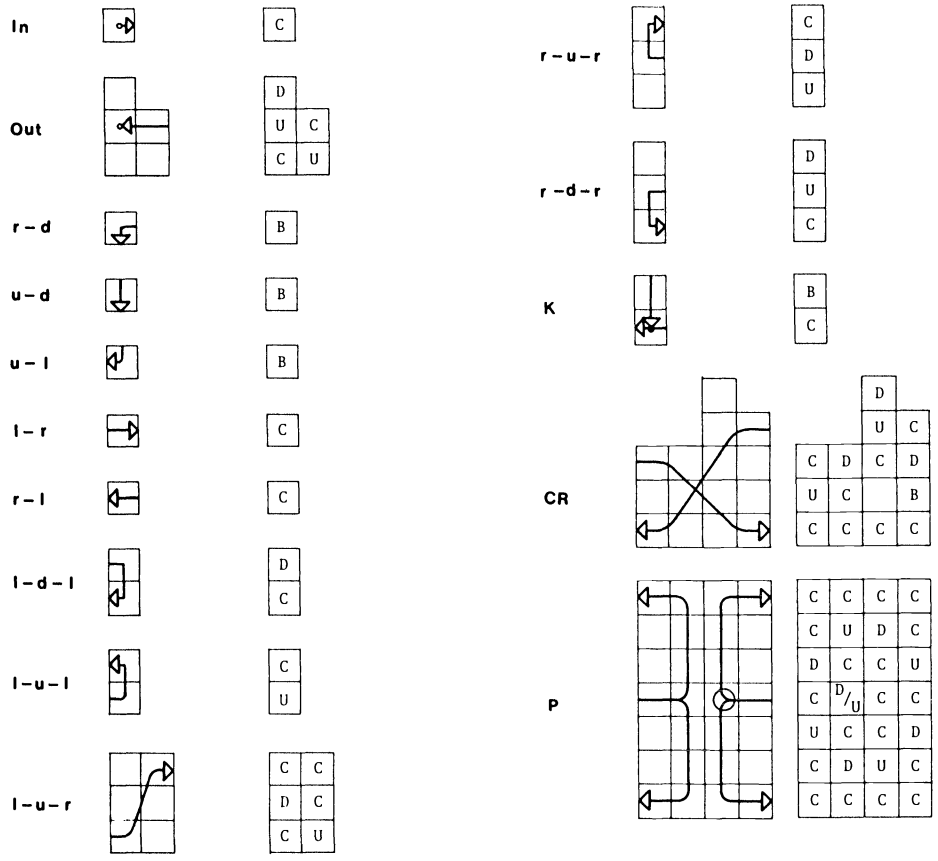
By Step 4, N is represented by an s-diagram of the standardized components plus diagrams for the necessary registers REG$^+$.

Implementation of the s-diagrams. All s-diagrams of Table 2 are implemented into finite sub-configurations as shown in Table 3. The two letters D/U in the P-implementation mean that this cell stores the letter D if P is in the state *down*, and U in the state *up*.

Figure 9 gives an example of how U operates on these sub-configurations. It is shown how a signal, that enters P in the state *up* via its input t, passes P to the correct output t^u. A number 1 or 2 in a cell describes the position of the head and state of the machine. If the head reads a letter C(for Continue) in state 1 ("move right") it moves right, leaving the letter and state unchanged. The same holds for the state 2 ("move left"). A letter U (D), always forces the head to move up (down) by changing the horizontal direction (state), while the letter might be replaced by a D (or U, respectively) if the head reads this letter in state 2. Thus, whenever the head is positioned on a cell, implementing an arc "*to the right*" of the s-diagrams, the state has to be 1. It should be obvious, that the given implementation in Table 3 of the s-diagrams of Table 2 ensures a correct movement of the head, which thus reflects the movement of a signal in

a normed network along the paths of its s-diagram representation. The changes of the states of the P-automaton are also correctly simulated, as one may easily test, analogous to Fig. 9. We always dropped the sign $*$ in the cells the head is not looking at. All cells that receive no letters in these figures are of no interest for the intended behavior (and may carry any letter).

TABLE 3



For a correct initialization of the first configuration C_0 of a computation the head is positioned in state 1 on the *In*-implementation, this being the origin (0, 0) with the letter C.

A configuration codes the result of a U-computation if the head has entered (and remains captured in) the one *Out*-implementation, or if U stops. Note that U can only stop by a stop-by-error in the case that U operates on the plane $F = \mathbb{N}_0^2$ and the head tries to leave F. U operating on $F = \mathbb{Z}^2$ allows for no stop!

This brief comment should elucidate the principle of U simulating any normed network over K and P.

An implementation of a REG$^+$-component is shown in Fig. 10.

This implementation requires an infinite number of cells in the blank state, B, of breadth 3 and a one-sided infinite length. As an implementation of a normed network over K and P involves only a finite number of cells, say in a rectangle R with the left

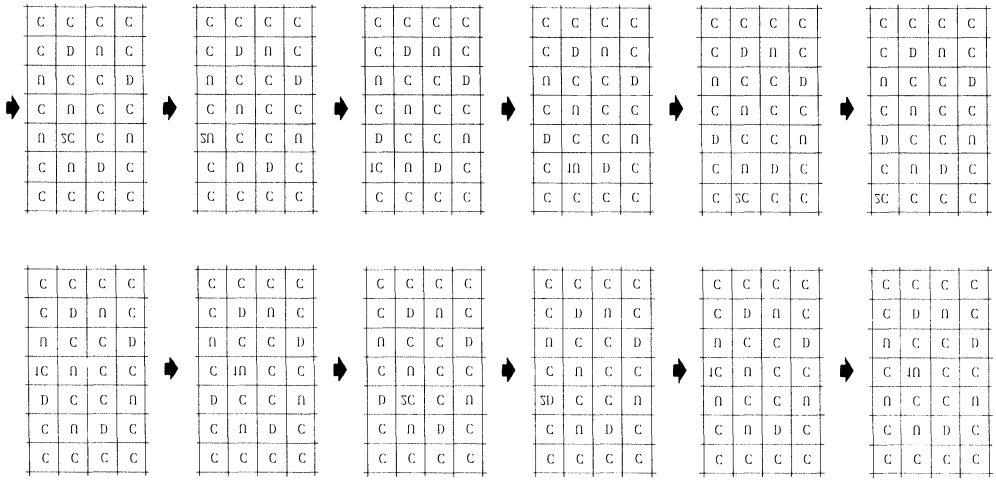


FIG. 9

lower corner (0, 0), such REG^+-implementations may be placed on the right periphery of R.

The difference m of the length between the lower and upper C-string represents the state $m+1$ of REG^+. For example, REG^+ in state 0 receives in Fig. 11 the part with the difference $m=-1$:

. . . B B B B B B B B

. . . C C C C C B B B

. . . C C C C B B B B

The Turing machine head, entering the implementation $\mathbf{R}^+$ of REG^+ via *test* in state 1 changes one letter B to C, thus alters the difference m to $m+1$, and reaches the

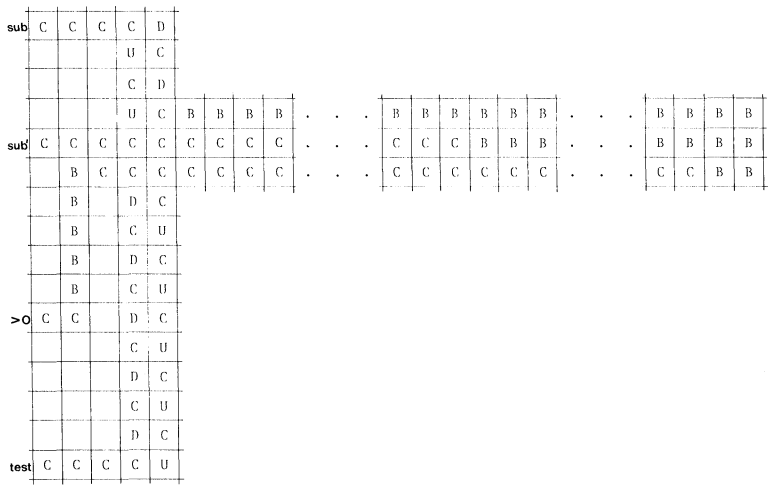


FIG. 10

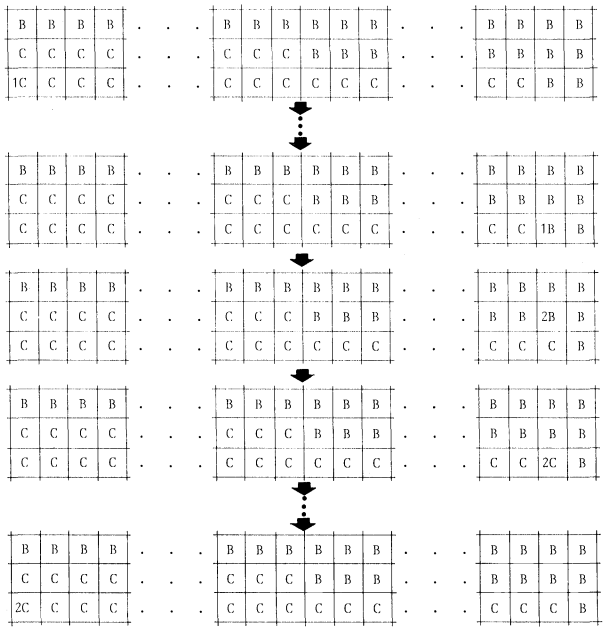


FIG. 11

cell >0, if $m>0$ holds (see Fig. 11) and the cell *sub'* for $m \leqq 0$ (see Fig. 12). Entering $\mathbf{R}^+$ via cell *sub* also changes one letter B to C (on the upper C-string) thus m changes to $m-1$ and the cell *sub'* is reached. Any operation *test* or *sub* always results in a prolonging of the C-strings whilst their difference contains the required information of the state of REG^+. With this technique negative integers may also be stored.

This proves U to be a universal machine.

3. Discussion. One may try to compare the different machine concepts with a structural complexity measure: Define the complexity of a Turing machine M, say with

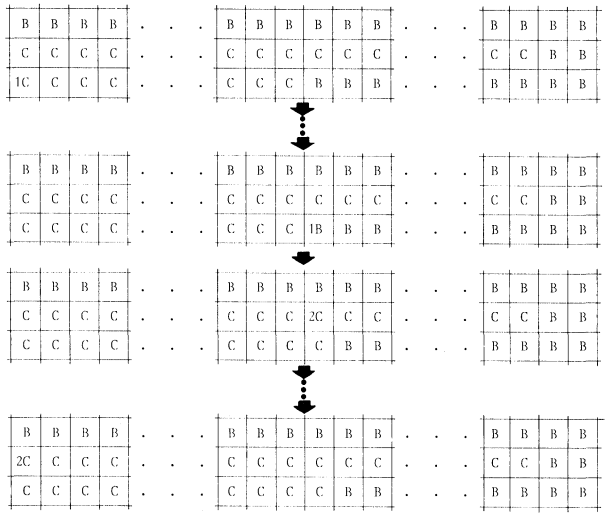


FIG. 12

n states, l letters, d dimensions, k tapes, h heads, etc., to be the total number of all possible instructions with at most n states, l letters, d dimensions, k tapes, h heads, etc. Taking into account whether a stop-instruction shall be available or not and whether a movement of the head has to be fulfilled in any step or not, we see that the mentioned Turing machines are of the following complexities:

U is of complexity $(4 \cdot 2 \cdot 4)^{2 \cdot 4} = 32^8 = 10^{12}$. Wagner's machine is of complexity $(4 \cdot 8 \cdot 4 + 1)^{4 \cdot 8} = 129^{32} = 3 \cdot 10^{67}$. Kleine Büning's two machines are of the complexities $(2 \cdot 5 \cdot 4 + 1)^{2 \cdot 5} = 41^{10} = 10^{16}$ and of $(10 \cdot 2 \cdot 4 + 1)^{10 \cdot 2} = 81^{20} = 10^{38}$ and the machine of Kleine Büning and Ottmann possesses the complexity $(3 \cdot 6 \cdot 4 + 1)^{3 \cdot 6} = 73^{24} = 5 \cdot 10^{44}$. Watanabe's and Minsky's machines possess the complexities $(8 \cdot 5 \cdot 2)^{8 \cdot 5} = 80^{40} = 10^{76}$ and $(7 \cdot 4 \cdot 2 + 1)^{7 \cdot 4} = 57^{28} = 10^{49}$. Although Hooper's machine seems to be pretty small it is of complexity $(2^4 \cdot 1 \cdot 3^4 + 1)^{1 \cdot 24} = 1297^{16} = 6 \cdot 10^{49}$, due to the four independently operating heads that increase the number of possible instruction sets. A further universal Turing machine of Hooper (1963) operates with 2 states, 3 letters and 2 heads, resulting in complexity $(3^2 \cdot 2 \cdot 3^2 + 1)^{2 \cdot 32} = 163^{18} = 6 \cdot 10^{39}$.

Comparing these results, we observe that the machine U has the lowest complexity. However, the comparison of different machine-concepts cannot be done without some force. Even Shannon's (1956) complexity measure (number of states times number of letters) for classical Turing machines (with one head and one one-dimensional tape) is by no means self-evident. One may also try to classify Turing machines with the help of a vector $v = (n, l, d, h)$ of the number of states (n), letters (l), the dimension of the tape (d), the number of heads (h), but in this case different concepts are no longer comparable by a linear ordering. However, such a classification offers the nice advantage that one may receive a sharp boundary for the existence and absence of universal machines. Due to Wagner (1973) there is no universal Turing machine in the class $(2, 2, d, 1)$ for all integers d. As we have shown the existence of a universal Turing machine in the class $(2, 4, 2, 1)$ a characterization of the class $(2, 3, 2, 1)$ would give a precise boundary.

One may object that the Turing machine U possesses no special stop-instruction. However, as a simulation without a need of a stop-instruction is reasonably definable (and on the $\mathbb{N}_0^2$-plane a stop-by-error is achievable, as a compensation) and Wagner's impossibility results holds good whether a stop-instruction is available or not, we see no necessity for such a stop-instruction. This research is restricted to Turing machines that operate only on finite tape-inscriptions. One may also investigate Turing machines operating on infinite, periodic or nonperiodic, initial inscriptions, but in this case all computation may be concealed in these inscriptions by using Rödding's (1969/70) technique of infinite periodic wire-connections to prove the undecidability of expressions of first order predicate logic without equality, functions, constants, predicates with 3 or more attached variables and with exactly one predicate with 2 attached variables.

As a simple result, also nonprinting, 1-state Turing machines, operating on an infinite, but periodic, initial inscription may be universal. Thus infinite inscriptions are not reasonable for research on the complexity of nonuniversal and universal machines.

A conclusion for two-head machines. It can easily be verified that our machine U in $(2, 4, 2, 1)$ yields a universal machine, U′, in the class $(2, 2, 2, 2)$: U′ operates on two copies of the initial tape (plane) inscriptions of U (the plane can be covered by two such inscriptions as we needed only inscriptions that cover $\mathbb{N} \times [1, m]$ for some fixed $m \in \mathbb{N}$) where both heads move synchronously on both inscriptions just as the single head of U does. By a simple coding two letters are now sufficient in both inscriptions to receive the behavior of U.

By Wagner's result (2, 2, 2, 1) cannot contain universal machines. This gives a first sharp boundary of the complexity between universal and nonuniversal Turing machines. The diagram of Fig. 13 visualizes the current situation:

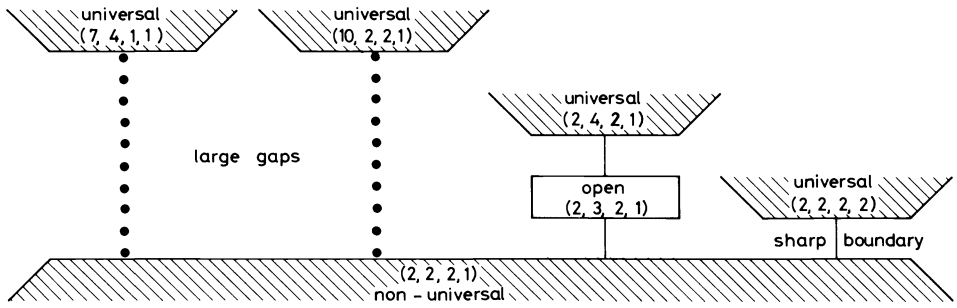


FIG. 13

REFERENCES

P. C. FISCHER (1965), *On formalisms for Turing machines*, J. Assoc. Comput. Mach., 12, pp. 570–580.

J. HARTMANIS AND R. E. STEARNS (1966), *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Englewood Cliffs, NJ.

PH. K. HOOPER (1963), *Some small, multitape universal Turing machines*, Computation Laboratory, Harvard Univ., Cambridge, MA.

H. KLEINE BÜNING (1977), *Über Probleme bei homogener Parkettierung von $\mathbb{Z} \times \mathbb{Z}$ durch Mealy-Automaten bei normierter Verwendung*, Ph.D. Thesis, Institut für math. Logik, Münster.

H. KLEINE BÜNING AND T. OTTMANN (1977), *Kleine universelle mehrdimensionale Turingmaschinen*, Elektron. Informations Verarbeit. Kybernetik, 13, pp. 179–201.

M. L. MINSKY (1962), *Size and structure of universal Turing machines using tag systems*, Proc. 5th Symp. in Appl. Math. (1962), American Mathematical Society, Providence, RI, pp. 229–238.

T. OTTMANN (1973), *Über Möglichkeiten zur Simulation endlicher Automaten durch eine Art sequentieller Netzwerke aus einfachen Bausteinen*, Z. Math. Logik Grundlagen Math., 19, pp. 223–238.

L. PRIESE (1974), *Über einfache unentscheidbare Probleme: Computational-und constructional-universelle asynchrone Räume*, Ph. D. thesis, Institut für math. Logik, Münster.

——— (1978), *Normed Networks: Their Mathematical Theory and Applicability*, Applied General Systems Research: Recent Developments and Trends, NATO Conference Series (II-Systems Science), vol. 5, pp. 381–394.

L. PRIESE AND D. RÖDDING (1974), *A combinatorial approach to self-correction*, J. Cybernetics, 4, pp. 7–25.

D. RÖDDING (1969/70), *Reduktionstypen der Prädikatenlogik*, Institut für math. Logik, Münster.

W. RÖDDING AND D. RÖDDING (1978), *Networks of finite automata*, Progress in Cybernetics & System Research, vol. 3, to appear.

C. E. SHANNON (1956), *A universal Turing machine with two internal states*, Automata Studies, Princeton University Press, Princeton, NJ, pp. 157–166.

J. C. SHEPHERDSON AND H. E. STURGIS (1963), *Computability of recursive functions*, J. Assoc. Comput. Mach., 10, pp. 217–255.

K. WAGNER (1973), *Universelle Turinmaschinen mit n-dimensionalem Band*, Elektron. Informations Verarbeit. Kybernetik, 9, pp. 423–431.

S, WATANABE (1961), *5-symbol 8-state and 5-symbol 6-state Turing machines*, J. Assoc. Comput. Mach., 8, pp. 476–584.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0004 $01.00/0

ON V-OPTIMAL TREES*

A. BAGCHI† AND J. K. ROY‡

Abstract. The problem of determining minimal cost trees when the cost function is a linear combination of degree path length and external path length arises in the study of optimal disk merge patterns, as first pointed out by Knuth. This paper considers the special case when internal nodes of degrees two and three only are permitted. The notion of the V-cost of a tree is introduced, and the properties of trees of minimal V-cost are found, with particular emphasis on the case when the external weights are all equal. Some algorithms are presented which generate trees of minimal V-cost under certain restrictive conditions.

Key words. 3-trees, Huffman cost, V-cost, V-optimal tree, convex function

1. Introduction. Binary trees of minimum weighted path length have applications in many areas such as construction of variable length codes, searching files, and merging sorted lists. Huffman's well-known algorithm can be used to build such trees. The generalized version of this algorithm is as follows:

ALGORITHM H. Let $\{w_1, w_2, \cdots, w_n\}$ be a multiset of n positive integer weights, and let $m \geqq 2$ be a given integer. We construct a tree $H(m, n)$ on the given multiset of weights in the manner described below:

If $m \geqq n$ then combine the n weights and stop. Otherwise, if $m = 2$ go to step H2; else go to step H1.

H1. Find s such that $1 \leqq s \leqq m-1$ and $s \equiv n \bmod (m-1)$. If $s = 1$ go to step H2. Else combine any s smallest weights in the multiset and replace them in it by the combined weight.

H2. Combine any m smallest weights in the multiset and replace them in the set by the combined weight. Repeat this step until only one weight remains. □

We will call a tree an *m-tree* if it has no internal node of degree greater than m. So $H(m, n)$ is an m-tree. The *Huffman cost* of an m-tree will be the sum of the weights of all internal nodes in the tree. It is known that $H(m, n)$ has minimal Huffman cost among all m-trees built on a given multiset of n weights (see Knuth [2, pp. 399–405]). We concern ourselves in this paper only with 3-trees. It should be noted that if n is odd then $H(3, n)$ has internal nodes of degree 3 only, and if n is even then $H(3, n)$ has one internal node of degree 2 which is formed by combining the two smallest weights in the given multiset.

The notion of the V-cost of a 3-tree, which we now introduce, may be viewed as a generalization of the idea of Huffman cost. Let two (nonzero) positive real numbers U and V be given. Let T be a 3-tree built on n given positive integer weights $w_1, w_2, \cdots, w_n$. Let M and N be the sums of the weights of all internal nodes in T of degrees 2 and 3 respectively. We consider the problem of minimizing the quantity $MU + NV$. Since only the ratio V/U is of significance, we can put $U = 1$ and try to minimize $M + NV$. We call the quantity $M + NV$ the *V-cost* of T, and we write it as $c(T, n, V)$. The tree with minimal V-cost is *V-optimal*, and we designate this tree and its cost by $Q(n, V)$ and $c(Q, n, V)$. (For some multisets of weights and some values of V there can be more than one V-optimal tree. In such cases $Q(n, V)$ will indicate some one of the V-optimal trees.) To take an example, let the given multiset of weights be $\{2, 2, 3, 3, 6\}$. In Fig. 1 we show three different trees with the above weights as external nodes. It can be checked that T_1 is V-optimal when $V = 1$, T_2 is V-optimal when $V = 2$,

* Received by the editors February 15, 1977.
† Indian Institute of Management, Calcutta, India.
‡ Indian Statistical Institute, Calcutta, India.

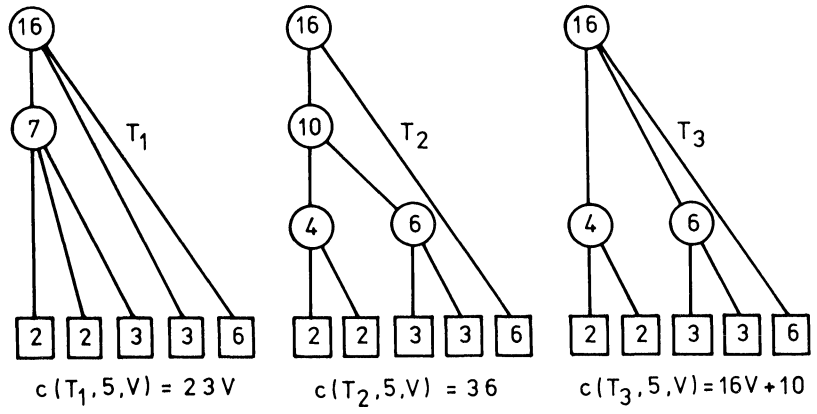


FIG. 1

and T_3 is V-optimal when $V = 1.5$. Note that Algorithm H generates the tree T_1 if $m = 3$ and the tree T_2 if $m = 2$.

Knuth [3, pp. 366–373] and Schlumberger and Vuillemin [4] have studied a more general problem in their efforts to find optimal disk merge patterns. Knuth uses a different notation to ours, and does not restrict himself to 3-trees. For a given tree T let $D(T)$ be the (weighted) degree path length of T, i.e., the sum over all leaves of T of the weight of a leaf multiplied by the sum of the degrees of the nodes on the path connecting this leaf to the root of T, and similarly let $E(T)$ be the (weighted) external path length of T. Knuth lets his cost function be $\alpha D(T) + \beta E(T)$, where α, β are given nonnegative real constants, and he presents a dynamic programming algorithm which minimizes the cost of n equal weights in $\mathcal{O}(n^2)$ time and $\mathcal{O}(n)$ memory space. To see how our problem resolves itself into a special case of Knuth's we confine ourselves to 3-trees and set

$$U = 2\alpha + \beta,$$

$$V = 3\alpha + \beta$$

in the cost function $MU + NV$ described in the last paragraph. Knuth remarks that usually in practical situations that arise in disk merging, $0 \leqq \alpha \leqq \beta$. In our notation, when we set $U = 1$, this would correspond to $1 \leqq V \leqq \frac{4}{3}$ provided $\beta \neq 0$.

In this paper we study the properties of V-optimal trees, and we try to determine which 3-trees are V-optimal. We succeed in our efforts only in some special cases. In § 2 we consider the general situation where the given weights are not necessarily equal. Section 3 deals with a particular class of sequences of weights which we name rapidly growing sequences. In § 4 the weights are all equal.

2. The general case. When the n given weights are not all equal it becomes quite difficult to construct a V-optimal tree for an arbitrary (positive) V. We have succeeded in showing that when $0 < V \leqq 1$ then $H(3, n)$ is V-optimal and when $V \geqq \frac{5}{3}$ then $H(2, n)$ is V-optimal. In the range $1 < V < \frac{5}{3}$ the problem is still unresolved in the general case, and in §4 we confine ourselves to this range.

THEOREM 1. *Let a multiset of n weights be given. Then*

(i) *the tree $H(3, n)$ constructed on the given multiset is V-optimal for $0 < V \leqq 1$;*

(ii) *the tree $H(2, n)$ constructed on the given multiset is V-optimal for $V \geqq \frac{5}{3}$.*

Proof. All trees referred to below are built on the given multiset of n weights. Let M_2 be the Huffman cost of $H(2, n)$, and let M_3 and N_3 be the sums of weights of internal

nodes in $H(3, n)$ of degrees 2 and 3 respectively. It is to be noted that $M_3 = 0$ when n is odd. Let T be any 3-tree and let M and N be the sums of weights of internal nodes in T of degrees 2 and 3 respectively. Then if T is V-optimal,

$$M + NV \leqq M_2 \tag{1}$$

and

$$M + NV \leqq M_3 + N_3 V. \tag{2}$$

Moreover, since $H(3, n)$ has minimal Huffman cost among all 3-trees,

$$M + N \geqq M_3 + N_3. \tag{3}$$

By Algorithm H, when n is even the two smallest weights in the given multiset combine to form the internal node of degree 2 in $H(3, n)$, so

$$M_3 \leqq M. \tag{4}$$

We now show that

$$M + \frac{5N}{3} \geqq M_2. \tag{5}$$

(For a more general result of a similar type see Bagchi and Roy [1].) Consider any internal node of degree 3 in T. Let the three sons of this node have weights a, b and c where $a \leqq b \leqq c$. We can split this node into two degree 2 nodes as shown in Fig. 2. The sum of the weights of these two degree 2 nodes is

$$2(a + b) + c$$

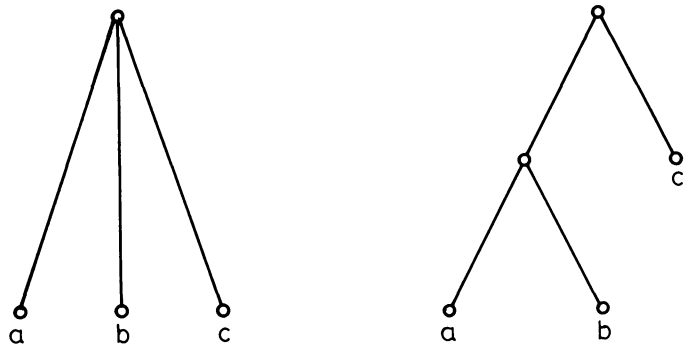


FIG. 2

which is less than or equal to

$$\tfrac{5}{3}(a + b + c).$$

This proves inequality (5), since $H(2, n)$ has minimal Huffman cost among all 2-trees.

By inequalities (1) and (5)

$$M + NV \leqq M + \frac{5N}{3}.$$

It follows that for $V > \frac{5}{3}$, $N = 0$. So for $V > \frac{5}{3}$, T may be taken to be identical with $H(2, n)$. Moreover, if $H(2, n)$ is V-optimal for $V > \frac{5}{3}$, $H(2, n)$ will be V-optimal for $V = \frac{5}{3}$ as well.

By inequality (4), $M - M_3$ is nonnegative, so by inequality (3), for $0 < V \leqq 1$,

$$M - M_3 \geqq (N_3 - N)V.$$

Hence by inequality (2), for any V in the range $0 < V \leqq 1$,

$$M - M_3 = (N_3 - N)V.$$

Ignoring those trees (if any) which are V-optimal at isolated values of V (i.e., at single points) and considering only those trees that are V-optimal over ranges of V values, we conclude that for $0 < V \leqq 1$,

$$M = M_3 \quad \text{and} \quad N = N_3.$$

Therefore, we conclude that T may be taken to be identical with $H(3, n)$ for $0 < V \leqq 1$. □

We now come to some general properties of V-optimal trees. We note to begin with that every subtree of a V-optimal tree is V-optimal on its own multiset of external weights. A more interesting property is the following. We know that at any step in construction of $H(2, n)$ using Algorithm H, a minimum weight pair get combined. Similarly, in the construction of $H(3, n)$ using Algorithm H, at the first step either a minimum weight pair or a minimum weight triple get combined, and at each subsequent step a minimum weight triple get combined. We show now that given V and any multiset of weights there exists a V-optimal tree on this set of weights which can be built in such a way that at any step in its construction either a minimum weight pair or a minimum weight triple get combined. The problem we face is in deciding whether to combine a minimum weight pair or to combine a minimum weight triple at any step. This problem, if solved, would give us a general algorithm for constructing V-optimal trees.

Before coming to the theorem we introduce the notion of *V-level*. Let us define the V-level of any node q in a 3-tree T built on n given weights as $l_q + m_q V$, where l_q and m_q are the total numbers of degree 2 and degree 3 nodes (excluding q) on the path joining q to the root of T. When $V = 1$ the V-level of any node in T is identical with its level in T as normally defined. Clearly,

$$c(T, n, V) = w_i(l_i + m_i V)$$

where the summation is over the n given external weights $w_1, w_2, \cdots, w_n$ numbered as nodes $1, 2, \cdots, n$ in T. If the tree T happens to be V-optimal, then for any two nodes q and r in T, internal or external, if the weight of q is less than the weight of r, then the V-level of q is greater than or equal to the V-level of r. In the proof below, when we talk about the V-level of a weight, we mean the V-level of the node, whether internal or external, with which that weight is associated.

THEOREM 2. *Let V and a multiset of n weights be given. Then there exists a V-optimal tree T on this multiset which can be so built that at any step in the construction of T:*

(i) *if two nodes (internal or external) get combined, the weights of these two nodes form a minimum weight pair at that step;*

(ii) *if three nodes (internal or external) get combined, the weights of these three nodes form a minimum weight triple at that step.*

Proof. Let S be a V-optimal tree on the given multiset of weights. Let a, b and c be the weights of any three nodes (internal or external) in S, such that $a \leqq b \leqq c$. Then, since S is V-optimal, V-level of $a \geqq V$-level of $b \geqq V$-level of c. Hence if the V-levels of a

and c are equal, then

$$V\text{-level of } b = V\text{-level of } a.$$

It follows that if any two node weights in S have the same V-level, then all intermediate weights also have an identical V-level. We can now get the tree T from S by simply reassigning nodes to their fathers at each V-level in order of increasing weights, starting at the numerically largest V-level in S and working towards the root of S. □

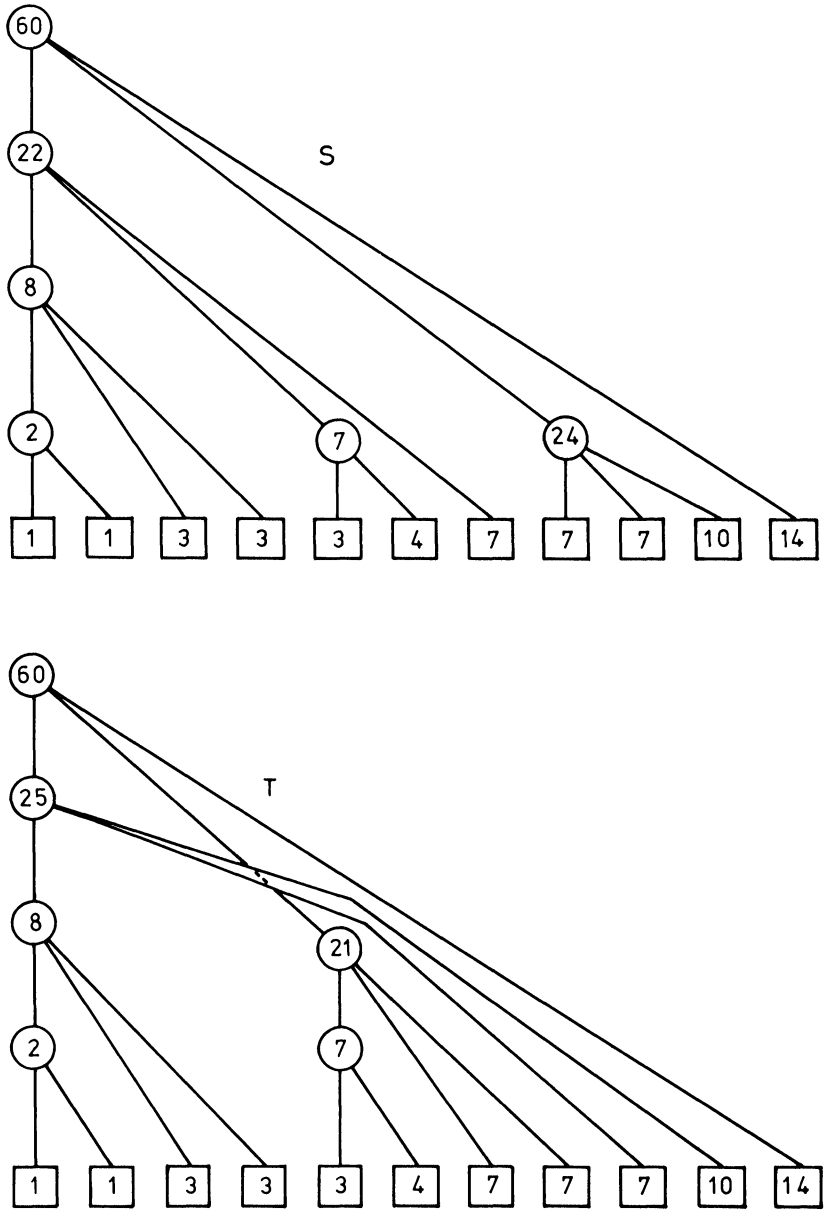


FIG. 3

In Fig. 3 we give an example of a tree S built on the multiset of weights $\{1, 1, 3, 3, 3, 4, 7, 7, 7, 10, 14\}$ which is V-optimal for $V = \frac{3}{2}$ but which does not satisfy

the conditions of Theorem 2. The corresponding tree T which does satisfy the conditions is also shown.

A question we have been unable to answer is the following. It is clear from Theorem 1 that for a given multiset of at least three weights the root node of a V-optimal tree is degree 3 for small V (i.e., $V \leqq 1$) and degree 2 for large V (i.e., $V \geqq \frac{5}{3}$). So there exist values of V, for instance, at which no V-optimal tree on the given multiset has root of degree 3. Consider the lower bound of all such values of V. When V exceeds this lower bound in value, must the root of a V-optimal tree necessarily be of degree 2? This is equivalent to asking whether the set of values of V at which a V-optimal tree with root of degree 3 exists is a connected subset of all real numbers. A similar question can be asked with 3 replaced by 2. The answer to the above questions is in the affirmative for rapidly growing sequences of weights studied in the next section.

3. Rapidly growing sequences. In this section we study a class of sequences of nondecreasing weights which we call rapidly growing sequences. As we shall see, the construction of V-optimal trees on such sequences can be achieved in a relatively straightforward manner, so this section serves to illustrate and clarify some of the concepts introduced earlier.

DEFINITION. A nondecreasing sequence of positive integer weights $w_1, w_2, w_3, \cdots$ is called a *rapidly growing sequence* if

$$w_i + w_{i+1} \leqq w_{i+2} \quad \text{for all } i \geqq 1.$$

When $w_1 = w_2 = 1$, and the inequality in the above definition is replaced by an equality, we get a Fibonacci sequence of weights $1, 1, 2, 3, 5, 8, 13, \cdots$.

Let V and a rapidly growing sequence of weights $w_1, w_2, w_3, w_4, \cdots$ be given. By Theorem 2, there is a V-optimal tree on the given sequence of weights which can be so built that at any step in its construction either a minimum weight pair or a minimum weight triple combine. For a rapidly growing sequence this V-optimal tree will lean completely to one side. This suggests the following "bottom-up" algorithm for creating a V-optimal tree $Q(i, V)$ on the i weights $w_1, w_2, \cdots, w_i$. We represent the V-cost of $Q(i, V)$ by $c(Q, i, V)$, and the degree of the root of $Q(i, V)$ by $d(Q, i, V)$.

ALGORITHM R. Let V and a rapidly growing sequence of n weights $w_1, w_2, \cdots, w_n$ be given. This algorithm creates a V-optimal tree $Q(n, V)$ on the weights by inductively determining the quantities $d(Q, i, V)$ for $2 \leqq i \leqq n$. Set

$$c(Q, 1, V) = 0,$$

$$c(Q, 2, V) = w_1 + w_2 \quad \text{and} \quad d(Q, 2, V) = 2.$$

For $3 \leqq i \leqq n$, let

$$c(Q, i, V) = \min \{W_i + c(Q, i-1, V),\ W_i V + c(Q, i-2, V)\}$$

and

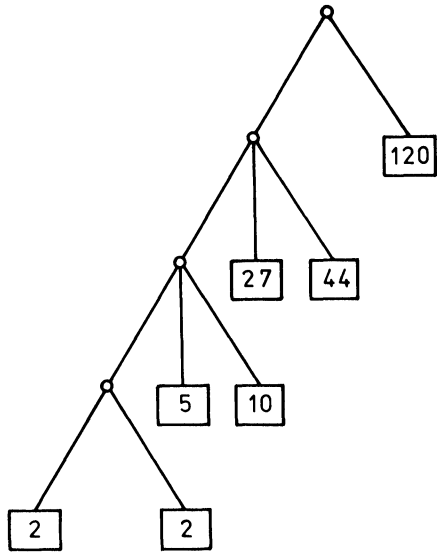
$$d(Q, i, V) = \begin{cases} 2 & \text{if } c(Q, i, V) = W_i + c(Q, i-1, V), \\ 3 & \text{otherwise;} \end{cases}$$

where

$$W_i = w_1 + w_2 + \cdots + w_i.$$

As an example, suppose $V = 1.4$ and the given rapidly growing sequence is 2, 2, 5, 10, 27, 44, 120. Figure 4 shows the table and the corresponding V-optimal tree. Since the table gives $d(Q, 7, V) = 2$, the root node of the tree must have degree 2.

i	c(Q,i+1,V)	d(Q,i+1,V)
1	4	2
2	12.6	3
3	30.6	3
4	76.6	2
5	156.6	3
6	366.6	2



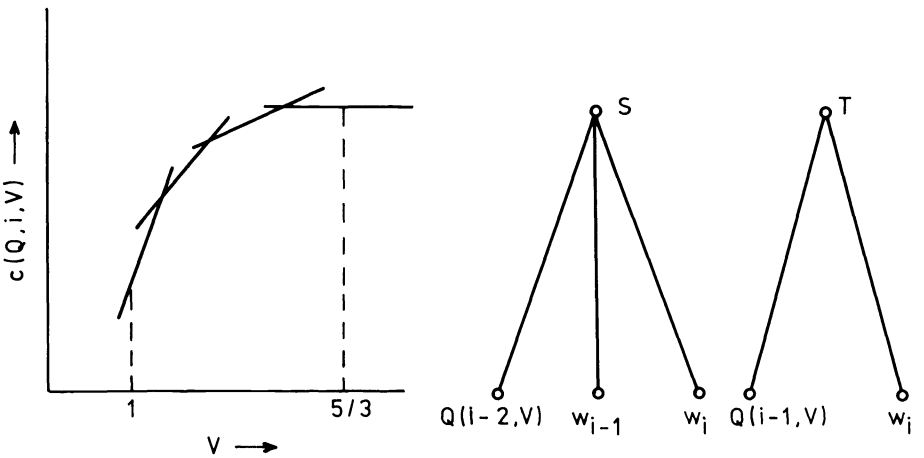


FIG. 4

Again, since $d(Q, 6, V) = 3$, the left son of the root has degree 3, and so on. On a sequence of n weights, Algorithm R requires $\mathcal{O}(n)$ computations (i.e., additions, multiplications, and comparisons).

We would now like to show that with increase of V, $d(Q, n, V)$ changes value just once from 3 to 2. We begin by remarking that $c(Q, i, V)$ for $i \geqq 2$ when plotted against V is a continuous curve consisting of a finite number of straight line segments, since there are finitely many 3-trees with the rapidly growing sequence of weights $w_1, w_2, \cdots, w_i$ as external node weights (see Fig. 4). By Theorem 1 this curve has constant slope for $V \leqq 1$ and zero slope for $V \geqq \frac{5}{3}$. So given a positive integer $n \geqq 3$, there are positive real numbers $V_0, V_1, \cdots, V_r$ for some $r > 0$ such that

(i) $$1 = V_0 < V_1 < V_2 \cdots < V_{r-1} < V_r = \tfrac{5}{3}$$

and

(ii) for *each* k, $0 < k \leqq r$, and for *each* i, $2 \leqq i < n$, $c(Q, i, V)$ is a straight line when plotted against V in the interval $V_{k-1} \leqq V \leqq V_k$.

We can therefore write, for $2 \leqq i < n$,

$$c(Q, i, V) = a_i(k) + b_i(k)V$$

where $a_i(k)$ and $b_i(k)$ are nonnegative integers which depend only on k, $0 < k \leqq r$.

Now consider the trees S and T shown in Fig. 4. Clearly, $Q(i, V)$ for $3 \leqq i \leqq n$ is either S or T. In particular, when $V \leqq 1$, $Q(i, V)$ is identical with S, and when $V \geqq \frac{5}{3}$, $Q(i, V)$ is identical with T. We can write

$$c(S, i, V) = W_i V + c(Q, i-2, V),$$
$$c(T, i, V) = W_i + c(Q, i-1, V)$$

where $W_i = w_1 + w_2 + \cdots + w_i$ as before. It follows that $c(S, i, V)$ or $c(T, i, V)$ when plotted against V will be a continuous curve consisting of at most $r+2$ straight line segments. In fact, $c(S, i, V)$ or $c(T, i, V)$ will be a straight line when plotted against V in the interval $V_{k-1} \leqq V \leqq V_k$ for $0 < k \leqq r$. Let us write

$$c(S, i, V) = e_i(k) + f_i(k)V,$$
$$c(T, i, V) = g_i(k) + h_i(k)V$$

where $e_i(k)$, $f_i(k)$, $g_i(k)$ and $h_i(k)$ are nonnegative integers which depend only on k. Then $b_i(k)$ equals either $f_i(k)$ or $h_i(k)$ for $3 \leqq i < n$, and

$$f_i(k) = \begin{cases} W_3 & \text{for } i = 3, \\ b_{i-2}(k) + W_i & \text{for } 4 \leqq i \leqq n; \end{cases}$$
$$h_i(k) = b_{i-1}(k) \quad \text{for } 3 \leqq i \leqq n.$$

If we can now show that $f_n(k) > h_n(k)$ for each k, $0 < k \leqq r$, then it will follow that the degree of the root node of $Q(n, V)$ changes just once from 3 to 2 as V increases from 1 to $\frac{5}{3}$.

FACT. *$f_n(k) > h_n(k)$ for each k, $0 < k \leqq r$, and for $n \geqq 3$.*

Proof. We fix k and do an induction on n. When $n = 3$, $f_n(k) = W_3$ and $h_n(k) = 0$, so the fact is true. We assume that $f_i(k) > h_i(k)$ for $3 \leqq i < n$. Now

$$f_n(k) = b_{n-2}(k) + W_n \quad \text{and} \quad h_n(k) = b_{n-1}(k).$$

There are four cases:

Case 1.

$$f_n(k) = f_{n-2}(k) + W_n,$$
$$h_n(k) = f_{n-1}(k);$$

Case 2.

$$f_n(k) = f_{n-2}(k) + W_n,$$
$$h_n(k) = h_{n-1}(k);$$

Case 3.

$$f_n(k) = h_{n-2}(k) + W_n,$$
$$h_n(k) = f_{n-1}(k);$$

Case 4.

$$f_n(k) = h_{n-2}(k) + W_n,$$
$$h_n(k) + h_{n-1}(k).$$

Taking Case 3 for example

$$\begin{aligned} f_n(k) - h_n(k) &= h_{n-2}(k) + W_n - f_{n-1}(k) \\ &= b_{n-3}(k) + W_n - b_{n-3}(k) - W_{n-1} \\ &= w_n > 0. \end{aligned}$$

The other cases can be taken care of in a similar manner. □

Let $V_0(n)$ be the value of V at which $d(Q, n, V)$ changes from 3 to 2. For an arbitrary rapidly growing sequence $w_1, w_2, \cdots, w_n$, $V_0(n)$ is a complicated function of the weights. When the sequence of weights is the Fibonacci sequence, however, it is not hard to show that

$$V_0(n) = \frac{w_{n+3} - 2}{w_{n+2} - 1}$$

which approximately equals the golden ratio when n is large. This yields a "top-down" procedure for generating a V-optimal tree in this case, since the degree of the root node can be determined by just comparing V with $V_0(n)$, and this procedure can be iterated to get the rest of the tree.

4. The equal weights case. We come finally to the equal weights case, where we can take all the weights in the multiset to be unit weights. Even in this apparently simple situation, we have been successful in evolving a simple algorithm for generating V-optimal trees only for certain ranges of values of V lying between 1 and $\frac{5}{3}$. For other ranges of values we have not been able to prove that the simple algorithm works, although it looks likely that it does. We begin by showing that when V lies between 1 and $\frac{4}{3}$, the V-optimal tree on n unit weights may be taken to be identical with $H(3, n)$.

THEOREM 3. *Given any V in the range $1 \leqq V \leqq \frac{4}{3}$, $H(3, n)$ built on n unit weights is a V-optimal tree.*

Proof. Let T be a V-optimal tree on n unit weights for the given V in the range $1 \leqq V \leqq \frac{4}{3}$. We first show that for $n \geqq 3$ the root node of T can be taken to be degree 3. Suppose the root node of T is not degree 3. Then one of the five cases shown in the

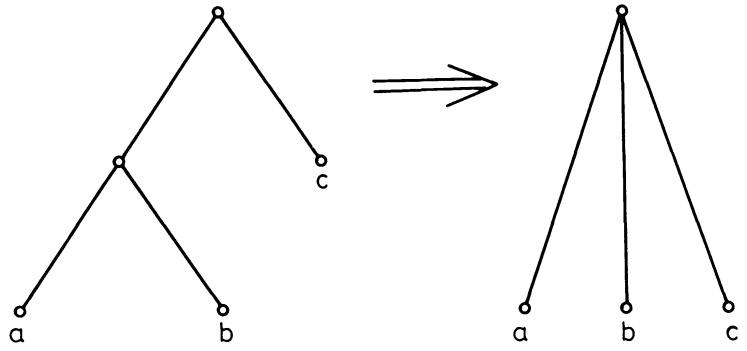


FIG. 5

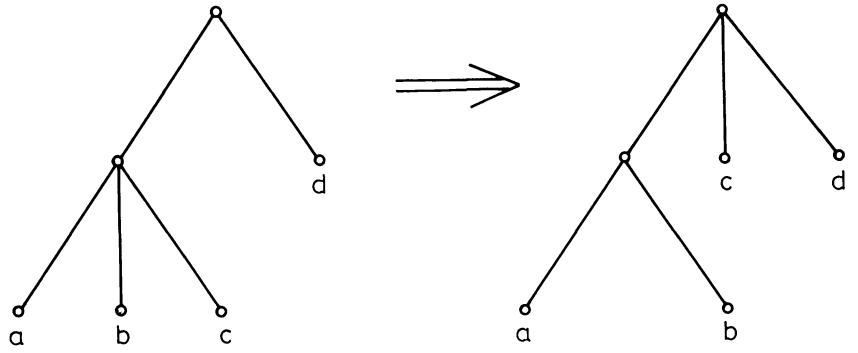


FIG. 6

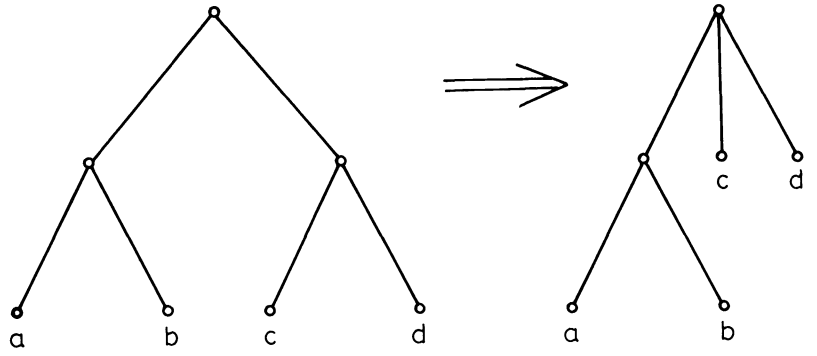


FIG. 7

lefthand sides of Figs. 5 through 9 will apply. In each case a simple transformation yields the tree shown in the corresponding righthand side which is of no greater V-cost but has a root of degree 3. In the figures we have labeled some nodes with their weights. In Fig. 5, $c = 1$, i.e., the node labeled with the weight c is an external node. Similarly, in Fig. 6, $d = 1$. All other nodes in the figures may be either internal or external. We now take the individual cases one by one:

Case 1. *Figure* 5. Since $c = 1$, we have $2(a+b) \geqq c$. Hence $2a+2b+c \geqq (a+b+c)V$ for V in the given range.

Case 2. *Figure* 6. Since $d = 1$ and $c \geqq 1$, $c+d = c+1 \geqq V = dV$. Hence

$$(a+b+c)V+(a+b+c+d) \geqq (a+b)+(a+b+c+d)V.$$

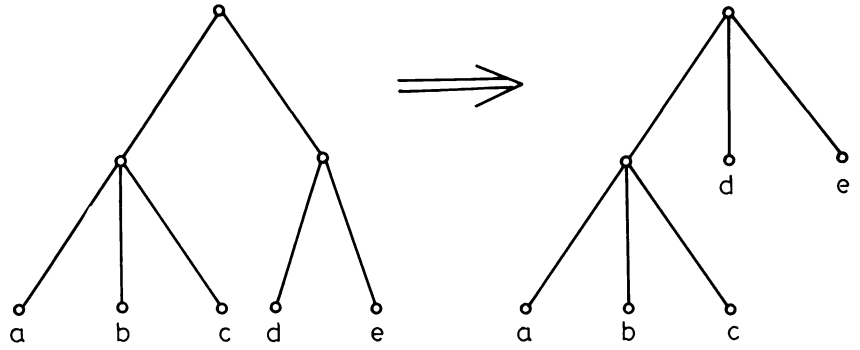


FIG. 8

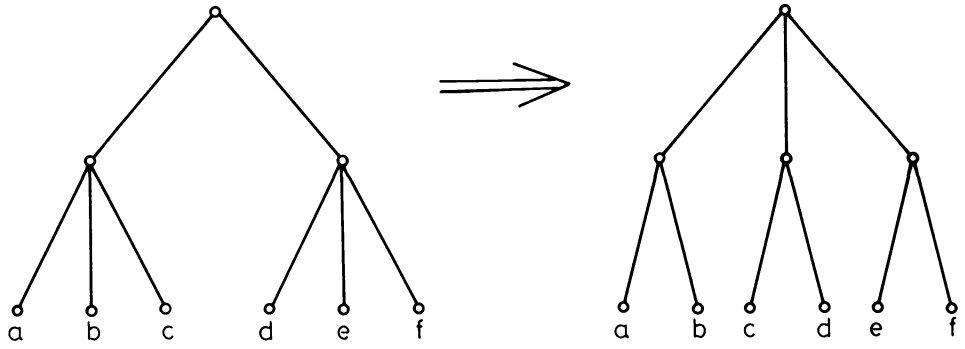


FIG. 9

Case 3. *Figure* 7. We assume here that $a \leqq b \leqq c \leqq d$. So, $2(a+b+c+d) \geqq (a+b)+(a+b+c+d)V$.

Case 4. *Figure* 8. We can assume here that $a \leqq b \leqq c \leqq d \leqq e$, because if $\min(d, e) < \max(a, b, c)$ then T is not V-optimal. So it follows that

$$(a+b+c)V+(a+b+c+2d+2e) \geqq (a+b+c)+(a+b+c+d+e)V.$$

Case 5. *Figure* 9. Here the two trees shown have the same V-cost.

Since every subtree of a V-optimal tree is V-optimal, once the root node of the entire tree has been made degree 3, the roots of the subtrees can be similarly transformed to degree 3. It is clear that if any degree 2 nodes remain in the V-optimal tree after all necessary transformations are completed, then the sons of such degree 2 nodes must be external nodes. Suppose there are two degree 2 nodes r and s in the V-optimal tree with weights a and b. Then $a = b = 2$. Suppose

$$l_r + m_r V \geqq l_s + m_s V.$$

Then if we attach one of the sons of r to s, and raise the other son of r one level, the change in V-cost is

$$\begin{aligned} &4+2(l_r+m_rV+l_s+m_sV)-3V-3(l_s+m_sV)-(l_r+m_rV) \\ &\qquad = (4-3V)+[l_r+m_rV-(l_s+m_sV)] \end{aligned}$$

which is nonnegative for $V \leqq \frac{4}{3}$. So by Theorem 2, the V-optimal tree we are left with may be taken to be identical with $H(3, n)$. □

As a consequence of Theorem 3, we need to consider only values of V lying between $\frac{4}{3}$ and $\frac{5}{3}$ for further study. We now take help of the concept of a convex function,

as defined and used in Knuth [3, p. 372]. We say that a function f with the positive integers as domain and the real numbers as range is *convex* if for every $n \geqq 2$

$$f(n)-f(n-1) \leqq f(n+1)-f(n).$$

Suppose we can show that for any given V in the range $\frac{4}{3} < V < \frac{5}{3}$, the V-cost $c(Q, n, V)$ of a V-optimal tree $Q(n, V)$ built on n equal unit weights is convex when viewed as a function of n. Then by Knuth [3, p. 372], when $Q(n, V)$ has root of degree 2, the two subtrees of the root are $Q(\lfloor n/2 \rfloor, V)$ and $Q(\lfloor (n+1)/2 \rfloor, V)$ and when $Q(n, V)$ has root of degree 3, the three subtrees of the root are $Q(\lfloor n/3 \rfloor, V)$, $Q(\lfloor (n+1)/3 \rfloor, V)$ and $Q(\lfloor (n+2)/3 \rfloor, V)$. This yields an algorithm very similar to Algorithm R for generating $Q(n, V)$:

ALGORITHM T. Let V in the range $\frac{4}{3} < V < \frac{5}{3}$ be given. This algorithm constructs a tree $T(n, V)$ on n equal unit weights. The tree $T(n, V)$ is V-optimal provided the V-cost of a V-optimal tree is convex when viewed as a function of n for the given V. We let $c(T, n, V)$ and $d(T, n, V)$ represent the V-cost $T(n, V)$ and the degree of the root node of $T(n, V)$ respectively, and we define them inductively. Two auxiliary functions $f(T, n, V)$ and $g(T, n, V)$ are used in the definition.

Set

$$c(T, 1, V) = 0,$$
$$c(T, 2, V) = 2,$$
$$d(T, 2, V) = 2.$$

Now let us put, for $3 \leqq i \leqq n$,

$$f(T, i, V) = i + c\left(T, \left\lfloor \frac{i}{2} \right\rfloor, V\right) + c\left(T, \left\lfloor \frac{i+1}{2} \right\rfloor, V\right)$$

and

$$g(T, i, V) = iV + c\left(T, \left\lfloor \frac{i}{3} \right\rfloor, V\right) + c\left(T, \left\lfloor \frac{i+1}{3} \right\rfloor, V\right) + c\left(T, \left\lfloor \frac{i+2}{3} \right\rfloor, V\right).$$

We define

$$c(T, i, V) = \min\{f(T, i, V), g(T, i, V)\}$$

and

$$d(T, i, V) = \begin{cases} 2 & \text{if } c(T, i, V) = f(T, i, V), \\ 3 & \text{otherwise.} \end{cases}$$

□

In the construction of the tree $T(n, V)$ we do not need to know $d(T, i, V)$ for all i in the range $2 \leqq i \leqq n$. For example, when $n = 20$, $d(T, 8, V)$ and $d(T, 9, V)$ are never needed. So a reduction in the computation is possible, but since this can only be achieved by computing in advance the values of i for which $c(T, i, V)$ needs to be known, the net saving does not appear to be substantial.

As an example, suppose $V = 1.574$ and $n = 15$. Figure 10 shows the table and the tree $T(15, V)$. Algorithm Q takes $\mathcal{O}(n)$ time to construct $T(n, V)$. So we would expect Algorithm Q to be very useful in the construction of a V-optimal tree $Q(n, V)$ for those values of V for which we have been able to prove that $c(Q, n, V)$ is convex viewed as a function of n. It so happens, however, that in our theorems below on the convexity of $c(Q, n, V)$ we are able to specify the degree of the root node of $Q(n, V)$, so that

		n = 15	V = 1.574
No.	i	c (T, i, V)	d (T, i, V)
1	2	2	2
2	3	4.722	3
3	4	8	2
4	5	11.722	2
5	6	15.444	2 or 3
6	7	19.722	2
7	8	24	2
8	15	58.722	2

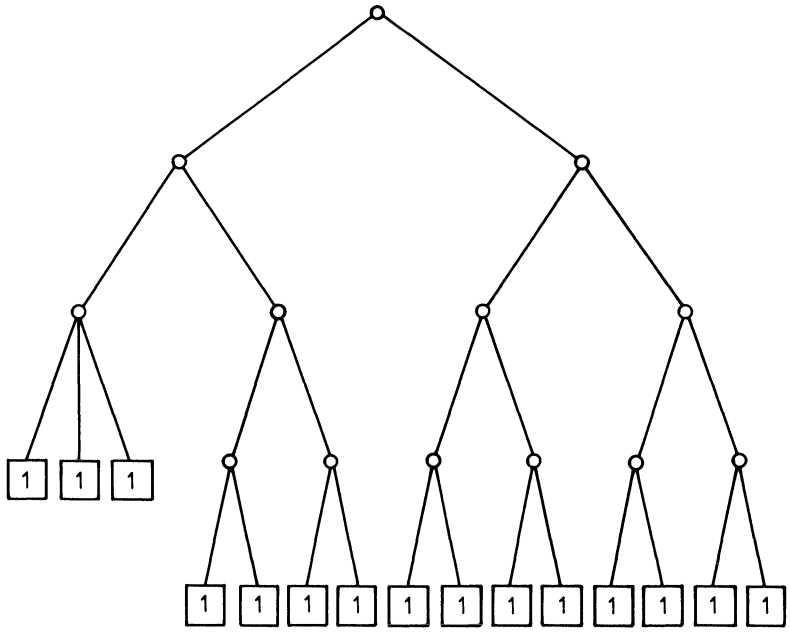


FIG. 10

Algorithm Q is no longer needed for constructing $Q(n, V)$. For some values of V in the range $\frac{4}{3} < V < \frac{5}{3}$ we have not been able to prove the convexity of $c(Q, n, V)$. If it should turn out that $c(Q, n, V)$ is not convex for some V (viewed as a function of n), then for that value of V Algorithm Q cannot be used for constructing a V-optimal tree, and a more general algorithm similar to that given in Knuth [3, p. 368] will be needed. It is interesting to observe in this connection that for $V < \frac{4}{3}$ the cost $c(Q, n, V)$ of a V-optimal tree built on n unit weights is not necessarily convex in n. Consider, for example, the case $n = 5$.

The major task remaining is to show that the V-cost function of a V-optimal tree, namely $c(Q, n, V)$, is convex viewed as a function of n for any given V in the range $\frac{4}{3} < V < \frac{5}{3}$. We have been able to show this *except* for V in the small range $\frac{30}{19} < V < \frac{35}{22}$.

No. of external nodes (n)	Range of V									
	$\frac{4}{3} < V \leqq \frac{3}{2}$		$\frac{3}{2} < V \leqq \frac{11}{7}$		$\frac{11}{7} < V \leqq \frac{30}{19}$		$\frac{35}{22} \leqq V < \frac{8}{5}$		$\frac{8}{5} \leqq V < \frac{5}{3}$	
	V-cost	Degree of root	V-cost	Degree of root	V-cost	Degree of root	V-cost	Degree of root	V-cost	Degree of root
2	2	2	2	2	2	2	2	2	2	2
3	$3V$	3	$3V$	3	$3V$	3	$3V$	3	$3V$	3
4	$4V+2$	3	8	2	8	2	8	2	8	2
5	$5V+4$	3	$3V+7$	2	$3V+7$	2	$3V+7$	2	$3V+7$	2
6	$6V+6$	2 or 3	$6V+6$	2 or 3	$6V+6$	2 or 3	$6V+6$	2 or 3	$6V+6$	2 or 3
7	$10V+4$	3	$10V+4$	3	$3V+15$	2	$3V+15$	2	$3V+15$	2
8	$14V+2$	3	$14V+2$	3	24	2	24	2	24	2
9	$18V$	3	$18V$	3	$18V$	3	$18V$	3	$3V+24$	2
10	$20V+2$	3	$16V+8$	3	$16V+8$	3	$16V+8$	3	$6V+24$	2
11	$22V+4$	3	$14V+16$	3	$14V+16$	3	$14V+16$	3	$9V+24$	2
12	$24V+6$	3	$12V+24$	3	$12V+24$	2 or 3	$12V+24$	2 or 3	$12V+24$	2 or 3
15	$30V+12$	3	$24V+21$	3	$3V+54$	2	$3V+54$	2	$3V+54$	2
18	$36V+18$	3	$36V+18$	3	$36V+18$	3	$36V+18$	2	$6V+66$	2
20	$46V+14$	3	$46V+14$	3	$32V+36$	3	$32V+36$	2	$12V+68$	2
24	$66V+6$	3	$66V+6$	3	$24V+72$	2 or 3	$24V+72$	2 or 3	$24V+72$	2 or 3
25	$71V+4$	3	$71V+4$	3	$43V+48$	3	$21V+83$	2	$21V+83$	2
30	$90V+6$	3	$78V+24$	3	$78V+24$	3	$6V+138$	2	$6V+138$	2
45	$135V+36$	3	$117V+63$	3	$54V+162$	3	$54V+162$	2	$39V+186$	2
60	$198V+42$	3	$198V+42$	3	$156V+108$	3	$12V+336$	2	$12V+336$	2
90	$360V+18$	3	$324V+72$	3	$324V+72$	3	$108V+414$	2	$78V+462$	2

FIG. 11

We summarize our conclusions in a series of theorems given below. Since the theorems have similar proofs, we give the proof on Theorem 8 only. Figure 11 gives expressions for $c(Q, n, V)$ for various values of V and certain typical values of n. The expressions for $c(Q, n, V)$ given in the theorems were initially obtained by generalizing from the values given in Fig. 11, which were determined by trial and error. To avoid confusion we note that for some ranges of values of V and some values of n, the root node of a V-optimal tree can have degree 2 or 3. Consider, for example, $V = \frac{3}{2}$ and $n = 6$. In such cases we have taken the degree of the root node to be 3 in Theorems 4, 5 and 6, and we have taken the degree of the root to be 2 in Theorems 7 and 8.

In the range $\frac{4}{3} < V \leqq \frac{3}{2}$, we arrive at the convexity of $c(Q, n, V)$ by generalizing the proof of Theorem L in Knuth [3, p. 371]. We state the theorem below:

THEOREM 4. *Let V in the range $\frac{4}{3} < V \leqq \frac{3}{2}$ be given. The V-cost $c(Q, n, V)$ of a V-optimal tree built on n equal unit weights is convex viewed as a function of n and has the following expression for $n \geqq 2$.*

$$c(Q, n, V) = \begin{cases} knV + 2(V-1)(n-3^k) & \text{for } 2 \cdot 3^{k-1} < n \leqq 3^k, \\ knV + 2(n-3^k) & \text{for } 3^k < n \leqq 2 \cdot 3^k. \end{cases}$$

The root node of $Q(n, V)$ has degree 3 for $n > 2$ and degree 2 for $n = 2$.

When the value of V crosses $\frac{3}{2}$, the root nodes of $Q(4, V)$ and $Q(5, V)$ become degree 2. This changes the expression for $c(Q, n, V)$ and we get the following theorem:

THEOREM 5. *Let V in the range $\frac{3}{2} < V \leqq \frac{11}{7}$ be given. The V-cost $c(Q, n, V)$ of a V-optimal tree built on n equal unit weights is convex viewed as a function of n and has the following expression for $n \geqq 2$.*

$$c(Q, n, V) = \begin{cases} k \cdot 3^k \cdot V + (n-3^k)[8-(4-k)V] & \\ \qquad\qquad \text{for } 3^k \leqq n < 4 \cdot 3^{k-1}, & \\ [V(k-1)+2] \cdot 4 \cdot 3^{k-1} + (n-4 \cdot 3^{k-1})[(k+2)V-1] & \\ \qquad\qquad \text{for } 4 \cdot 3^{k-1} \leqq n < 2 \cdot 3^k, & \\ (kV+1) \cdot 2 \cdot 3^k + (n-2 \cdot 3^k)[(k+3)V-2] & \\ \qquad\qquad \text{for } 2 \cdot 3^k \leqq n < 3^{k+1}. & \end{cases}$$

The root node of $Q(n, V)$ is degree 2 for $n = 2$, 4 and 5, and it is degree 3 otherwise.

Again when V crosses $\frac{11}{7}$, the root nodes of $Q(7, V)$ and $Q(8, V)$ become degree 2. This causes a further change in the expression for $c(Q, n, V)$.

THEOREM 6. *Let V in the range $\frac{11}{7} < V \leqq \frac{30}{19}$ be given. The V-cost $c(Q, n, V)$ of a V-optimal tree built on n equal unit weights is convex viewed as a function of n and has the following expression for $n \geqq 6$.*

$$c(Q, n, V) = \begin{cases} n[9-(4-k)V] + 8 \cdot 3^k(V-2) & \text{for } 2 \cdot 3^k \leqq n < 8 \cdot 3^{k-1} \\ n[(k+17)V-24] + 8 \cdot 3^{k+1}(3-2V) & \text{for } 8 \cdot 3^{k-1} \leqq n < 3^{k+1} \\ n[8-(3-k)V] + 4 \cdot 3^{k+1}(V-2) & \text{for } 3^{k+1} \leqq n < 4 \cdot 3^k \\ n[10-(4-k)V] + 16 \cdot 3^k(V-2) & \text{for } 4 \cdot 3^k \leqq n < 16 \cdot 3^{k-1} \\ n[(k+17)V-23] + 16 \cdot 3^{k+1}(3-2V) & \text{for } 16 \cdot 3^{k-1} \leqq n < 2 \cdot 3^{k+1}. \end{cases}$$

The V-cost $c(Q, n, V)$ for $2 \leqq n < 6$ is the same as in Theorem 5. The root node of $Q(n, V)$ is degree 2 for $n = 2$, 4, 5, 7, 8, 13, 14, 15, 16 and 17, and it is degree 3 otherwise.

For values of V lying between $\frac{11}{7}$ and $\frac{30}{19}$ the degree of the root node of a V-optimal tree for quite a few values of n can be 2 or 3 (i.e., both are possible). As V increases beyond $\frac{30}{19}$ the root nodes of V-optimal trees for most n tend to switch to degree 2 from degree 3. In the interval $\frac{30}{19} < V < \frac{35}{22}$ it gets difficult to deduce a closed-form expression for $c(Q, n, V)$. This difficulty disappears as V attains the value $\frac{35}{22}$. It is not totally clear to us whether the methods of this section would yield expressions for $c(Q, n, V)$ when $\frac{30}{19} < V < \frac{35}{22}$ even if we break up the interval into a number of smaller subintervals, since we may need infinitely many such subintervals. The problem arises because for all but finitely many n, the root of a V-optimal tree has degree 3 when $V \leqq \frac{30}{19}$ and degree 2 when $V \geqq \frac{35}{22}$. Nor have we succeeded in evolving a method for showing $c(Q, n, V)$ is convex in n for a given V which does not require that we first get a closed-form expression for $c(Q, n, V)$. So whether $c(Q, n, V)$ is convex in n for V in the range $\frac{30}{19} < V < \frac{35}{22}$ is still open.

We now see what happens when $V \geqq \frac{35}{22}$.

THEOREM 7. *Let V in the range $\frac{35}{22} \leqq V < \frac{8}{5}$ be given. The V-cost $c(Q, n, V)$ of a V-optimal tree built on n equal unit weights is convex viewed as a function of n and has the following expression for $n \geqq 6$.*

$$c(Q, n, V) = \begin{cases} (n - 3 \cdot 2^k)(k + 8 - 3V) + 3 \cdot 2^k (V + k) & \text{for } 3 \cdot 2^k \leqq n < 2^{k+2}, \\ (n - 2^{k+2})[18V - (25 - k)] + (k + 2)2^{k+2} & \text{for } 2^{k+2} \leqq n < 9 \cdot 2^{k-1}, \\ (n - 9 \cdot 2^{k-1})(k + 7 - 2V) + 2^{k-1}(18V - 25 + k) & \\ \qquad + (k + 2)2^{k+2} & \text{for } 9 \cdot 2^{k-1} \leqq n < 3 \cdot 2^{k+1}. \end{cases}$$

The V-cost $c(Q, n, V)$ for $2 \leqq n < 6$ is the same as in Theorem 5. The root node of $Q(n, V)$ is degree 2 for all n except $n = 3$, 9, 10 and 11, for which it is degree 3.

When V lies between $\frac{8}{5}$ and $\frac{5}{3}$ in value, the root node of a V-optimal tree $Q(n, V)$ is degree 2 for all n except $n = 3$.

THEOREM 8. *Let V in the range $\frac{8}{5} \leqq V < \frac{5}{3}$ be given. The V-cost $c(Q, n, V)$ of a V-optimal tree built on n equal unit weights is convex viewed as a function of n and has the following expression for $n \geqq 2$.*

$$c(Q, n, V) = \begin{cases} (n - 2^{k+1})(3V + k - 2) + (k + 1)2^{k+1} & \text{for } 2^{k+1} \leqq n < 3 \cdot 2^k, \\ (n - 3 \cdot 2^k)(k + 8 - 3V) + 3 \cdot 2^k (V + k) & \text{for } 3 \cdot 2^k \leqq n < 2^{k+2}. \end{cases}$$

The root node of a V-optimal tree $Q(n, V)$ is degree 3 for $n = 3$ and is degree 2 otherwise.

As pointed out earlier, Theorems 4 through 8 have similar proofs. So we only give the proof of Theorem 8 below:

Proof of Theorem 8. The proof is by induction on n. The theorem clearly holds for $n = 2$ and $n = 3$. For $n \geqq 4$, we define, following Knuth [3, p. 372]

$$c_2(Q, n, V) = c\left(Q, \left\lfloor \frac{n}{2} \right\rfloor, V\right) + c\left(Q, \left\lfloor \frac{n+1}{2} \right\rfloor, V\right)$$

and

$$c_3(Q, n, V) = c\left(Q, \left\lfloor \frac{n}{3} \right\rfloor, V\right) + c\left(Q, \left\lfloor \frac{n+1}{3} \right\rfloor, V\right) + c\left(Q, \left\lfloor \frac{n+2}{3} \right\rfloor, V\right).$$

Since the expression for $c(Q, n, V)$ given in the statement of the theorem is known to be valid for $2 \leqq m < n$ with m substituted for n in the expression, we get by direct

evaluation

$$c_2(Q, n, V) = \begin{cases} (n - 2^{k+2})(3V + k - 2) + (k+1)2^{k+2} \\ \qquad\qquad \text{for } 2^{k+1} \leqq n/2 < 3 \cdot 2^k \\ (n - 3 \cdot 2^{k+1})(k + 8 - 3V) + 2^{k+1}(3V + k - 2) + (k+1)2^{k+2} \\ \qquad\qquad \text{for } 3 \cdot 2^k \leqq n/2 < 2^{k+2}; \end{cases}$$

and

$$c_3(Q, n, V) = \begin{cases} (n - 3 \cdot 2^{k+1})(3V + k - 2) + (k+1)3 \cdot 2^{k+1} \\ \qquad\qquad \text{for } 2^{k+1} \leqq n/3 < 3 \cdot 2^k \\ (n - 9 \cdot 2^k)(k + 8 - 3V) + 3 \cdot 2^k(3V + k - 2) + 3 \cdot 2^{k+1}(k+1) \\ \qquad\qquad \text{for } 3 \cdot 2^k \leqq n/3 < 2^{k+2}. \end{cases}$$

It can now be checked that the given expression for $c(Q, n, V)$ has the property that

$$c(Q, n, V) = n + c_2(Q, n, V) \leqq nV + c_3(Q, n, V).$$

Moreover, we have

$$c(Q, n, V) - c(Q, n-1, V) = \begin{cases} 3V + k - 2 & \text{for } 2^{k+1} < n \leqq 3 \cdot 2^k, \\ k + 8 - 3V & \text{for } 3 \cdot 2^k < n \leqq 2^{k+2}, \end{cases}$$

so that

$$c(Q, n, V) - c(Q, n-1, V) \geqq c(Q, n-1, V) - c(Q, n-2, V).$$

Hence, $c(Q, n, V)$ is convex at n. □

The other theorems can be proved in a similar manner. In particular, we need to show the following:

(i) In Theorem 4, for all $n \geqq 3$,

$$c(Q, n, V) = c_3(Q, n, V) + nV \leqq c_2(Q, n, V) + n;$$

(ii) In Theorem 5, for all $n \geqq 6$,

$$c(Q, n, V) = c_3(Q, n, V) + nV \leqq c_2(Q, n, V) + n;$$

(iii) In Theorem 6, for all $n \geqq 18$,

$$c(Q, n, V) = c_3(Q, n, V) + nV \leqq c_2(Q, n, V) + n;$$

(iv) In Theorem 7, for all $n \geqq 12$,

$$c(Q, n, V) = c_2(Q, n, V) + n \leqq c_3(Q, n, V) + nV.$$

In each case, for values of n smaller than the given bounds, the validity of the expression for $c(Q, n, V)$ must be verified by actual construction of a V-optimal tree, i.e., by reference to Fig. 11.

An open problem of critical importance in the equal weights case is whether $c(Q, n, V)$ is convex viewed as a function of n for V lying in the range $\frac{30}{19} < V < \frac{35}{22}$. When this question is answered the equal weights case will be completely solved. An interesting query is whether for any V in the interval $\frac{30}{19} < V < \frac{35}{22}$ the root node of a V-optimal tree can be degree 2 for infinitely many n and degree 3 for infinitely many other n.

Acknowledgment. The problem studied in this paper was suggested to the first author in a more general form some years ago by Professor C. L. Liu of the Department of Computer Science, University of Illinois.

REFERENCES

[1] A. BAGCHI AND J. K. ROY, *Bounds on cost ratios of optimal trees*, Symposium on Graph Theory, Indian Statistical Institute (Calcutta, December 20–25, 1976).

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

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

[4] M. SCHLUMBERGER AND J. VUILLEMIN, *Optimal disk merge patterns*, Acta Informatica, 3 (1973), pp. 25–35.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0005 $01.00/0

A NOTE ON LOCATING A SET OF POINTS IN A PLANAR SUBDIVISION*

F. P. PREPARATA†

Abstract. In this note we algorithmically show that a set of k points can be located in the planar subdivision induced by a straight-line planar graph with n vertices in time $O(k \log k)+O(n)+O(k \log n)$, given a preprocessing time $O(n \log n)$.

Key words. Computational geometry, computational complexity, point location, point set location

A planar straight-line graph (PSLG) G with n vertices induces a subdivision of the plane into regions which are simple polygons. Locating a given point P_0 (*target point*) in this subdivision means finding the subdivision region which contains P_0.

In this note we consider the problem of *collectively* locating a set $S=\{P_1, \cdots, P_k\}$ of points in the planar subdivision induced by G; a solution to this problem is relevant to finding the intersection of planar maps and the intersection of convex polyhedra [1]; indeed, a recent result on the latter problem [2] makes crucial use of the method to be presented. Notice that set S could be located in $O(kn)$ time by an obvious brute force method (with no additional preprocessing), and in $O(k \log^2 n)$[1] time by the Lee–Preparata algorithm (with $O(n \log n)$ preprocessing) [3]. We shall now describe a simple extension of the latter for locating a set of k points in time $O(k \log k)+O(n)+O(k \log n)$;[2] this new variant uses the same data structure given in [3], to which the reader is referred for a detailed description. We now just recall the essentials.

Let $\{v_1, \cdots, v_n\}$ be the vertex set of G, with ordinates $y(v_1) \geqq \cdots \geqq y(v_n)$. G is assumed to be *regular*, i.e. for each vertex $v_j (j \neq 1, n)$ of G there are integers $i<j<k$ such that (v_i, v_j) and (v_j, v_k) are edges of G (if G is not regular, it can be made so by adding edges in time $O(n \log n)$). In a regular graph G, one can find a set $\mathscr{C}=\{c_1, c_2, \cdots, c_m\}$ of polygonal lines consisting of edges of G, called *chains*, from v_1 to v_n, with the following properties:

(i) each edge of G belongs to at least one chain;
(ii) each chain $c \in \mathscr{C}$ is *monotone* with respect to the y-axis, i.e., if $(u_1, u_2, \cdots, u_p)$ is the sequence of vertices in c $y(u_1) \geqq y(u_2) \geqq \cdots \geqq y(u_p)$;
(iii) for any two chains c_i and c_j of $\mathscr{C}$, the vertices of c_i which are not vertices of c_j lie on the same side of c_j (the set $\mathscr{C}$ is ordered).

Clearly $c_1 \cup \cdots \cup c_m$ embeds G (see Fig. 1 for an example of a graph G and of the set of chains embedding it).

A location of a single point can be accomplished by a nested binary search. In fact, a (primary) binary search on $\mathscr{C}$ locates P_0 between two consecutive chains; by property (ii), for any $c \in \mathscr{C}$, a (secondary) binary search on the ordinates of the vertices of c locates P_0 on either side of c.

* Received by the editors September 19, 1977. This work was supported in part by the National Science Foundation under Grant MCS76-17321 and in part by the Joint Services Electronics Program under Contract DAAB-07-72-C-0259.

† Coordinated Science Laboratory, University of Illinois at Urbana. Also, Department of Electrical Engineering and of Computer Science.

[1] All logarithms in this note are to the base 2.

[2] After the original submission of this note, Lipton and Tarjan presented an algorithm [4] which locates a point in time $O(\log n)$ on a data structure which is also constructible in time $O(n \log n)$. On the point set location problem their method has a time performance comparable in the order to that of our approach, but is algorithmically far more complicated.

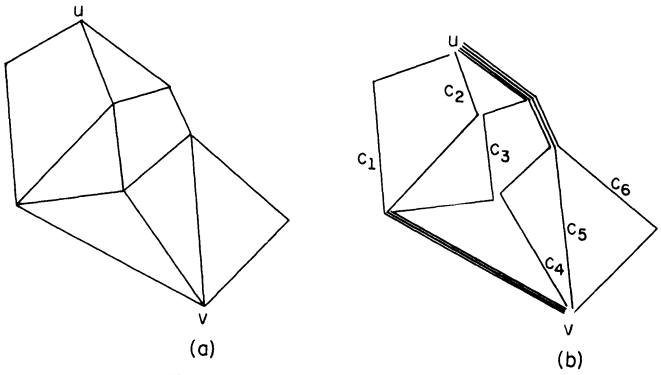


FIG. 1. *Examples of G and* $\mathscr{C}$.

The location procedure for a set of points is essentially based on the observation that the secondary binary search on the y-coordinates of the vertices of c can be replaced by a linear search, and that, for a set of points, the corresponding linear searches can be done concurrently as a single linear search (*after ordering the k points in S according to their y-coordinate*). The latter search would be straightforward, were it not for some complications due to the nature of the searchable data structure.

Indeed, we recall that the set $\mathscr{C}$ is organized as the set of nodes of a balanced rooted binary tree $T(\mathscr{C})$, whose paths from the root correspond to sequences of chains, in the order in which they are used when performing point locations. Thus each edge e of G—although possibly shared by several consecutive chains of $\mathscr{C}$—need be stored only *once* in the list of the chain which contains e and is closest to the root of $\mathscr{C}$. This achieves $O(n)$ storage; however the typical edge list of a chain appears as in Fig. 2, where the pointers "bypass" edges which have been assigned to other chains in $T(\mathscr{C})$.

The following procedure PARTITION (U, c) splits a sequence of points U into two sequences U' and U'' of points which lie respectively to the left and to the right of chain c. Notationally, $y'(e)$ and $y''(e)$ $(y'(e) \geqq y''(e))$ respectively denote the ordinates of the upper and lower extremes of an edge e; also, with each edge e we associate the pair

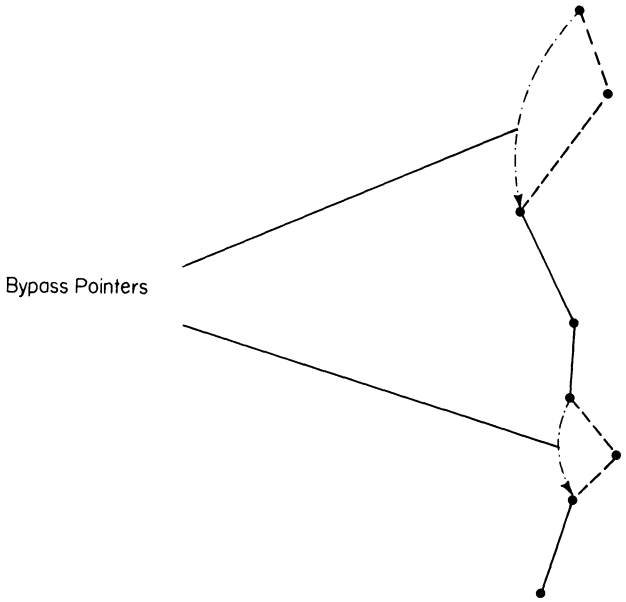


FIG. 2. *Illustration of typical edge list of a chain.*

of names ($L[e]$, $R[e]$) of the two regions sharing e, and the pair of integers ($I_{\min}[e]$, $I_{\max}[e]$), which are respectively the smallest and the largest indices of the chains containing e. With each point P to be located there is an associated triple of parameters ($R(P)$; $l(P)$, $r(P)$), where $R(P)$ is the region to which P is tentatively assigned, and $l(P)$ and $r(P)$, ($l(P) < r(P)$), are integers denoting that P lies between $c_{l(P)}$ and $c_{r(P)}$ in $\mathscr{C}$. We may also assume that, for any point P, $\max_{e \text{ in } G} y'(e) \geqq y(P) \geqq \min_{e \text{ in } G} y''(e)$.

Input: a list U: $(P_1, P_2, \cdots, P_t, P_{t+1})$ where $i < j \Rightarrow y(P_i) \geqq y(P_j)$; a list $c = (e_1, e_2, \cdots, e_s, e_{s+1})$ where $h < l \Rightarrow y''(e_h) \geqq y'(e_l)$, and an integer INDEX ($c$) P_{t+1} and e_{s+1} are dummy sentinels, with $y(P_{t+1}) = y'(e_{s+1}) = y''(e_{s+1}) = -\infty$.

Output: two lists U' and U'' of points.

1. $k \leftarrow i \leftarrow j \leftarrow 1$, $U' \leftarrow U'' \leftarrow \varnothing$.
2. **While** $k \leqq t + s$ **do**
3. **begin If** $y(P_i) > y'(e_j)$ **then**
4. **If** $l(P_i) \geqq$ INDEX (c) **then** $U'' \leftarrow U'' \cup \{P_i\}$, **else** $U' \leftarrow U' \cup \{P_i\}$
5. $i \leftarrow i + 1$
6. **else If** $y(P_i) < y''(e_j)$ **then** $j \leftarrow j + 1$
7. **else If** P_i lies to the right of e_j **then**
$U'' \leftarrow U'' \cup \{P_i\}$, $l(P_i) \leftarrow I_{\max}[e_j]$, $R(P_i) \leftarrow R[e_j]$
8. **else** $U' \leftarrow U' \cup \{P_i\}$, $r(P_i) \leftarrow I_{\min}[e_j]$, $R(P_i) \leftarrow L[e_j]$
9. $i \leftarrow i + 1$
10. $k \leftarrow k + 1$

end

11. **return** $\{U', U''\}$

Notice that a point P_i is assigned to one of the two sets U' or U'' either in steps 7–8 (by discriminating P_i against edge e_j) or in step 4, when $y''(e_j) > y(P_i) > y'(e_{j+1})$: in this case P_i lies in the horizontal strip corresponding to a bypass pointer (see Fig. 2), and the assignment to U' or U'' is governed by the parameters ($l(P_i)$, $r(P_i)$). It is easily realized that PARTITION (U, c) runs in time proportional to $|U| + |c|$, where $|c|$ is the number of the edges *actually* assigned to c in the construction of $T(\mathscr{C})$.

We can now describe the location procedure, where $T[c]$ denotes the subtree whose root is $c \in T(\mathscr{C})$. The symbol Λ denotes the empty tree.

LOCATE (S, T)

Input: S, T. For each $P \in S$, $l(P) = 0$, $r(P) = |T| + 1$, $R(P) = \Lambda$.

Output: a set $K = \{(P, R(P)) | P \in S, R(P) =$ a region of the subdivision containing $P\}$

1. **begin** $K \leftarrow \varnothing$
2. **If** $S \neq \varnothing$ **then**
3. **begin** $c \leftarrow$ ROOT (T)
4. $\{S', S''\} \leftarrow$ PARTITION (S, c)
5. **If** RIGHTSON (c) $\neq \Lambda$ **then** $K'' \leftarrow$ LOCATE (S'', T [RIGHTSON (c)])
6. **else** $K'' \leftarrow \{(P, R(P)) | P \in S''\}$
7. **If** LEFTSON (c) $\neq \Lambda$ **then** $K' \leftarrow$ LOCATE (S, T [LEFTSON (c)])
8. **else** $K' \leftarrow \{(P, R(P)) | P \in S'\}$
9. $K \leftarrow K' \cup K''$
10. **end**
11. **return** K
12. **end**

We now evaluate the performance of the described algorithm. The bulk of the computational work is performed in step 4, and we have already noted that PARTITION (S, c) runs in time $O(|S|+|c|)$. Since the algorithm entails at most one visit to each node of $T(\mathscr{C})$, it is convenient to "charge" the work to the individual nodes of $T(\mathscr{C})$. Specifically, let $S(c) \subseteq S$ be the set of points to be discriminated against chain c. Thus the global computational effort is

$$O\Big(\sum_{c \in T(\mathscr{C})} |S(c)|\Big) + O\Big(\sum_{c \in T(\mathscr{C})} |c|\Big);$$

but, by the construction of the data structure $T(\mathscr{C})$, $\sum_{c \in T(\mathscr{C})} |c|$ equals the number of edges of G, i.e., it is $O(n)$ due to the planarity of G. Moreover, since obviously $|S(\text{LEFTSON}\ (C))| + |S(\text{RIGHTSON}\ (c))| = |S(c)|$, at any given depth in $T(\mathscr{C})$ the sum of $|S(c)|$ is a constant and is equal to $|S| = k$. Since $T(\mathscr{C})$ has at most $\lceil \log_2 m \rceil$ levels, and m is $O(n)$, we conclude that $\sum_{c \in T(\mathscr{C})} |S(c)| = O(k \log n)$.

In summary,—excluding preprocessing of the search structure—the set of points can be collectively located in time $O(k \log k) + O(n) + O(k \log n)$, where the term $O(k \log k)$ is due to the initial sorting of the set $\{y(P_i) | P_i \in S\}$. The corresponding work for the algorithm of [3] is $O(k \log^2 n)$; a comparison of these two measures indicates that for a wide range of k (typically when $k = O(n)$) the present algorithm has a better worst-case performance.

REFERENCES

[1] M. I. SHAMOS, *Computational Geometry*, Dept. of Comp. Sci., Yale University, 1977. To be published by Springer-Verlag.

[2] D. E. MULLER AND F. P. PREPARATA, *Finding the intersection of two convex polyhedra*, Theoretical Computer Science, to appear; also available as Report ACT-6, Coordinated Science Lab., University of Illinois, Urbana, Illinois, November 1977.

[3] D. T. LEE AND F. P. PREPARATA, *Location of a point in a planar subdivision and its applications*, this Journal, 6 (1977), pp. 594–606.

[4] R. J. LIPTON AND R. E. TARJAN, *Application of a planar separator theorem*. Proc. of the 18th Symp. on Found. of Comp. Sci. (Providence, RI), 1977, pp. 162–170.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0006 $01.00/0

ON THE SEMANTICS OF "DATA TYPE"*

JAMES DONAHUE†

Abstract. This paper considers the general problem of specifying the meaning of programming languages that include "data type definition facilities." The fundamental question posed in attempting to define such languages is: "what meaning should be given to a data type definition," or more simply, "what does data type mean?" In this paper we describe a new approach to defining the meaning of data types, treating data types as values, and give its application to the definition of a typed lambda calculus extension. We also prove a theorem stating that our lambda calculus definition is "strongly typed."

Key words. Data types, semantics, polymorphism, lambda calculus

1. Introduction. A recent trend in programming languages is to allow definitions of data types in programs. Although there is currently much ferment about what form such definitions should take (cf., PASCAL (Jensen and Wirth [7]), Euclid (Lampson et al. [8]), and CLU (Schaffert [17]), there seems to be general agreement about their utility as a programming tool.

The introduction of type definitions requires that we have a clear idea of what we mean by "data type," so that we can know just what is being defined. A clear notion of "data type" is also important for understanding languages without type definition facilities. In attempting to give the semantics (using any kind of formalism) of a language with either a fixed or changeable set of data types, we somehow must give some meaning to the type identifiers and expressions appearing in programs.

This paper presents an approach to defining the meaning of "data type" in programming languages. We do not give the semantics of a language involving type definitions; how specific data type values are created is left unspecified. Instead, we focus on the interpretation of programs in which data type values are used. Indeed, given the lack of agreement as to how type definition should be done, we are well-advised not to choose any single form of type definition. We hope the reader will see how his favorite form of definition can be viewed using this approach.

The paper is organized as follows. In the next section we describe in informal terms our interpretation of the meaning of "data type," using examples from Algol 60. We then give a formal definition of a far simpler language, a version of the typed lambda calculus, to show how the approach allows a rigorous formulation of the meaning of data types. Finally, we prove a theorem showing that this simple language is "strongly typed." In doing so, we provide a semantic justification for the common use of "type-checking" to catch erroneous programs.

2. An informal view of data types. Our treatment of data type is a radical departure from the usual view of types as sets of values. Instead, we take a data type to be a set of operations specifying an interpretation of values. What such a set comprises depends on the particular programming language under discussion. Below, we will discuss informally the meaning of Algol 60 data types using this view; in the next section we will apply this same view to a formal treatment of a typed lambda calculus variant.

The notion of type as a set of operations specifying an interpretation (in the sense of "bringing out the meaning of") of values is drawn from the algebraic approach to abstract data types. One way of viewing the claim that "an abstract data type is the

* Received by the editor October 27, 1977.

† Computer Science Department, Cornell University, Ithaca, New York 14853. This work was partially supported by the National Science Foundation under Grant MCS76-14293.

isomorphism class of an initial algebra in a category of algebras" (Gougen et al. [3]) is that the meaning of an abstract data type is not embodied in the carrier set of its algebra, but in the effect of the operations of the algebra on elements of the carrier. Thus, we may focus our attention on the initial algebra, knowing that there exists a homeomorphism from it to any other algebra in the category, i.e., that the initial algebra captures the effect of the operations as well as any other algebra in the category.

We have generalized this algebraic approach to include a larger class of operations that are necessary to specify fully how values are interpreted in common programming languages. Consider, for example, a language like Algol 60 that has only primitive (nonstructured) data types. In such a language, we may store values in variables, extract values from variables and compose values by application of certain primitive functions. Our approach to data types is to take the meaning of this set of primitive operations over some value space as the meaning of a data type. As in the algebraic approach, we make no prior assumptions about the structure of the value space manipulated by these operations. The space may be a union of disjoint components or may be "typeless," like $\mathbb{P}\omega$ (Scott [18]) or the memory of most machines. We are interested only in how values are interpreted by operations.

So, in the following Algol 60 fragment

integer x, y;
$x := 0$;
$y := x$

the meaning (or denotation) of "integer" would provide the meaning of the following operations:

1. Value extraction, specifying how to interpret the values "stored" in variables,
2. Assignment, specifying how a value is used to "update" a variable, and
3. the evaluation of the nullary function 0, i.e., which value "represents" 0.

A similar, but less formal, treatment of data types can be found in the axiomatic definition of PASCAL (Hoare and Wirth [6]). In the rule of assignment,

$$\{P_x^y\} \quad x := y \quad \{P\},$$

if the type T of X is a subrange of the type of y, then P_x^y is replaced by $P_x^{T(y)}$ (where $T(y)$ is the restriction of y to a value of type T). In this case the meaning of the assignment $x := y$ is explicitly stated to be dependent on the type of x. All we are doing is formalizing this by saying that the meaning of a data type is the semantics it gives to assignment (i.e., the axiom $\{P_x^{T(y)}\} \quad x := y \quad \{P\}$).

There are two important reasons for adopting this approach to data types. Firstly, it allows us to side-step neatly the question of whether each value belongs to only one type (as hypothesized in (Hoare [5])). Values are essentially "meaningless" in this approach; it is only through the application of some operation that they are given an interpretation. Thus, while we do not rule out the possibility that each value has some distinguishing attribute called "type," we do not demand it. It is perfectly reasonable to assume that the same value may be interpreted many different ways. For example, the same sequence of bits in the memory of a machine is commonly interpreted as logical value, integer, floating point or piece of program depending on the operations used.

The second important characteristic of this approach is that data types themselves may be treated as values. If we choose a value space that is "universal" (Milne and Strachey, [12]), i.e., it is isomorphic to the space of operations over it, then data types (which are just sets of operations) are values in the space. This importance of treating data types as values is that we can give a simple interpretation to polymorphic definitions, i.e., definitions that have types as parameters. In the next section, we will

give a simple denotational semantics for a polymorphic lambda calculus using this approach.

The idea of treating data types as values also appears in Shamir and Wadge [19] and Scott [18]. However, unlike Shamir and Wadge, we do not add special "data type" values to form an extended domain, but simply use values of a particular form as data types. Thus, our approach is similar to that of Scott in [18]. In fact, in the formal semantics to follow, we will use "retracts" as the meaning of data type for our lambda calculus variant, as is done in Scott [18]. And we will give an intuitive motivation for this choice of the meaning of type in terms of our preceding discussion.

3. A lambda calculus extension and its semantics.

3.1. The language. The language we use as an example of this approach to giving meaning to data types is an extension of the typed lambda-calculus first suggested by Reynolds [16]. As in the basic typed lambda-calculus (Morris [13]), it includes the usual forms of abstraction and application. For example, $\lambda x \in t.x$ defines the identity function of type t ($\lambda x \in t.x$ has type $t \to t$), and if y has type t, ($\lambda x \in t.x \quad y$) denotes the application of the identity function to the argument y (yielding y). Additionally, we extend the language in two ways:

1. If T is a type, then $T\$c$ will be used to denote the *constant* c of type T. The syntactic form of constants will be left unspecified to shorten the presentation.
2. Just as we allow normal abstraction and application, so we allow *type* abstraction and application. For example, given the type t identity function $\lambda x \in t.x$, we can abstract it with respect to the *type identifier* t by writing $\Lambda t.\lambda x \in t.x$ to produce the *polymorphic* ("of many types") identity function (of type $\Delta t.t \to t$). From $\Lambda t.\lambda x \in t.x$ we can produce the identity function of any particular type by applying our polymorphic function to the type. Thus,

$$\Lambda t.\lambda x \in t.x \text{ [integer]}$$

produces the integer identity function as its result.

Below, we give a complete formal description of the language using the denotational approach to semantics.

3.2. Formal semantics. We begin by giving a formal syntax for the language (using identifiers beginning with a capital letter to denote nonterminals). *Exp*, the set of lambda expressions, is defined as follows:

$Exp ::= Id\$Const$	constants of type Id (Const will be left unspecified)
$\mid Id$	normal variables
$\mid \lambda Id \in W.Exp$	normal lambda abstraction (where W is a type expression, as defined below)
$\mid (Exp \quad Exp)$	normal application
$\mid \Lambda Id.Exp)$	type abstraction
$\mid Exp\ [W]$	type application

W, the set of type expressions, is defined by:

$W ::= Id$	type identifiers
$\mid W \to W$	functional types (the types of λ terms)
$\mid \Delta Id.W$	polymorphic types (the types of Λ terms)

We give formal semantics to the language by defining a function **Me** mapping each lambda expression to its meaning as an element of some domain. As in treatments of other forms of the lambda calculus, we will take the meaning of expressions as elements of the domain

$$D = B + [D \to D] + [T \to D]$$

where B is a domain of primitive or basic values (which we assume will be used to interpret constants) and T is the domain of type denotations. (One can think of $D \to D$ values as the meanings of λ-terms, $T \to D$ values as the meanings of Λ-terms.) Obviously, the heart of the matter is the definition of T, and it is here that we apply our previously stated view of data types.

The most basic operation in the lambda calculus is clearly the evaluation of the basic expressions, i.e., identifiers and constants. For the untyped lambda calculus, the usual interpretation of the meaning of a variable x is its value in the current environment, a member of the domain $Id \to D$. However, in the typed calculus, the value produced by evaluating x in the current environment must be consistent with the type associated with x. One approach to satisfying this requirement in the semantics is to say that the environment must be "type-respecting," i.e., that the value associated with x in the environment must be compatible with the type of x. (This approach is adopted in Reynolds [16].)

Our approach is to take as the meanings of data types the interpretations of "taking the value of an identifier in an environment." Thus, we take T to be functions of type $D \to D$ such that for any environment e, if x has type t, then the "meaning of t" applied to $e(x)$ produces a "type-respecting" value. For the basic data types in the language, the meaning of the data type will also include a component of type $Const \to B$ giving the meaning of constants of that type. (Since we have included no data type definition facility in the language, how constants are given denotations is left unspecified.)

This interpretation of data types can be informally explained in terms of the operation of an SECD machine (Landin [9]) evaluation of lambda expressions. In an SECD definition of the lambda calculus, the meaning of "taking the value of a variable" is to push its current value in the environment onto the stack. Now consider the implementation of the stack on some machine of fixed word length. The action of the machine will involve:

1. allocating a number of words of storage (the number may differ for various types) on the top of S to hold the value to be copied, and
2. filling this space with the current value of the variable in E.

Our use of a function of type $D \to D$ as the meaning of "data type" can be viewed as simply the denotational encoding of this action of an SECD machine, i.e., as giving the interpretation of any value in D as an element of the type.

This interpretation of the meaning of data types points out one further aspect of our semantics worthy of note. It is natural at first glance to view the $D \to D$ component of types that introduce constants (i.e., types in $[Const \to B] \times [D \to D]$) as simply restrictions of the identity function. For example, if we decide to define the Boolean constants **true** and **false** by the integers 1 and 0 respectively, a natural choice for the second component of the meaning of Booleans would be

$$\lambda x \in D.\ x \textbf{ is } \mathit{Integer} \textbf{ then } (x > 1 \textbf{ then } \perp \textbf{ else } x) \textbf{ else } \perp$$

This choice can be viewed as choosing a single-bit representation of Booleans and checking to make sure one never moves nonprimitive values (functions, for example)

onto the stack. When viewed in this light other possibilities suggest themselves. For example,

$$\lambda x \in D.x \textbf{ is } \textit{Integer} \textbf{ then } (x > 255 \textbf{ then } \perp \textbf{ else } x) \textbf{ else } 255$$

suggests the choice of eight bits to represent Booleans and less care in ruling out the assignment of nonprimitive values to Boolean variables. In fact, all we really require is that the interpretations of constants (given by the first component of a primitive type) remain unaltered by the application of the $D \to D$ (second) component of the data type. In more formal terms, we require that these functions from D to D be "retracts," i.e., functions f such that $f = f \circ f$, with range including at least the denotations of all constants introduced by the type. And clearly this restriction makes sense in terms of the SECD analogy; it simply says that when allocating space on the stack, we always allocate enough space for the denotation of any constant of the type.

Thus, we now give a formal definition of the meaning of "data type" for the polymorphic lambda calculus.

DEFINITION. A function $f: D \to D$ is a *retract* iff $f = f \circ f$. (Note that all elements of the range of f are fixed points of f, i.e., $x = f(x)$.) An element $t \in [\textit{Const} \to B] \times [D \to D]$ is a *data type* iff:

1. $[t]_2$ (the second component) is a doubly strict retract (a function f is doubly strict iff $f(\perp) = \perp$ and $f(T) = T$), and
2. $\forall c \in \textit{Const}$, $[t]_1[\![c]\!]$ is a fixed point of $[t]_2$, i.e., the meanings of all constants are preserved by the second component of the type.

An element $f \in D \to D$ is a *data type* iff f is a doubly strict retract.

Given this restriction on data types, we can now give a simple characterization of what it means for an element of D to be a member of a data type.

DEFINITION. An element $x \in D$ is said to be *an element of a data type* t iff x is a fixed point of $\lambda x \in D.\text{Apply } (t, x)$, i.e., x in the range of $\lambda x.\text{Apply } (t, x)$, where Apply: $[T \times D] \to D$ is defined by

$$\text{Apply } (t, d) = t \textbf{ is } D \to D \textbf{ then } t(d) \textbf{ else } [t]_2(d).$$

Several points need to be made about these definitions. First, we will impose no requirement that each element of D belong to a unique data type. We are using our data types to impose a type structure on the basically "typeless" domain D, and it is quite natural to assume that elements of D will serve as elements of many data types. In fact, because of our requirement that these retracts be doubly strict, $\perp$ and $\top$ will be elements of every data type.

Second, for the basic data types (those giving meaning to constants), we require only that the denotations of constants be elements of the data type, not that these denotations be *the only* elements of the data type. The weakness of the assumption can be viewed as allowing an implementor to choose any representation for the data type he finds appropriate, as long as all of the constants can be represented satisfactorily.

Now, given this basic decision about the meaning of data types, we are now ready to "lay our domains on the table" to specifying the complete semantic structure of the language. Informal commentary follows the formal description.

Basic Domains

Exp	as above
W	as above
$D = B + [D \to D] + [T \to D]$	expression denotations
$T = [D \to D] + [[\textit{Const} \to B] \times [D \to D]]$	data types

Meaning functions

Me: $Exp \to Q \to Te \to Env \to D$	meaning of expressions
where	
$Q = Id \to W$	syntactic type environment
$Te = Id \to T$	semantic meaning of type identifiers
$Env = Id \to D$	meaning of free identifiers
Mt: $W \to Te \to T$	meaning of types

The meaning of an expression (the particular element in $D = B + [D \to D] + [T \to D]$ it denotes) depends in several ways on the free identifiers appearing in the expression; the domains Q, Te, and Env are used to capture these various dependencies. Q is used to provide the "syntactic type" (element of W) associated with each free identifier. For example, inside the body of $\lambda x \in t.exp$, x "has type t", i.e., the element of Q used to interpret exp will associate t with x. The "type environment" Te provides the mechanism for giving a meaning (as elements of T) to these syntactic types of providing the element of T associated with each free *type* identifier. Finally, the domain Env associates a value in D with each free normal identifier in the expression. Before giving the definitions of **Mt** and **Me**, we first must define an auxiliary function TypeOf which produces the "syntactic type" of every expression:

TypeOf: $Exp \to Q \to W$

is defined by

TypeOf $[\![id\$const]\!](q) = id$
The type of a constant is determined by the constant.

TypeOf $[\![id]\!](q) = q[\![id]\!]$
The type of an identifier is found in q.

TypeOf $[\![(exp_1\ exp_2)]\!](q) = \text{Range}(\text{TypeOf}[\![exp_1]\!](q))$
where

Range: $W \to W$ and
Dom: $W \to W$

extract the range and domain of type expressions of the form $W_1 \to W_2$, i.e.,

Range $[\![W_1 \to W_2]\!] = W_2$ and
Dom $[\![W_1 \to W_2]\!] = W_1$

Note that in specifying the *syntactic* type of a normal application, all we demand is that the type of the left-hand expression be some functional type; we do not require argument/parameter type correspondence. Expressions where such mismatches occur will have a proper (i.e., defined) type, but improper semantics.

$$\text{TypeOf}[\![\lambda id \in w.exp]\!](q) = w \to \text{TypeOf}[\![exp]\!](q[id \leftarrow w])$$

where $q[id \leftarrow w]$ is $\lambda id'.id' = id$ **then** w **else** $q[\![id]\!]$. (This notation for "updating" a function will be used extensively in what follows.)

Normal abstractions are of functional data types.

Type Of$[\![\Lambda id.exp]\!](q) = \Delta id.\text{TypeOf}[\![exp]\!](\text{Replace}\ (q, id, \perp))$ where

$$\text{Replace}\ (q, id, w) = \lambda id'.q[\![id']\!]\Big|_{id}^{w}$$

i.e., all occurrences of the type identifier *id* in q are replaced by w. This use of Replace is necessary to reflect the fact that previous bindings of identifiers to type expressions involving t are no longer valid inside the body of $\Lambda t.exp$.

$$\text{TypeOf } [\![exp[w]]\!](q) = \text{Body } (\text{TypeOf } [\![exp]\!](q))|^{w}_{\text{BV}(\text{TypeOf } [\![exp]\!](q))}$$

where Body: $W \to W$ and BV: $W \to Id$ extract the body and bound variable of type expressions of the form $\Delta id.w$.

One aspect of our treatment of syntactic types, as specified by TypeOf, is worthy of further comment. That is, the expression $\lambda f \in t \to t.\lambda x \in t.(fx)$ always has syntactic type $(t \to t) \to (t \to t)$ independent of surrounding context. Thus, for example, the expression

$$\Lambda t.\lambda f \in t.\lambda x \in t'.(f\,x)[t' \to t'']$$

has improper type because f is not of some functional type, even though normal beta-reduction would reduce this to

$$\lambda f \in t' \to t''.\lambda x \in t'.(f\,x)$$

which has a proper type. In implementation-oriented terms, this restriction imposed by TypeOf can be viewed as allowing us to "compile" a polymorphic procedure by being able to typecheck its body independently of any type arguments to which the polymorphic procedure may be applied. Note that the desired effect of the above expression could be achieved by the correctly typed expression

$$\Lambda t_1.\Lambda t_2.\lambda f \in t_1 \to t_2.\lambda x \in t_1.(f\,x)[t'][t''],$$

which can be shown to be semantically equivalent to

$$\lambda f \in t' \to t''.\lambda x \in t'.(f\,x).$$

We first present the clauses of the meaning function **Mt**, which give a meaning to data type expressions (remember **Mt** has functionality $W \to Te \to T$).

$$\mathbf{Mt}[\![id]\!](te) = te[\![id]\!]$$

$$\mathbf{Me}[\![w_1 \to w_2]\!](te) = \text{Arrow } (\mathbf{Mt}[\![w_1]\!](te), \mathbf{Mt}[\![w_2]\!](te))$$

Arrow: $[T \times T] \to T$ is defined by

$$\text{Arrow}(t_1, t_2) = \lambda f \in D.(\lambda x \in D.\text{Apply } (t_2, x)) \circ f|D \to D \circ (\lambda x \in D.\text{Apply } (t_1, x))$$

where $f_2 \circ f_1$ is normal function composition, i.e., $(f_2 \circ f_1)(x) = f_2(f_1(x))$

The meaning of functional types is to map their arguments to "type respecting" functions, i.e., functions that accept values of type w_1 and produce results of type w_2.

$$\mathbf{Mt}[\![\Delta id.w]\!](te) = \text{Delta } (\lambda t \in T.\mathbf{Mt}[\![w]\!](te[id \leftarrow t]))$$

where

Delta: $[T \to T] \to T$ is defined by

$$\text{Delta } (y) = \lambda x \in D.\lambda t \in T.\text{Apply } (y(t), (x|T \to D)(t))$$

The meaning of polymorphic types is to take polymorphic values to "type-respecting" polymorphic values.

It is clear that **Mt** produces elements of T; however, for us to justify it as a meaning function for "data type expressions," we must prove that it maps retracts to retracts.

The following theorem shows that **Mt** in fact produces "data types" as results.

THEOREM 1. *If* $\forall id \in Id$, $te[\![id]\!]$ *is a data type, then for all* $w \in W$, $\mathbf{Mt}[\![w]\!](te)$ *is a data type.*

Proof. The proof is by induction on the structure of w.

Basis. $w = id \in Id$. Then $\mathbf{Mt}[\![w]\!](te) = te[\![w]\!]$ and $te[\![w]\!]$ is a data type by assumption.

Induction. There are two cases: 1. $w = w_1 \to w_2$. Then $\mathbf{Mt}[\![w]\!](te) =$ Arrow $(\mathbf{Mt}[\![w_1]\!](te), \mathbf{Mt}[\![w_2]\!](te))$. From the induction hypothesis, we have that both $\mathbf{Mt}[\![w_1]\!](te)$ and $\mathbf{Mt}[\![w_2]\!](te)$ are data types. So we need to show that for any data types t_1 and t_2, Arrow (t_1, t_2) is a data type (a doubly strict retract), i.e., that

$$\text{Arrow}\,(t_1, t_2) = \text{Arrow}\,(t_1, t_2) \circ \text{Arrow}\,(t_1, t_2).$$

But

$$\text{Arrow}\,(t_1, t_2)(f) = \lambda x \in D.\text{Apply}(t_2, x) \circ f|D \to D \circ \lambda x \in D.\text{Apply}\,(t_1, x)$$

and because t_1, t_2 are data types

$$= (\lambda x \in D.\text{Apply}\,(t_2, x) \circ \lambda x \in D.\text{Apply}\,(t_2, x)) \circ f|D \to D$$
$$\circ \lambda x \in D.\text{Apply}\,(t_1, x) \circ \lambda x \in D.\text{Apply}\,(t_1, x))$$

and simply collecting terms

$$= (\text{Arrow}\,(t_1, t_2) \circ \text{Arrow}\,(t_1, t_2))(f).$$

Q.E.D.

2. $w = \Delta id.w'$. Then $\mathbf{Mt}[\![w]\!](te) = \text{Delta}\,(\lambda t \in T.\mathbf{Mt}[\![w']\!](te[id \leftarrow t]))$
From the induction hypothesis, we have that $\lambda t \in T.\mathbf{Mt}[\![w']\!](te[id \leftarrow t])$ maps data types to data types, so we need to show that if $y \in T \to T$ maps data types that

$$\text{Delta}\,(y) = \text{Delta}\,(y) \circ \text{Delta}\,(y).$$

But

$$\text{Delta}\,(y)(x)(t) = \text{Apply}\,(y(t), (x \mid T \to D)(t))$$

and if $y(t)$ is a data type

$$= \text{Apply}\,(y(t), \text{Apply}\,(y(t), (x \mid T \to D)(t)))$$
$$= (\text{Delta}\,(y) \circ \text{Delta}\,(y))(x)(t).$$

Q.E.D. Theorem 1.

We are finally ready to give the clauses of the definitions of our meaning functions **Me**, which gives the meaning of lambda-calculus expressions

$$\mathbf{Me}[\![id\,\$const]\!](q)(te)(e) = te[\![id]\!]\ \mathbf{is}\ [Const \to B] \times [D \to D]\ \mathbf{then}\ [te[\![id]\!]]_1[\![const]\!]$$
$$\mathbf{else}\ \top.$$

If *id* is a type for which constants are defined, produce as value the meaning of the constant (i.e., the value of the first component of the meaning of the type applied to the constant); otherwise, the expression is erroneous.

$$\mathbf{Me}[\![id]\!](q)(te)(e) = \text{Apply}\,(\mathbf{Mt}[\![\text{TypeOf}\,[\![id]\!](q)]\!](te), e[\![id]\!])$$

The meaning of an identifier is simply its value in the current environment applied to the meaning of the type of the identifier.

$$\mathbf{Me}[\![(exp_1\ exp_2)]\!](q)(te)(e) =$$
$$\text{TypeOf}\,[\![exp_2]\!](q) \simeq \text{Dom}\,(\text{TypeOf}\,[\![exp_1]\!](q))\ \textbf{then}$$
$$(\mathbf{Me}[\![exp_1]\!](q)(te)(e)\,|\,D \to D)\ (\mathbf{Me}[\![exp_2]\!](q)(te)(e))$$
$$\textbf{else}\ \bot$$

where $w_1 \simeq w_2$ is true iff w_1 is alpha-convertible to w_2 and $x\,|\,D \to D$ restricts $x \in D$ to elements of $D \to D$, i.e., $x\,|\,D \to D$ is

$\bot$ if $x = \bot$
$\top$ if $x = \top$ or $x \notin D \to D$
z if $x \in D \to D$

The meaning of an application is to produce a proper result only when the type of the argument is consistent with the type of the parameter.

$$\mathbf{Me}[\![\lambda id \in w.exp]\!](q)(te)(e) =$$
$$\lambda y \in D.\mathbf{Me}[\![exp]\!](q[id \leftarrow w])(te)(e[id \leftarrow y])$$

$$\mathbf{Me}[\![\Lambda id.exp]\!](q)(te)(e) =$$
$$\lambda t \in T.\mathbf{Me}[\![exp]\!]\ (\text{Replace}\ (q, id, \bot))(te[id \leftarrow t])(e)$$

$$\mathbf{Me}[\![exp\ [w]]\!](q)(te)(e) =$$
$$(\mathbf{Me}[\![exp]\!](q)(te)(e)\,|\,T \to D)(\mathbf{Mt}[\![w]\!](te))$$

The meaning of type application is simply to apply the meaning of the expression (as an element of $T \to D$) to the meaning of the type (an element of T).

Two points of interest in the semantics above should be noticed:

1. Our treatment of data type abstraction and application is a straightforward extension of the treatment of normal abstraction and application in the lambda calculus. In particular, we avoid the "serious lacuna" of Reynolds's semantics for this same language, where the meaning given to polymorphic types may not be elements of a domain (unlike our choice of T). Moreover, his earlier semantics requires the use of category theory to give the meaning of functional and polymorphic types, a mathematical complication we avoid.

2. When read operationally, these meaning functions are consistent with common implementations of statically typed programming languages (ignoring the extra complexities caused by polymorphism). We have already described our interpretation of data types in terms of an SECD machine. The other point to note in this regard is the treatment of parameter-passing and type-checking in the semantics. The meaning of applications involves a simple *syntactic* check that the type of the argument is consistent with the type of the parameter. This sort of type-checking has been described in many other places (Morris [13], Ledgard [10], Gannon and Horning [1]) and is readily contrasted with the checking of the types of values found in "dynamically typed" languages like GEDANKEN (Reynolds [15], Tennent [21]). Also the interpretation of λ-abstraction involves no test of the "acceptability" of the parameter in the body of the abstraction. This "blind faith" in the validity of an argument is precisely the reason that "compile time" type checking is performed in statically typed languages.

We end the presentation of the semantics by proving that our meaning function **Me** is "correctly typed," i.e., the meaning of an expression is an element of the meaning of the type of the expression.

THEOREM 2. $\forall q \in Q$, $\forall te \in Te$, $\forall e \in Env$, *if* $\forall id \in Id$, $te[\![id]\!]$ *is a data type, then*

$$\mathbf{Me}[\![exp]\!](q)(te)(e)$$

is an element If data type

$$\mathbf{Mt}[\![\text{TypeOf}\,[\![exp]\!](q)]\!](te),$$

i.e., the semantics is "correctly typed."

Proof. The proof is by induction on the structure of exp.

Basis. There are two cases: 1. $exp = id\$const$. Then

$$\mathbf{Me}[\![exp]\!](q)(te)(e) = te[\![id]\!] \textbf{ is } [Const \to B] \times [D \to D]$$

$$\textbf{then } (te[\![id]\!])_1[\![const]\!] \textbf{ else } \top.$$

If $[te[\![id]\!]]_1[\![const]\!]$ is defined, then the theorem is true because $te[\![id]\!]$ is a data type; otherwise, $\top$ is an element of every data type.

2. $exp = id$. Then

$$\mathbf{Me}[\![\text{exp}]\!](q)(te)(e) = \text{Apply } (\mathbf{Mt}[\![\text{TypeOf}\,[\![id]\!](q)(te), e[\![id]\!])$$

and the theorem is true because

$$\mathbf{Mt}[\![\text{TypeOf}\,[\![id]\!](q)]\!](te) \quad \text{is a data type by Theorem 1.}$$

Induction. There are four subcases: 1. $exp = \lambda id \in w.exp'$. Then $\mathbf{Me}[\![exp]\!](q)(te).(e) = \lambda y \in D.\mathbf{Me}[\![exp']\!](q[id \leftarrow w])(te(e[id \leftarrow y])$, To show that this function is an element of type $\mathbf{Mt}[\![w \to \text{TypeOf}[\![exp]\!](q[\![id \leftarrow w]\!](te)$, we need the following lemma.

LEMMA. $\mathbf{Me}[\![exp']\!](q[id \leftarrow w])(te)(e[id \leftarrow y]) = \mathbf{Me}[\![exp']\!](q[id \leftarrow w])(te)(e[id \leftarrow \text{Apply } (\mathbf{Mt}[\![w]\!](te), y)])$.

Proof of Lemma. That this is true can be seen from the fact that $\mathbf{Mt}[\![w]\!](te)$ is a data type (and thus a retract) and that if *id* appears in an expression in exp', its meaning will always be an element of $\mathbf{Mt}[\![w]\!](te)$.

Then, given this lemma, we have that

$$\begin{aligned}
&\mathbf{Me}[\![exp]\!](q)(te)(e) \\
&\quad = \lambda y \in D.\mathbf{Me}[\![exp']\!](q[id \leftarrow w])(te)(e[id \leftarrow y]) \\
&\quad = \lambda y \in D.\mathbf{Me}[\![exp']\!](q[id \leftarrow w])(te)(e[id \leftarrow y]) \\
&\qquad \circ \lambda x \in D.\text{Apply } (\mathbf{Mt}[\![w]\!](te), x)
\end{aligned}$$

and from the application of the induction hypothesis and our previous theorem

$$\begin{aligned}
&= \lambda x \in D.\text{Apply } \mathbf{Mt}[\![\text{TypeOf}[\![exp']\!](q[id \leftarrow w])]\!](te), x) \\
&\quad \circ \lambda y \in D.\mathbf{Me}[\![exp']\!](q[id \leftarrow w])(te)(e[id \leftarrow y]) \\
&\quad \circ \lambda x \in D.\text{Apply } \mathbf{Mt}[\![w]\!](te), x).
\end{aligned}$$

Finally, from the definition of **Mt**, it is clear that this value is an element of

$$\mathbf{Mt}[\![w \to \text{TypeOf}\,[\![exp]\!](q[id \leftarrow w])]\!](te);$$

thus the theorem is true.

2. $exp = (exp_1\ exp_2)$. Then

$$\begin{aligned}
&\mathbf{Me}[\![exp]\!](q)(te)(e) = \\
&\quad \text{TypeOf}\,[\![exp_2]\!](q) \simeq \text{Dom } (\text{TypeOf}\,[\![exp_1]\!](q)) \textbf{ then} \\
&\qquad (\mathbf{Me}[\![exp_1]\!](q)(te)(e) \,|\, D \to D)(\mathbf{Me}[\![exp_2]\!](q)(te)(e)) \\
&\quad \textbf{else } \top.
\end{aligned}$$

The error case (if the type-checking fails) is taken care of by the doubly strict nature of data types. Otherwise, the simple fact that exp_1 is of a functional type, i.e., TypeOf $[\![exp_1]\!](q) = w_1 \to w_2$ is sufficient to guarantee the truth of the theorem. By the induction hypothesis, if TypeOf $[\![exp_1]\!](q) = w_1 \to w_2$, then $\mathbf{Me}[\![exp_1]\!](q)(te)(e)$ is an element of type $\mathbf{Mt}[\![w_1 \to w_2]\!](te)$. And from this, we know that **for any** $d \in D$, $(\mathbf{Me}[\![exp_1]\!](q)(te)(e) \mid D \to D)(d)$ is an element of data type $\mathbf{Mt}[\![w_2]\!](te)$, as required by the theorem.

The cases of type abstraction and application are straightforward and are left to the reader. Q.E.D.

An important aspect of the preceding theorem is that its truth is independent of the type-checking clause introduced in the meaning of normal applications. One is tempted to cite this as a failing of the semantics (after all, type-checking shouldn't be superfluous). But this property is symptomatic of the dangers of using an "untyped" domain to give the semantics of a language (whether it be the domain D defined above or 32-bit words); function applications may always produce results that look reasonable, even though the argument to which the function is applied was not reasonable. However, as we show below, this syntactic type-checking is important is guaranteeing the representation independence we desire.

3.3. A "strong typing" theorem. Although the type-checking clause in the rule for normal applications does not affect the type-correctness of the semantics (as was shown above), it does allow us to prove a stronger theorem asserting that the semantics has a degree of "representation independence." The sort of representation independence we desire is the following.

Let us assume that for some expression **exp** in our language, we have that:

1. TypeOf $[\![\mathbf{exp}]\!](q) = t$ for some $q \in Q$ and
2. when t is bound to some basic type (i.e., a type for which constants are defined), we have that $\mathbf{Me}[\![\mathbf{exp}]\!](q)(te)(e) = \mathbf{Me}[\![t\$const]\!](q)(te)(e)$ for some $const \in Const$.

If this is true for some particular choice of a meaning for t, then it should be true for all choices of the meaning of t (or any other data type).

If we ignore the type-checking clause of the role for normal application, then this property need not be true, i.e., we could produce "representation-dependent" results. These results could arise because of two aspects of our treatment of basic data types, i.e., ones introducing the meaning of constants.

First, we have simply required that the meaning of constants of a basic type of elements of the type, i.e., be part of the "representation" of the type. These values need not be the only values of the type, however. Thus, for example, the meaning of the expression

$$(\lambda x \in \text{Boolean}.x\ \text{Integer\$0})$$

(which has syntactic type Boolean) some cases might have a nonerroneous value (not $\bot$ or $\top$) that is not the denotation of any Boolean constant.

Also, we have not assumed that the representation of different basic data types are disjoint. Thus, the meaning of

$$(\lambda x \in \text{Integer}.x\ \text{Boolean\$true})$$

(which has type Integer) could have the meaning of any of several Integer constants depending on the meaning of the Boolean constant true.

1. If Boolean\$true and Integer\$0 denote the same value, then the meaning of the above expression is Integer\$0.

2. If Boolean$true and Integer$1 denote the same value (in a fashion similar to PL/I), now the meaning of the expression becomes Integer$1.

(Note that one can achieve precisely this effect in PL/I using EXTERNAL procedures.) We now prove that our semantics disallows the possibility of producing such spurious results.

To prove this, we first define a notion of "similar data types," i.e., data types that are structurally similar and may be freely substituted one for another.

DEFINITION. Two data types t and t' are *similar* iff:

1. $t, t' \in [Const \to B] \times [D \to D]$, i.e., both allow the introduction of constants, or
2. $t = \text{Arrow}(t_1, t_2)$ and $t' = \text{Arrow}(t_1', t_2')$ and t_1 is similar to t_1' and t_2 is similar to t_2', or
3. $t = \text{Delta}(y)$ and $t' = \text{Delta}(y')$ and for all t_1, t_1' t_1 is similar to t_1' implies $y(t_1)$ is similar to $y'(t_1')$.

LEMMA. *If* $\forall id \in Id$, $te[\![id]\!]$ *is similar to* $te'[\![id]\!]$, *then* $\forall w \in W$, $\mathbf{Mt}[\![w]\!](te)$ *is similar to* $\mathbf{Mt}[\![w]\!](te')$.

Proof. The proof (which is by structural induction on w) is omitted.

Now, we define a representation relation between elements of D stating when d and d' are representations of the same "abstract value" according to similar types t and t'.

DEFINITION. Let $\mathbf{Rep}(t, t'): D \longrightarrow\!\!\!\!\!\times\ D$ be a relation in $(D \times D)$ for similar data types t and t' such that $\mathbf{Rep}(t, t'): d \mapsto d'$ iff d is an element of t, d' an element of t' and either:

1. $t, t' \in [Const \to B] \times [D \to D]$ and $\exists c \in Const$ such that $[t]_1[\![c]\!] = d$ and $[t']_1[\![c]\!] = d'$, i.e., d and d' are denotations given to the same constant, or
2. $\mathbf{Rep}(\text{Arrow}(t_1, t_2), \text{Arrow}(t_1', t_2'))$: $d \mapsto d'$ iff $\forall d_1, d_1'$ $\mathbf{Rep}(t_1, t_1')$: $d_1 \mapsto d_1'$ implies $\mathbf{Rep}(t_2, t_2')$: $d(d_1) \mapsto d'(d_1')$, or
3. $\mathbf{Rep}(\text{Delta}(y), \text{Delta}(y'))$: $d \mapsto d'$ iff $\forall$ *similar* t_1, t_1' $\mathbf{Rep}(y(t_1), y'(t_1'))$: $d(t_1) \mapsto d'(t_1')$.

And now, we can prove that **Me** preserves **Rep**.

THEOREM 3. *If we have,* (a) $\forall id \in Id$, $te[\![id]\!]$ *similar to* $te'[\![id]\!]$, *and*

(b) $\forall id$ $\mathbf{Rep}(\mathbf{Mt}[\![q[\![id]\!]]\!](te), \mathbf{Mt}[\![q[\![id]\!]]\!](te'))$: $e[\![id]\!] \mapsto e'[\![id]\!]$, *i.e., each identifier in* e, e' *refer to the same "abstract value," then* $\forall exp \in Exp$,

$$\mathbf{Rep}(t, t'): \mathbf{Me}[\![exp]\!](q)(te)(e) \mapsto \mathbf{Me}[\![exp]\!](q)(te')(e')$$

where

$$t = \mathbf{Mt}[\![\text{TypeOf}\ [\![exp]\!](q)]\!](te) \text{ and}$$

$$t' = \mathbf{Mt}[\![\text{TypeOf}\ [\![exp]\!](q)]\!](te').$$

Proof. The proof is by structural induction on *exp*. Most cases are simple applications of the definitions of **Rep** and **Me** and the preceding theorems (to show that **Me** produces values in t and t'). The only interesting case is normal application, where we have that if $\text{exp} = (\text{exp}_1\ \text{exp}_2)$, then

$$\mathbf{Rep}(\text{Arrow}(t_1, t_2), \text{Arrow}(t_1', t_2')): \mathbf{Me}[\![exp_1]\!](q)(te)(e) \mapsto$$
$$\mathbf{Me}[\![exp_1]\!](q)(te')(e')$$

from the induction hypothesis and the fact that $\text{TypeOf}\ [\![exp_1]\!](q) = w_1 \to w_2$ for some w_1 and w_2. Now, for our definition of **Rep** to assure us that

$$\mathbf{Rep}(t_2, t_2'): \mathbf{Me}[\![exp]\!](q)(te)(e) \mapsto \mathbf{Me}[\![exp]\!](q)(te')(e'),$$

we must have

$$\mathbf{Rep}(t_1, t_1'): \mathbf{Me}[\![exp_2]\!](q)(te)(e) \to \mathbf{Me}[\![exp_2]\!](q)(te')(e'),$$

which we can assure only if TypeOf $[\![exp_2]\!](q) \simeq w_1$. (Here we need a trivial lemma that for all w, w', $w \simeq w'$ implies $\mathbf{Mt}[\![w]\!](te) = \mathbf{Mt}[\![w']\!](te)$.) And this is exactly what the type-checking clause of the meaning of application allows us to assert. Q.E.D.

One way to view this result is that "the implementor is free to choose any representation he desires." For example, let us assume for the moment that the domain B includes the usual flat domains of integers and Booleans, *Int* and *Bool*, i.e., $B = Int + Bool + \cdots$. Then consider the following data types t_1 and t_2 used to implement Booleans:

1. $t_1 = \langle \lambda c \in Const.C = \text{true}$ **then** 1 **else** $c = \text{false}$ **then** 0 **else** $\perp$, $\lambda x \in d.x \rangle$ and
2. $t_2 = (\lambda c \in Const.C = \text{true}$ **then true else** $c = \text{false}$ **then false else** $\perp$, $\lambda x \in D.x | Bool \rangle$.

Note that the retract used in t_2 only produces value in *Bool*, i.e., the only elements of data type t_2 are the elements of the domain *Bool*, while all values in D are elements of the data type t_1.

Now from our definition of **Rep**, $\mathbf{Rep}(t_1, t_2)$ relates the following pairs of values (1, **true**), (0, **false**), ($\perp$, $\perp$), and ($\top$, $\top$). And from the theorem above, we know that if TypeOf $[\![exp]\!](q) =$ Boolean, then

$$\mathbf{Rep}(t_1, t_2)(\mathbf{Me}[\![exp]\!](q)(te[Boolean \leftarrow t_1)(e),$$
$$\mathbf{Me}[\![exp]\!](q)(te'[Boolean \leftarrow t_2])(e'))$$

if te, te' and e, e' are as required in the theorem. This means that:

1. even using t_1 as the meaning of Boolean, the only values produced by Boolean expressions are 0, 1, $\perp$, or $\top$, and
2. the denotation of the same Boolean constant is produced in either case, i.e., if the "abstract meaning" of an expression is Boolean$true, then its meaning will always be the meaning of Boolean$true.

Thus, when read in implementation-oriented terms, this says that the implementor may choose the meanings of the basic data types as he pleases. If we view the purpose of type-checking as guaranteeing representation independence of the sort we define above, then our semantics is not only "correctly typed," but "strongly typed."

4. Conclusions. We have described a new approach to the meaning of "data type" and applied it to the definition of a typed lambda-calculus extension. In conclusion, we would like to step back from the details of the previous section and suggest the advantages of this approach as illustrated by our lambda-calculus semantics.

First, we claim that our lambda calculus semantics, while admittedly abstract, treats data types in an intuitively appealing manner. As we described above, the choice of

$$T = [Const \to B] \times [D \to D] + D \to D$$

can be motivated in terms of an SECD implementation and our restriction of T to retracts again makes sense in terms of this machine-oriented view. (It is interesting to note that Scott also uses retracts to define data types of his "universal domain" $P\omega$ (Scott [18]).) And, as is common in statically typed languages, our semantics does not involve any checking of the "types" of values, but only the "syntactic types" associated with identifiers and expressions. Finally in this regard we note the weakness of the

assumptions used in proving the theorems of the previous section. Of particular interest is the fact that no assumptions were made about the domain B of basic values. Although we are using B to interpret constants, we do not require it to be "flat", (i.e., without structure). Indeed, we could even do without B altogether and make elements $D \to D$ serve double duty as both functions and constants without changing the interpretation of data types at all.

This definition also shows nicely the differences between typed language and "typeless" languages, like BCPL. One way to view a "typeless" language is to say that in fact there is only a single type. In BCPL, this view can be seen in the single interpretation of assignment; the operational meaning of $x := y$ is "move k bits from y to x." In the untyped lambda calculus, the same view suggests a semantics with only a single meaning for "take the value of the variable x." This is exactly what one finds in semantic treatments of untyped or "typeless" lambda calculus and its variants (cf., Stoy [20], Scott [18]).

Finally, two points relating to language design should be made. As languages have included more complex data type definitions, the rules for type-checking have also become far more complex (see, for example, the descriptions of type-checking in Euclid (Lampson et al. [8]) or Mesa (Geschke et al. [2])). Although the expressed purpose of such type-checking is to catch program errors, one can't help wondering whether these type-checking rules do not, in fact, introduce subtle semantic errors. Our representation theorem, however, suggests a means of judging the soundness of any proposed type-checking rules; the type-checking must be sufficient to guarantee the representation independence of the semantics in the fashion we described above.

Most interestingly, our view of data types allows a simple and straightforward treatment of constructs allowing types as parameters. To illustrate the lack of understanding of such constructs, we need only refer to the following comments of Wirth [22]:

> I would caution, however, against any further generalization [or array type specifications of procedure parameters]. Allowing the component type of an array to be a parameter too, for example, would destroy many advantages of the Pascal type concept at once.

Although PASCAL data types are certainly more complicated than typed lambda calculus data types, we are convinced that the same principles used in this paper apply. To this end, we (with A. Demers and G. Skinner) are designing an extension of PASCAL (called Russell) incorporating polymorphic procedures and data types. Both denotational and axiomatic semantics for this language are also being developed.

5. Acknowledgment. Robert Constable, David Gries and the referees provided many helpful comments on earlier drafts of this paper.

REFERENCES

[1] J. D. GANNON AND J. J. HORNING, *The impact of language design on the production of reliable software*, SIGPLAN Notices, 10 (1975), pp. 10–22.

[2] C. M. GESCHKE, J. H. MORRIS AND E. H. SATTERTHWAITE, *Early experiences with Mesa*, Comm. ACM, 20 (1977), pp. 540–552.

[3] J. A. GOUGEN, J. W. THATCHER AND E. G. WAGNER, *An initial algebra approach to the specification, correctness and implementation of abstract data types*, RC6487, IBM. T. J. Watson Research Center, Yorktown, New York, 1976.

[4] J. V. GUTTAG, *The specification and application to programming of abstract data types*, Ph.D. thesis, Department of Computer Science, University of Toronto, 1975.

[5] C. A. R. HOARE, *Notes on Data Structuring. Structured Programming*, Academic Press, London–New York, 1972.

[6] C. A. R. HOARE AND N. WIRTH, *Axiomatic definition of the programming language Pascal*, Acta Informat., 2 (1973), pp. 335–355.
[7] K. JENSEN AND N. WIRTH, *PASCAL User Manual and Report*, Springer-Verlag, New York 1975.
[8] B. LAMPSON, J. J. HORNING, R. L. LONDON AND G. L. POPEK, *Report on the proramming language Euclid*, SIGPLAN Notices, 12 (1977), pp. 1–79.
[9] P. J. LANDIN, *The mechanical evaluation of expressions*, Comput. J., 6 (1964), pp. 308–320.
[10] H. LEDGARD, *A model for type checking—with an application to Algol 60*, Comm. ACM, 15 (1972), pp. 956–966.
[11] B. LISKOV AND S. ZILLES, *Programming with abstract data types*, SIGPLAN Notices, 9 (1974), pp. 50–59.
[12] R. MILNE AND C. STRACHEY, *A Theory of Programming Language Semantics*, Halstead Press, 1976.
[13] J. H. MORRIS, *Lambda-calculus models of programming languages*, MAC-TR-57, Project MAC, MIT, 1968.
[14] ———, *Types are not sets*, ACM Symposium on Principles of Programming Languages (Boston, 1973), pp. 120–124.
[15] J. C. REYNOLDS, *GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept*, Comm. ACM, 13 (1970), pp. 308–318.
[16] ———, *Towards a theory of type structure*, Colloquim on Programming, Paris, 1974.
[17] C. SCHAFFERT, A. SNYDER AND R. ATKINSON, *The CLU reference manual*, Project MAC, MIT, 1975.
[18] D. SCOTT, *Data types as lattices*, this Journal, 5 (1976), pp. 522–580.
[19] A. SHAMIR AND W. W. WADGE, *Data Types as Objects*, Automata Languages and Programming, 4th Colloquium, Lecture Notes in Computer Science No. 52 (Turku, 1977), pp. 465–479.
[20] J. STOY, *The Scott–Strachey approach to the mathematical semantics of programming languages*, Project MAC, MIT, 1974.
[21] R. D. TENNENT, *The denotational semantics of programming languages*, Comm. ACM, 19 (1976), pp. 437–453.
[22] N. WIRTH, *An assessment of the programming language Pascal*, SIGPLAN Notices, 10 (1975), pp. 23–30.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0007 $01.00/0

A PATCHING ALGORITHM FOR THE NONSYMMETRIC TRAVELING-SALESMAN PROBLEM*

RICHARD M. KARP†

Abstract. We present an algorithm for the approximate solution of the nonsymmetric n-city traveling-salesman problem. An instance of this problem is specified by a $n \times n$ distance matrix $D = (d_{ij})$. The algorithm first solves the assignment problem for the matrix D, and then patches the cycles of the optimum assignment together to form a tour. The execution time of the algorithm is comparable to the time required to solve an $n \times n$ assignment problem.

If the distances d_{ij} are drawn independently from a uniform distribution then, with probability tending to 1, the ratio of the cost of the tour produced by the algorithm to the cost of an optimum tour is $< 1 + \varepsilon(n)$, where $\varepsilon(n)$ goes to zero as $n \to \infty$. Hence the method tends to give nearly optimal solutions when the number of cities is extremely large.

Key words. traveling-salesman problem, combinatorial optimization, approximation algorithms, probabilistic analysis of algorithms

1. Introduction. Let Σ_n denote the set of all permutations of $\{1, 2, \cdots, n\}$ and let Σ_n^* denote the set of all cyclic permutations of $\{1, 2, \cdots, n\}$. For any $n \times n$ matrix $D = (d_{ij})$ of nonnegative real numbers and any permutation $\pi \in \Sigma_n$, define $c(\pi, D) = \sum_{i=1}^{n} d_{i,\pi(i)}$.

The (*nonsymmetric*) *traveling-salesman problem* is stated as follows: given D, find a cyclic permutation $\pi^*(D)$ (or simply π^*, when D is understood) such that $c(\pi^*, D) = \min_{\pi \in \Sigma_n^*} c(\pi, D)$. This problem typically arises in machine scheduling applications, where d_{ij} represents the set-up cost for job j upon the completion of job i, and an optimum sequence of job execution is desired. Since the directed traveling-salesman problem is $\mathcal{NP}$-hard [6], it is not reasonable to expect to find a polynomial-time algorithm for its exact solution. Well-designed branch-and-bound methods are capable of efficiently solving problem instances of size up to about $n = 100$ [8].

By an *approximation algorithm* for the traveling-salesman problem we mean an algorithm $\mathcal{A}$ that, given any matrix D, produces a cyclic permutation $\hat{\pi}(D)$. The *relative error* associated with the execution of $\mathcal{A}$ on D is

$$\varepsilon(D) = \frac{c(\hat{\pi}(D), D) - c(\pi^*(D), D)}{c(\pi^*(D), D)}.$$

Sahni and Gonzales have shown that, given any $\varepsilon > 0$, it is $\mathcal{NP}$-hard to solve the traveling-salesman problem with relative error $< \varepsilon$. Thus, we cannot expect to find a polynomial-time approximation algorithm with uniformly bounded relative error.

In this paper we present a polynomial-time approximation algorithm which tends to give solutions with small relative error. The algorithm starts by solving the $n \times n$ assignment problem, which is stated as follows: given D, find a permutation $\bar{\pi}(D)$ (or simply $\bar{\pi}$) such that

$$c(\bar{\pi}, D) = \min_{\pi \in \Sigma_n} c(\pi, D).$$

There are algorithms which solve the assignment problem in time $O(n^3)$ [4], [9]. Our approximation algorithm produces a cyclic permutation $\hat{\pi}$ by patching together the

* Received by the editors January 23, 1978, and in final revised form September 8, 1978. This research supported by National Science Foundation under Grant MCS74-017680-A02.

† Computer Science Division, Department of Electrical Engineering and Computer Sciences and the Electronics Research Laboratory, University of California at Berkeley, Berkeley, California 94720.

cycles of the optimal assignment permutation $\bar{\pi}(D)$. The running time of the algorithm is $O(n^3)$. The algorithm also yields an upper bound on the relative error $\varepsilon(D)$. Our main theorem states that, if the d_{ij} are drawn independently from the uniform distribution on $[0, 1]$, then with probability tending to 1 as $n \to \infty$, this upper bound is very small.

A companion paper to the present one [7] gives similar results for the traveling-salesman problem in the plane.

It is interesting that a patching algorithm similar to ours has been proven to give strictly optimum solutions for an important special class of traveling-salesman problems [5].

2. The patching algorithm. We begin by stating the $m \times n$ *assignment problem.* Let m and n be positive integers with $m \leqq n$. Let $S_{m,n}$ denote the set of single-valued one-one functions from $\{1, 2, \cdots, m\}$ into $\{1, 2, \cdots, n\}$. In particular, when $m = n$, $S_{m,n} = \Sigma_n$, the set of permutations of $\{1, 2, \cdots, n\}$. Given an $m \times n$ matrix $A = (a_{ij})$ of real numbers, the assignment problem asks for a function $\bar{\pi} \in S_{m,n}$ such that

$$\sum_{i=1}^{m} a_{i,\bar{\pi}(i)} = \min_{\pi \in S_{m,n}} \sum a_{i,\pi(i)}.$$

There are algorithms to solve the $m \times n$ assignment problem in $O(m^2 n)$ steps [4], [9].

Given an $n \times n$ matrix D, the patching algorithm begins by finding an optimum assignment $\bar{\pi}$. If, fortuitously, $\bar{\pi}$ is a cyclic permutation, then the traveling-salesman problem is solved. Otherwise, $\bar{\pi}$ will have two or more cycles. The algorithm patches these cycles together into a single cycle, thereby obtaining a cyclic permutation.

We next describe how the patching is done. Let $\rho_{ij} \in \Sigma_n$ be the permutation that interchanges elements i and j, leaving all other elements fixed. The transformation

$$R_{ij}: \Sigma_n \to \Sigma_n,$$

defined by $R_{ij}(\pi) = \pi \circ \rho_{ij}$, is called the i, j *patching operation.* Also, define

$$\Delta_{ij}(\pi, D) = d_{i,\pi(j)} + d_{j,\pi(i)} - d_{i,\pi(i)} - d_{j,\pi(j)}.$$

The following lemma is immediate.

LEMMA 1. *For all i, j,*

$$c(\pi \circ \rho_{ij}, D) = c(\pi, D) + \Delta_{ij}(\pi, D).$$

Also, if i and j are in different cycles of π, then the elements in these two cycles lie in a single cycle of $\pi \circ \rho_{ij}$, and the other cycles of π remain unchanged.

Figure 1 indicates the effect of the i, j patching operation.

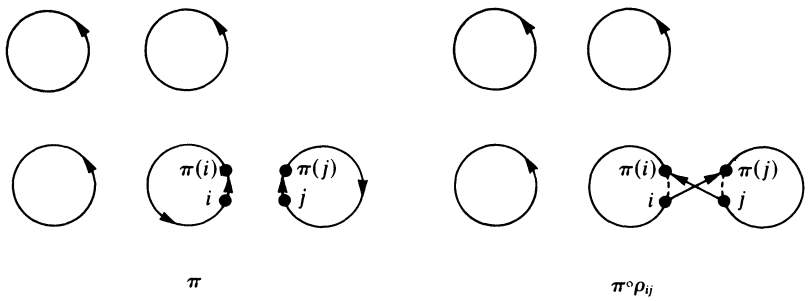


FIG. 1. *Effect of the i, j patching operation.*

We next describe a *patching process* which, given a permutation α with k cycles, attempts to transform α to a cyclic permutation by applying a sequence of $k-1$ patching operations. This sequence is selected as follows. First, some cycle $\bar{C}$ of maximum length in α is selected. Let the remaining cycles be $C_1, C_2, \cdots, C_{k-1}$. Ambiguously, we let the name of a cycle also stand for the set of elements of the cycle. If $|\bar{C}| < k-1$, then the algorithm reports failure and halts. (We shall see that this event has negligible probability.) Otherwise, a $(k-1)\times|\bar{C}|$ assignment problem is set up whose solution gives an optimum way to patch all the cycles $C_1, C_2, \cdots, C_{k-1}$ into $\bar{C}$ at distinct places. The matrix A defining this problem has $k-1$ rows and a column for each $j \in \bar{C}$. The i-j entry is

$$(*) \qquad a_{ij} = \min_{l \in C_i} \Delta_{lj}(\alpha, D).$$

Thus, a_{ij} is the least cost of a patching operation involving element $j \in \bar{C}$, and any element in C_i. Let the minimizing l in $(*)$ be denoted $l(i, j)$. Let the solution to this assignment problem be a 1-1 function $\theta: \{1, 2, \cdots, k-1\} \to \bar{C}$. Then, for $i = 1, 2, \cdots, k-1$, the patching process performs the patching operation $R_{l(i,\theta(i)),\theta(i)}$. These operations commute, and may thus be performed in any order.

Figure 2 indicates how a permutation α with four cycles might be converted into a tour. Let $\delta(\alpha, D, \bar{C}) = \sum_{i=1}^{k-1} a_{i,\theta(i)}$. Then the cyclic permutation obtained by applying the

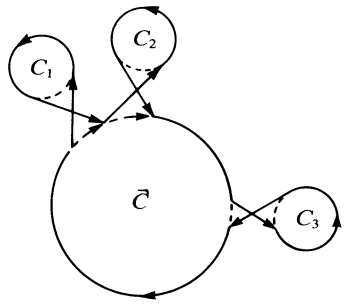


FIG. 2. *Application of the patching process.*

patching process to α has cost $C(\alpha, D) + \delta(\alpha, D, \bar{C})$. The time required for the patching process is $O((k-1)^2|\bar{C}|) = O(n^3)$.

The over-all patching algorithm is now easily stated:

(i) find an optimal assignment $\bar{\pi}$ for the matrix D;

(ii) if $\bar{\pi}$ has k cycles and no cycle is of length $\geqq k-1$, then halt and report failure;

(iii) otherwise, apply the patching process to obtain a cyclic permutation $\hat{\pi}(D)$ of cost $c(\bar{\pi}, D) + \delta(\bar{\pi}, D, \bar{C})$;

(iv) print out the permutation $\hat{\pi}$ and the error bound

$$\varepsilon(D) \leqq \frac{\delta(\bar{\pi}, D, \bar{C})}{c(\bar{\pi}, D)}.$$

The execution time of the patching algorithm is $O(n^3)$.

Section 3 analyzes the distribution of $\varepsilon(D)$. Section 4 gives a heuristic error analysis for a variant of the patching algorithm.

3. Relative error of the patching algorithm. Recall that, for any matrix D, $\hat{\pi}(D)$ denotes the cyclic permutation produced by the patching algorithm, and $\pi^*(D)$ denotes

an optimal solution to the traveling-salesman problem for D. Thus the relative error of the patching algorithm is given by

$$\varepsilon(D)=\frac{c(\hat{\pi}, D)-c(\pi^*, D)}{c(\pi^*, D)}.$$

Let $U_{n\times n}$ be the uniform distribution over the set of $n\times n$ matrices whose elements lie in $[0, 1]$.

THEOREM 1. *Let D be drawn from $U_{n\times n}$. Then with probability tending to 1 as $n\to\infty$,*

$$\varepsilon(D)<9(\ln n)^{3/2}n^{-.24}. \tag{1}$$

The present section is devoted to the proof of Theorem 1. We begin with some preliminary remarks and propositions needed for the proof.

Drawing a matrix D from $U_{n\times n}$ is equivalent to drawing each element independently from the uniform distribution on $[0, 1]$. With probability 1, a D drawn from $U_{n\times n}$ has the property that no two sets of its elements have the same sum. We assume that all distance matrices considered have this property. Thus, in particular, we assume that every matrix D presented to the algorithm has a unique optimum assignment.

The first proposition gives upper bounds on the tails of the binomial distribution. The first of these bounds is given in Chvátal [3], where it is attributed to S. N. Bernstein. The bound has also been derived by D. Angluin [1], using a technique due to Chernoff [2]. The second bound is a direct consequence of the first. The third bound has been derived from the first by Angluin [1].

PROPOSITION 1. *For $0\leqq p\leqq 1$, N a positive integer and a a nonnegative integer less than or equal to N*

(a) *if $a\leqq Np$ then*

$$\sum_{k=0}^{a}\binom{N}{k}p^k(1-p)^{N-k}\leqq\left(\frac{Np}{a}\right)^a\left(\frac{N(1-p)}{N-a}\right)^{N-a};$$

(b) *if $a\geqq Np$ then*

$$\sum_{k=a}^{N}\binom{N}{k}p^k(1-p)^{N-k}\leqq\left(\frac{Np}{a}\right)^a\left(\frac{N(1-p)}{N-a}\right)^{N-a};$$

(c) *for all $\beta\in[0, 1]$*

$$\sum_{k=0}^{[(1-\beta)Np]}\binom{N}{k}p^k(1-p)^{N-k}<\exp\left(-\frac{\beta^2Np}{2}\right).$$

The second proposition concerns random permutations. Ambiguously, let Σ_n denote both the set of all permutations of $\{1, 2, \cdots, n\}$, and the uniform distribution over this set. The symbol $m(\alpha)$ denotes the maximum length of any cycle of the permutation α.

PROPOSITION 2. *Let α be drawn from Σ_n. Then, with probability tending to 1 as $n\to\infty$,*

(a) *α has at most $3\ln n$ cycles;*

(b) $n/(3\ln n)\leqq m(\alpha)\leqq n-n^{2/3}$

and

(c) *α has exactly one cycle of length $m(\alpha)$.*

Proof. Let α be a permutation in Σ_n. By the *cycle structure* of α is meant that partition of n in which an integer l appears as many times as α has cycles of length l. The uniform distribution over Σ_n induces a probability distribution P_n over all the possible cycle structures.

Consider the following method of selecting a random partition of n.

PROCEDURE PARTITION(n)
begin;
$i \leftarrow n$;
while $i > 0$ **do begin**;
1. select a random integer k from the uniform distribution over $\{1, 2, \cdots, i\}$;
$i \leftarrow i - k$
end
end

The multiset of integers k selected during the process forms the desired partition.

We claim that executing PARTITION(n) is equivalent to sampling from the distribution P_n. This follows from two observations. First, for each k between 1 and n, the number of permutations in Σ_n such that some fixed element x lies in a cycle of length k is $(n-1)!$. Therefore, the length of the cycle containing x is uniformly distributed over $\{1, 2, \cdots, n\}$. Secondly, given that x lies in a cycle of length k, all permutations of the remaining $n-k$ elements are equally likely.

We prove (a) by considering the execution of PARTITION(n). Call an execution of step 1 a *success* if $\lceil \log_2 i \rceil > \lceil \log_2 (i-k) \rceil$. At each step the probability of success is $\geqq \frac{1}{2}$. Also, the process terminates no later than the $\lceil \log_2 n \rceil$-th success. Hence, the probability that α has more than $3 \ln n$ cycles is less than or equal to the probability that $3 \ln n$ flips of a fair coin will result in fewer than $\lceil \log_2 n \rceil$ heads; and, by Proposition 1, this probability is $o(1)$.

To prove (b), note that

$$Pr\left\{m(\alpha) < \frac{n}{3 \ln n}\right\} < Pr\{\alpha \text{ has at least } 3 \ln n \text{ cycles}\} = o(1)$$

and

$$\begin{aligned} Pr\{m(\alpha) > n - n^{2/3}\} &\leqq Pr\{\text{element 1 lies in a cycle of length } < n^{2/3} \text{ or } > n - n^{2/3}\} \\ &= O(n^{-1/3}). \end{aligned}$$

To prove (c), note that

$$\begin{aligned} &Pr\{\alpha \text{ contains two cycles of length } k\} \\ &\leqq \frac{n}{2k}(Pr\{\alpha \text{ contains two cycles of length } k, \text{ and element } x \text{ is in one of them}\}) \\ &\leqq \frac{n}{2k} \cdot \frac{1}{n} \cdot Pr\{\beta \in \Sigma_{n-k} \text{ has a cycle of length } k\} \\ &= \frac{n}{2k} \cdot \frac{1}{n} \cdot \frac{1}{k} = \frac{1}{2k^2}. \end{aligned}$$

Hence,

$$Pr\left\{\text{for some } k \geqq \frac{n}{3 \ln n},\ \alpha \text{ contains two cycles of length } k\right\}$$
$$\leqq \sum_{k=n/\ln n}^{n/2} \frac{1}{2k^2} = O\left(\frac{\ln n}{n}\right).$$

The third proposition concerns properties of permutations, given an upper bound on the lengths of their cycles.

PROPOSITION 3. *Let m and n be integers. Let Σ_n^m denote $\{\alpha \mid \alpha \in \Sigma_n \text{ and } m(\alpha) \leqq m\}$. Let A_k^n denote the expected number of elements occurring in cycles of length k in a permutation drawn at random from Σ_n^m. Then*

(a) $\sum_{k=1}^{m} A_k^n = n$ *and*

(b) $A_1^n \leqq A_2^n \leqq \cdots \leqq A_m^n$.

Proof. Part (a) is immediate, since every element is in exactly one cycle. To prove (b) let $F_n = |\Sigma_n^m|$. Then

$$\text{for all } l, \quad F_l \leqq lF_{l-1}. \tag{2}$$

This follows from the observation that the lth element can be added into each member of Σ_{l-1}^m in at most l ways to produce a permutation in Σ_l^m, and all permutations in Σ_l^m are produced by such insertions. Also

$$A_k^n = \frac{n(n-1)\cdots(n-k+1)F_{n-k}}{F_n}, \tag{3}$$

since A_k^n is n times the probability that element x lies in a cycle of length k. Hence

$$\frac{A_k^n}{A_{k-1}^n} = \frac{(n-k+1)F_{n-k}}{F_{n-k+1}} \geqq 1. \qquad \square$$

The fourth proposition concerns matrices drawn from $U_{n\times n}$.

PROPOSITION 4. *Let D be drawn from $U_{n\times n}$. Let $\bar{\pi}$ be the optimal assignment for D. Then, with probability tending to 1 as $n \to \infty$, $\frac{1}{3} < c(\bar{\pi}, D) < 3$.*

Proof. To prove the lower bound on $c(\bar{\pi}, D)$, note that

$$Pr\{c(\bar{\pi}, D) > \tfrac{1}{3}\} \geqq Pr\left\{\sum_i \min_j d_{ij} > \tfrac{1}{3}\right\} \geqq Pr\left\{\left|\left\{i \,\middle|\, \min_j d_{ij} > \frac{2}{3n}\right\}\right| \geqq \frac{n}{2}\right\}.$$

But, for any fixed i,

$$Pr\left\{\min_j d_{ij} > \frac{2}{3n}\right\} = \left(1 - \frac{2}{3n}\right)^n \xrightarrow[n\to\infty]{} e^{-2/3}.$$

Applying Proposition 1a (with $N = n$, $\frac{1}{2} < p < e^{-2/3}$, $(1-\beta) = 1/(2p)$), it follows that $Pr\{|\{i \mid \min_j d_{ij} > 2/(3n)\}| \geqq n/2\}$ tends to 1.

The upper bound on $c(\bar{\pi}, D)$ is due to David Walkup [10]. $\square$

Now we embark on the proof of Theorem 1. Let $D = (d_{ij})$ be drawn from $U_{n\times n}$. Call D *exceptional* if any of the following are violated:

(i) $\bar{\pi}(D)$ has at most $3 \ln n$ cycles;

(ii) $n/(3 \ln n) \leqq m(\bar{\pi}(D)) \leqq n - n^{2/3}$;

(iii) $\bar{\pi}(D)$ has a unique longest cycle;

(iv) $\frac{1}{3} < c(\bar{\pi}, D) < 3$.

By the above propositions, the probability that D is exceptional tends to 0 as $n \to \infty$. For

any $\sigma \in \Sigma_n$, define the matrix D^σ by $(D^\sigma)_{ij} = d_{i,\sigma(j)}$. Thus D^σ is obtained by permuting the columns of D. Let $[D]$ denote the set $\{D^\sigma | \sigma \in \Sigma_n\}$. The following lemma is the basis of our proof.

LEMMA 2. *For any permutation α, $c(\alpha, D) = c(\sigma^{-1}\alpha, D^\sigma)$. Hence, $\bar{\pi}$ is an optimal assignment for D if and only if $\sigma^{-1}\bar{\pi}$ is an optimal assignment for D^σ. Also, $\Delta_{ij}(\alpha, D) = \Delta_{ij}(\sigma^{-1}\alpha, D^\sigma)$.*

Given any set $T \subseteq \{1, 2, \cdots, n\}$, let Σ_n^T denote the set of all permutations α in Σ_n such that

(a) T is the set of elements in a cycle of α and

(b) the cycle containing these elements is a longest cycle of α;

i.e., $m(\alpha) = |T|$.

Let $\mathcal{S} = \{T \subseteq \{1, 2, \cdots, n\} | n/(3 \ln n) \leqq |T| \leqq n - n^{2/3}\}$. For any $T \in \mathcal{S}$ let $[D, T]$ denote the set of matrices in $[D]$ whose optimum assignment is in Σ_n^T. Note that, unless D^σ is exceptional, it lies in exactly one set $[D, T]$.

In the next four lemmas, let S be a fixed set in $\mathcal{S}$. For any permutation $\alpha \in \Sigma_n^S$, we construct a *patching matrix* $\Delta^\alpha(D, S)$ of dimension $(n - |S|) \times |S|$. The rows of this matrix correspond to the elements of $\{1, 2, \cdots, n\}$ not in S, and the columns, to the elements of S. The i, j entry gives the patching cost $\Delta_{ij}(\alpha, D)$, which, by Lemma 2, is equal to $\Delta_{ij}(\sigma^{-1}\alpha, D^\sigma)$.

A *bad element* of $\Delta^\alpha(D, S)$ is one which is $> n^{.26} m^{-1/2}$. An element which is not bad is a *good element.* A *bad row* of $\Delta^\alpha(D, S)$ is one that contains fewer than $3 \ln n$ good elements. The matrix $\Delta^\alpha(D, S)$ is a *bad matrix* if it contains more than $\sqrt{n}$ bad rows; otherwise $\Delta^\alpha(D, S)$ is a *good matrix.*

LEMMA 3. *If D is drawn at random from $U_{n\times n}$ and α is drawn at random from Σ_n^S, then $Pr\{\Delta^\alpha(D, S)$ is bad$\} = o(4^{-n})$.*

Proof. Define a matrix $\Omega^\alpha(D, S)$ (or, briefly, Ω^α) with the same rows and columns as $\Delta^\alpha(D, S)$, such that $(\Omega^\alpha)_{ij} = d_{i,\alpha(j)} + d_{j,\alpha(i)}$. Then Ω^α is element-by-element greater than or equal to $\Delta^\alpha(D, S)$, and it remains only to prove that $Pr\{\Omega^\alpha$ is bad$\} = o(4^{-n})$. The elements of Ω^α are independent, and each is the sum of two independent samples from the uniform distribution on $[0, 1]$. Thus, independently for each pair i, j, $Pr\{\Omega_{ij}^\alpha$ is good$\} = n^{.52}/(2m)$. Thus, applying Proposition 1c with $N = m$, $p = n^{.52}/(2m)$ and $\beta = 1 - 6 \ln n n^{-.52}$, we obtain:

$$Pr\{\text{row } i \text{ has} \leqq 3 \ln n \text{ good elements}\} < \exp\left(-(1 - 6 \ln n n^{-.52})^2 \frac{n^{.52}}{2} \cdot \frac{1}{2}\right)$$

$$= O(\exp(-n^{.51})).$$

Thus each row has probability $O(\exp(-n^{.51}))$ of being bad. Then, the probability that there are more than $\sqrt{n}$ bad rows is bounded above by substituting $N = n - m$, $p = O(\exp(-n^{.51}))$, $a = \sqrt{n}$ in Proposition 1b. The resulting upper bound is $o(4^{-n})$. □

LEMMA 4. *Let D be drawn from $U_{n\times n}$, let D^σ be drawn at random from $[D, S]$, and let $\bar{\pi} = \bar{\pi}(D^\sigma)$. Then $Pr\{D^\sigma$ is not exceptional and $\Delta^{\bar{\pi}}(D^\sigma, S)$ is bad$\} = o(\frac{3}{4})^n$.*

Proof.

$$Pr\{D^\sigma \text{ is not exceptional and } \Delta^{\bar{\pi}}(D^\sigma, S) \text{ is bad}\}$$

$$\leqq Pr\{c(\bar{\pi}, D^\sigma) < 3 \text{ and } \Delta^{\bar{\pi}}(D^\sigma, S) \text{ is bad}\}$$

$$\leqq Pr\{\exists \alpha \in \Sigma_n^S | c(\alpha, D^\sigma) < 3 \text{ and } \Delta^\alpha(D^\sigma, S) \text{ is bad}\}$$

$$\leqq E|\{\alpha \in \Sigma_n^S | c(\alpha, D^\sigma) < 3 \text{ and } \Delta^\alpha(D^\sigma, S) \text{ is bad}\}|$$

$$(*) \qquad \leqq n! Pr\{c(\alpha, D^\sigma) < 3 \text{ and } \Delta^\alpha(D^\sigma, S) \text{ is bad}\},$$

where α is a random element of Σ_n^S. But (*) is equal to

$$n!Pr\{\Delta^\beta(D, \sigma^{-1}S) \text{ is bad and } c(\beta, D) < 3\} \leqq n!Pr\{\Omega^\beta(D, \sigma^{-1}S) \text{ is bad and } c(\beta, D) < 3\}$$

where $\beta = \sigma^{-1}\alpha$ is a random permutation in $\Sigma_n^{\sigma^{-1}S}$. The two events "$\Omega^\beta(D, \sigma^{-1}S)$ is bad" and "$C(\beta, D) < 3$" are independent, since the first depends only on matrix entries d_{ij} such that $j \neq \beta(i)$, and the second depends only on $\{d_{i,\beta(i)}\}$. By Lemma 3 the first event has probability $o(4^{-n})$. The probability of the second event is

$$\int\cdots\int_{\substack{x_1+\cdots+x_n \leqq 3 \\ x_1,\cdots,x_n \geqq 0 \\ x_1,\cdots,x_n \leqq 1}} dx_1\, dx_2 \cdots dx_n \leqq \frac{3^n}{n!}.$$

Thus $(*) \leqq (n!3^n/n!)o(4^{-n}) = o(\frac{3}{4})^n$. □

LEMMA 5. *If $\Delta^\alpha(D, S)$ is a good matrix and D^σ is drawn at random from $\{D^\theta \in [D] \mid \theta^{-1}\alpha \in \Sigma_n^S\}$ then, $Pr\{\theta^{-1}\alpha$ has a cycle, all of whose elements correspond to bad rows of $\Delta^\alpha\} = O(\ln n n^{-1/6})$.*

Proof. The permutation $\theta^{-1}\alpha$ is a random element of Σ_n^S. Thus, restricting $\theta^{-1}\alpha$ to the domain $\{1, 2, \cdots, n\} - S$ gives a random permutation ϕ from Σ_{n-m}^m, where $m = |S|$. If the number of bad rows in $\Delta^\alpha(D, S)$ is t, then the expected number of cycles of ϕ with all rows bad is

$$\sum_{k=1}^{\min(m,t)} \frac{1}{k} A_k^{n-m} \frac{\binom{t}{k}}{\binom{n-m}{k}}$$

where $(1/k)A_k^{n-m}$ gives the expected number of cycles of length k in a random permutation from Σ_{n-m}^m, and the ratio $\binom{t}{k}\Big/\binom{n-m}{k}$ is the probability that all the rows of a cycle of length k are bad. Using the facts that

(a) $\binom{t}{k}\Big/\binom{n-m}{k}$ is a decreasing function of k;

(b) A_k^{n-m} is an increasing function of k (cf. Proposition 3); and

(c) $\sum_{k=1}^m A_k^{n-m} = n - m$,

we conclude that

$$\sum_{k=1}^{\min(m,t)} \frac{1}{k} A_k^{n-m} \frac{\binom{t}{k}}{\binom{n-m}{k}} \leqq \sum_{k=1}^{\min(m,t)} \frac{1}{k} \frac{n}{m} \frac{\binom{t}{k}}{\binom{n-m}{k}}$$

$$\leqq \frac{n}{m} \sum_{k=1}^{\infty} \left(\frac{1}{k}\right)\left(\frac{t}{n-m}\right)^k$$

$$= \frac{n}{m}\left(-\ln\left(1 - \frac{t}{n-m}\right)\right).$$

Using the inequalities

$$\frac{n}{\ln n} \leqq m \leqq n - n^{2/3} \quad \text{and} \quad t \leqq \sqrt{n}$$

$$Pr\{\phi \text{ has a cycle with all rows bad}\}$$

$$\leqq \ln n\left(-\ln\left(1-\frac{\sqrt{n}}{n^{2/3}}\right)\right) = O(\ln n \cdot n^{-1/6}). \qquad \square$$

LEMMA 6. *If $\theta^{-1}\alpha \in \Sigma_n^S$ has at most $3 \ln n + 1$ cycles, and has no cycle whose elements all correspond to bad rows of $\Delta^{\alpha}(D, S)$, then*

$$\delta(\theta^{-1}\alpha, D^{\theta}, S) \leqq 3(\ln n)^{3/2} n^{-.24}.$$

Proof. Under the stated assumptions it is possible to carry out the patching process so that each patch has cost $\leqq n^{.26}/\sqrt{m}$. Hence $\delta(\theta^{-1}\alpha, D^{\theta}) \leqq 3 \ln n(n^{.26}/\sqrt{m})$. Using the inequality $m \geqq n/(\log n)$, the result follows. $\square$

LEMMA 7. *Let D be drawn from $U_{n\times n}$. Let $\bar{\pi}$ be the optimal assignment for D. Let $\delta(\bar{\pi}, D)$ denote the cost of applying the patching algorithm to D. Then*

$$Pr\{\delta(\bar{\pi}, D) > 3(\ln n)^{3/2} n^{-.24}\} = o(1).$$

Proof. All elements of $[D]$ are equally likely to be drawn. Hence, the desired probability is equal to

$$\frac{1}{n!} E(|\{D^{\sigma} \in [D] | \delta(\sigma^{-1}\bar{\pi}, D^{\sigma}) > 3(\ln n)^{3/2} n^{-.24}\}|).$$

In order that $\delta(\sigma^{-1}\bar{\pi}, D^{\sigma})$ be greater than this bound one of three events must occur:

(a) D^{σ} is exceptional;

(b) D^{σ} is not exceptional and, for some $T \in \mathcal{S}$, $D^{\sigma} \in [D, T]$ and $\Delta^{\bar{\pi}(D^{\sigma})}(D, T)$ is bad;

(c) D^{σ} is not exceptional and, for some $T \in \mathcal{S}$, $D^{\sigma} \in [D, T]$, $\Delta^{\bar{\pi}(D^{\sigma})}(D, T)$ is good, and $\delta(\bar{\pi}(D^{\sigma}), D^{\sigma}, T) > 3(\ln n)^{3/2} n^{-.24}$.

The expected number of matrices for which the first event is true is $o(n!)$. The expected number of matrices for which the second event is true is $\sum_{T\in\mathcal{S}} |[D, T]| o(\frac{3}{4})^n$. Here, $o(\frac{3}{4})^n$ is an upper bound on the probability that $\Delta^{\pi}(D, T)$ is bad (cf. Lemma 4). By Lemmas 5 and 6 the expected number of matrices for which the third event is true is $\sum_{T\in\mathcal{S}} |[D, T]| O(\ln n \cdot n^{-1/6})$. Finally, recalling that a matrix occurs in only one set $[D, T]$ unless it is exceptional, the result follows. $\square$

Proof of Theorem 1. The inequality (1) can fail only if $c(\bar{\pi}(D), D) < \frac{1}{3}$ or $\delta(\bar{\pi}(D), D) > 3(\ln n)^{3/2} n^{-.24}$. By Proposition 4 and Lemma 7, the probability of each of these events tends to zero. $\square$

Results analogous to Theorem 1 hold whenever the d_{ij} are drawn from a distribution over $[0, \infty]$ having a bounded density function continuous at 0^+.

4. Heuristic analysis of a modified patching algorithm. Theorem 1 shows that, when the number of cities is sufficiently large, the patching algorithm tends to give nearly optimal solutions to random nonsymmetric traveling-salesman problems. The result is not entirely satisfying, however, because the upper bound on $\varepsilon(D)$ given in the theorem tends to zero very slowly, and is acceptably small only when n is astronomically large. Also, the probability that the upper bound will be exceeded tends to zero very slowly.

In this section we present a modified patching algorithm and offer a heuristic argument indicating that its expected patching cost is less than $2n^{-1/2}$.

In the modified patching algorithm, all entries d_{ii} are set to $+\infty$. This ensures that the optimal assignment permutation will have no fixed points; i.e., no cycles of length 1. All permutations without fixed points remain equally likely to occur as the optimum assignment.

Having constructed the optimal assignment $\bar{\pi}$, the algorithm converts it to a tour as follows:

MODIFIED PATCHING PROCESS

$\sigma \leftarrow \bar{\pi}$

while σ has more than one cycle **do**;

begin;

let C be a shortest cycle of σ;

let R_{ij} be a minimum-cost patching operation such that $i \in C$, $j \notin C$,

and neither i nor j has been involved in a previous patching operation;

$\sigma \leftarrow R_{ij}(\sigma)$

end.

Thus, we no longer restrict attention to patching operations that join the short cycles of $\bar{\pi}$ directly into the longest cycle of $\bar{\pi}$. Figure 3 indicates how the modified patching process converts a permutation to a tour.

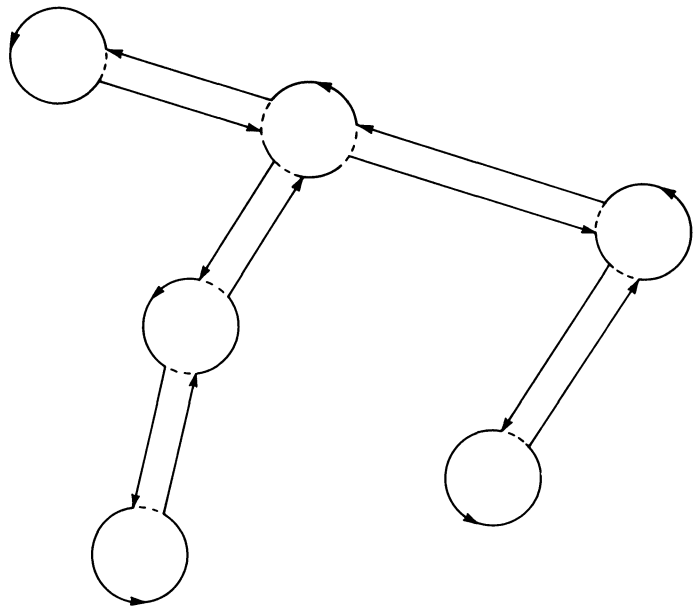


FIG. 3. *Application of the modified patching process.*

Next we study the behavior of the modified patching algorithm on a special class of matrices, and argue heuristically that the algorithm should have similar behavior when applied to matrices from $U_{n\times n}$.

We denote the special class of matrices by $\mathcal{M}$. A matrix D is in the class $\mathcal{M}$ if the row minima in D lie in distinct columns, and hence determine the optimal assignment for D. Formally, $D \in \mathcal{M}$ if there is a permutation $\bar{\pi}$ such that, for all i and j, $d_{i,\bar{\pi}(i)} \leqq d_{ij}$.

The following theorem states that, when a matrix is drawn at random from $\mathcal{M}$, the patching costs Δ_{ij} tend to be at least as small as they would be if the Δ_{ij} were independent, and each were distributed as the sum of two independent samples from the uniform distribution over [0, 1].

To frame the theorem precisely, we introduce the concept of stochastic dominance. Let $X = (x_1, x_2, \cdots, x_m)$ and $Y = (y_1, y_2, \cdots, y_n)$ be two random variables over $\mathcal{R}^m$, where $\mathcal{R}$ denotes the reals. We say $X < Y$ (X is stochastically smaller than Y) if, for every $A = (a_1, a_2, \cdots, a_m) \in \mathcal{R}^n$, $Pr\{X < A\} \geqq Pr\{Y < A\}$. Here $X < A$ if, for all i, $x_i < a_i$.

Let $\bar{\pi}$ be a fixed permutation of $\{1, 2, \cdots, n\}$ without fixed points. Let $X = \{x_{ij} \mid 1 \leqq i < j \leqq n, j \neq \bar{\pi}(i) \text{ and } i \neq \bar{\pi}(j)\}$ be the random variable over $\mathcal{R}^{\binom{n}{2}-n}$ determined by the following experiment: draw a matrix D from the set of matrices in $\mathcal{M}$ having $\bar{\pi}$ as their optimal assignment; then let $x_{ij} = \Delta_{ij}(\bar{\pi}, D)$. Let $Y = \{y_{ij} \mid 1 \leqq i < j < n, j \neq \bar{\pi}(i) \text{ and } i \neq \bar{\pi}(j)\}$ be the random variable over $\mathcal{R}^{\binom{n}{2}-n}$ determined as follows: the y_{ij} are independent, and each is the sum of two independent samples from the uniform distribution over [0, 1].

THEOREM 2. $X < Y$.

Proof. We condition on arbitrary fixed values for the entries $d_{i,\bar{\pi}(i)}$. Then the d_{ij}, $j \neq \bar{\pi}(i)$, are independent, with d_{ij} distributed according to a uniform distribution over $[d_{i,\bar{\pi}(i)}, 1]$. Hence the differences $d_{ij} - d_{i,\bar{\pi}(i)}$ are independent, and each such difference is drawn from a uniform distribution over $[0, 1 - d_{i,\bar{\pi}(i)}]$. Hence the Δ_{ij} are independent of one another, and each particular patching cost $\Delta_{ij} = (d_{ij} - d_{i,\bar{\pi}(i)}) + (d_{ji} - d_{j,\bar{\pi}(j)})$ is the sum of two independent random variables; one drawn from the uniform distribution over $[0, 1 - d_{i,\bar{\pi}(i)}]$, and the other from the uniform distribution over $[0, 1 - d_{j,\bar{\pi}(j)}]$. The result now follows, since a random variable uniformly distributed over $[0, 1 - d_{i,\bar{\pi}(i)}]$ stochastically dominates a random variable uniformly distributed over [0, 1]. □

We conjecture that an analogous property holds when matrices drawn from $U_{n\times n}$, rather than $\mathcal{M}$, are considered. More precisely, let $\bar{\pi}$ be a fixed permutation without fixed points. Let $Z = \{z_{ij} \mid 1 \leqq i < j \leqq n, j \neq \bar{\pi}(i), i \neq \bar{\pi}(j)\}$ be a random variable over $\mathcal{R}^{\binom{n}{2}-n}$ determined by the following experiment: draw a matrix D from the set of matrices in $U_{n\times n}$ having $\bar{\pi}$ as their optimal assignment; then let $z_{ij} = \Delta_{ij}(\bar{\pi}, D)$.

CONJECTURE. $Z < Y$.

As a heuristic argument in support of the conjecture, we define a mapping $\tau: U_{n\times n} \to \mathcal{M}$ as follows. Let $D \in U_{n\times n}$ have $\bar{\pi}$ as its optimal assignment. Then

$$(\tau(D))_{i,\bar{\pi}(i)} = \min_j d_{ij}$$

for $k \neq \pi(i)$

$$(\tau(D))_{ik} = \begin{cases} d_{ik} & \text{if } d_{ik} \neq \min_j d_{ij}, \\ d_{i,\bar{\pi}(i)} & \text{if } d_{ik} = \min_j d_{ij}. \end{cases}$$

Thus, $\tau(D)$ is obtained by interchanging the minimum element in each row with the element of that row which occurs in the optimum assignment. The following facts are immediate.

(a) The matrices D and $\tau(D)$ have the same optimal assignment $\bar{\pi}$;

(b) For all i and j, $\Delta_{ij}(\bar{\pi}, D) \leqq \Delta_{ij}(\bar{\pi}, \tau(D))$.

Theorem 2, coupled with condition (b), which asserts that the patching costs associated with $\tau(D)$ are at least as great as those associated with D, tends to support the conjecture. To prove the conjecture, it would be necessary to show that, when D is

uniformly distributed over $U_{n\times n}$, its image $\tau(D)$ is approximately uniformly distributed over $\mathcal{M}$.

In view of Theorem 2 and Conjecture 1, it is of interest to elucidate the behavior of the modified patching algorithm when $\bar{\pi}$ is a random permutation without fixed points, the Δ_{ij} are independent, and each is the sum of two independent random variables uniformly distributed over [0, 1]. We do so briefly, omitting details. Let the random variable γ_n denote the cost of the modified patching process under these assumptions.

THEOREM 3. $\lim_{n\to\infty} n^{1/2}E(\gamma_n) \leqq 2$.

The underlying ideas of the proof are as follows:

(a) the expected number of cycles of length k in a random permutation without fixed points is $(1/k)(1+O(n^{-1}))$;

(b) given a cycle C of length k, $E(\min\{\Delta_{ij} \mid i \in C, j \notin C\}) \leqq \sqrt{(3.1416)/(k(n-k))}$. These facts suggest that the expected patching cost is bounded above by $\sum_{k=2}^{n}(1/k)(1+O(n^{-1}))\sqrt{(3.1416)/(2k(n-k))} \sim 2n^{-1/2}$. The proof becomes more complicated than this sketch because of the possibility that the short cycles of $\bar{\pi}$ may become joined as the patching process takes place. We omit further details.

COROLLARY 1. *If the previous Conjecture is true, then* $\lim_{n\to\infty} n^{1/2}E[c(\hat{\pi}, D) - c(\pi^*, D)] \leqq 2$ *where* D *is drawn from* $U_{n\times n}$.

A Monte Carlo simulation was conducted to further determine the behavior of the random variable γ_n. The simulation was equivalent to determining γ_n at each of 100 random choices of $\bar{\pi}$ and $\{\Delta_{ij}\}$, for each of the values $n = 100$, $n = 1{,}000$, and $n = 10{,}000$. The simulation avoided explicit generation of random permutations and random patching costs; instead, it conducted a probabilistically equivalent experiment using theoretical properties of the cycle structure of a random permutation, and of the distribution of the minimum of a given number of independent patching costs. The results were as follows.

TABLE 1
Simulated behavior of the random variable γ_n

n	100	1,000	10,000
sample size	100	100	100
sample mean	.18	.067	.018
sample mean $\times\sqrt{n}$	1.8	2.1	1.8
sample median $\times\sqrt{n}$	1.6	2.0	1.7
sample maximum $\times\sqrt{n}$	5.4	4.9	4.0

Acknowledgment. I would like to express my appreciation to an anonymous referee who corrected the original proof of Theorem 1.

REFERENCES

[1] D. ANGLUIN, personal communication, 1978.

[2] H. CHERNOFF, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Annals Math. Statist., 23 (1952), pp. 493–507.

[3] V. CHVÁTAL, *Determining the stability number of a graph*, Report STAN-CS-76-583, Stanford University Computer Science Department, 1976, Stanford, CA.

[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] P. GILMORE AND R. GOMORY, *Sequencing a one state variable machine: A solvable case of the travelling-salesman problem*, Operations Res., 12 (1964), pp. 655–679.

[6] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. Thatcher, eds., Plenum Press, New York, 1972.

[7] ———, *Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane*, Math. of Operations Res. (1977).

[8] G. L. THOMPSON, *Algorithmic and computational methods for solving symmetric and asymmetric traveling-salesman problems*, Working Paper presented at the Workshop on Integer Programming (Bonn, 1975).

[9] N. TOMIZAWA, *On some techniques useful for solution of transportation problems*, Networks, 1 (1972), pp. 173–194.

[10] D. WALKUP, *On the expected value of a random assignment problem*, this Journal, 8 (1979), pp. 440–444.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0904-0008 $01.00/0

CLASSES OF PEBBLE GAMES AND COMPLETE PROBLEMS*

TAKUMI KASAI†, AKEO ADACHI‡ AND SHIGEKI IWATA§

Abstract. A "pebble game" is introduced and some restricted pebble games are considered. It is shown that in each of these games the problem to determine whether there is a winning strategy (two-person game) is harder than the solvability problem (one-person game). We also show that each of these problems is complete in well-known complexity classes. Several familiar games are presented whose winning strategy problems are complete in exponential time.

Key words. Turning machine, two-person game, winning strategy, pebble game, log-space, polynomial time, NP, polynomial space, exponential time

1. Introduction. A number of complete problems in various complexity classes are reported. Jones and Laaser [9] showed some familiar problems which are complete in deterministic polynomial time with respect to log space reducibility. A great number of familiar problems have been reported which are complete in NP (nondeterministic polynomial time) [1], [3], [11]. Even and Tarjan [6] considered generalized Hex and showed that the problem to determine who wins the game if each player plays perfectly is complete in polynomial space. Schaefer [13] derived some two-person games from NP complete problems which are complete in polynomial space. Chandra and Stockmeyer [2] proved some two-person games to be complete in exponential time.

We introduce a "pebble game" which involves moving pebbles according to certain rules. The goal of the game is to put a pebble on a particular place. The pebble game introduced here is somewhat different from the pebble game which appears in [8]. We show that when the game is played by two persons the problem to determine whether there is a winning strategy is complete in exponential time, and when played by one person, the problem to determine whether one can put a pebble on a particular place (called the solvability problem) is complete in polynomial space. Then we consider various classes of restricted pebble games and study their complexity classes. In particular, it has been shown that the problem of determining whether there is a winning strategy in a game played by two persons is harder in a sense than the solvability problem played by one person.

Our results are summerized in Table 1.1, where NLOGSPACE, P, NP, PS, EXP stand for nondeterministic log space, deterministic polynomial time, nondeterministic polynomial time, polynomial space, deterministic exponential time, respectively.

TABLE 1.1

	Solvability problem (played by one person)	Winning strategy problem (played by two persons)
Pebble game of fixed rank	NLOGSPACE complete	P complete
Acyclic pebble game	NP complete	PS complete
Pebble game	PS complete	EXP complete

* Received by the editors June 22, 1978.
† Research Institute for Mathematical Sciences, Kyoto University, Kyoto, Japan.
‡ Academic and Scientific Programs, IBM Japan Ltd., Roppongi, Tokyo, Japan.
§ Department of Information Science, Sagami Institute of Technology, Fujisawa, Kanagawa, Japan.

The basic results are applied to show that certain problems are complete in exponential time. We consider a game, so called "Chinese checkers game," and a game similar to the "Towers of Hanoi." It has been shown that the winning strategy problems of these games are exponential time complete. As a corollary, we have that the reachability problem for some restricted class of vector addition systems is complete in polynomial space.

2. Preliminaries. In this section, the basic objects with which we are concerned are reviewed. For additional details and background, see [1], [2], [7].

By Turing machine, we mean a machine with a finite-state control, a two-way read-only input tape and a single two-way read-write work tape; the machine halts whenever it enters the accepting state.

Let w be the input to a Turing machine and $|w| = n$. DTIME($T(n)$) is defined to be the class of languages accepted by deterministic Turing machines within $T(n)$ time. NTIME($T(n)$) is defined analogously for nondeterministic Turing machines. Similarly, DSPACE($S(n)$) and NSPACE($S(n)$) are defined to be the classes of languages accepted within $S(n)$ space by deterministic and nondeterministic Turing machines, respectively. Now, let

$$\text{NLOGSPACE} = \bigcup_{i \geq 0} \text{NSPACE}(i \cdot \log n),$$

$$\text{P} = \bigcup_{i \geq 0} \text{DTIME}(n^i),$$

$$\text{NP} = \bigcup_{i \geq 0} \text{NTIME}(n^i),$$

$$\text{PS} = \bigcup_{i \geq 0} \text{DSPACE}(n^i) = \bigcup_{i \geq 0} \text{NSPACE}(n^i),$$

$$\text{EXP} = \bigcup_{i \geq 0} \text{DTIME}(2^{n^i}).$$

Let Γ be a set of tape symbols. A function $f: \Gamma^* \to \Gamma^*$ is *computable in log-space* if and only if there is a deterministic Turing machine M additionally equipped with a one-way write-only output tape such that for any input w of M, M halts with $f(w)$ on its output tape, having scanned no more than log ($|w|$) work tape symbols. Let $L, L' \subseteq \Gamma^*$. L is *log-space reducible* to L' if and only if there is a function f computable in log-space such that for any input w, $w \in L$ if and only if $f(w) \in L'$. For a class of languages C, a language L is called C *complete* if L is in C, and L' is log-space reducible to L for any language L' in C.

DEFINITION [2]. A *k-tape alternating Turing machine* (ATM for short) is an 8-tuple $M = (Q, \Sigma, \Gamma, \delta, b, q_1, q_a, U)$ where:

(1) Q, q_1, q_a are the finite set of *states*, *initial state*, *accepting state*, respectively, $q_1, q_a \in Q$.

(2) Σ, Γ are the finite set of *input symbols* and the set of *tape symbols* respectively, with $\Sigma \subseteq \Gamma$.

(3) b, in $\Gamma - \Sigma$, is the *blank* symbol.

(4) δ, the *next move function*, maps a subset of $Q \times \Sigma \times \Gamma^k$ to subsets of $Q \times \Gamma^k \times \{-1, 0, +1\}^{k+1}$. An element of $\{-1, 0, +1\}^{k+1}$ represents changes of head locations of the input tape and k work tapes.

(5) U is a set of *universal* states, $U \subseteq Q$.

(6) $Q - U$ is a set of *existential* states.

The ATM has a read-only input tape, with the reading head initialized to the first symbol of the input. A *configuration* of the ATM consists of the state, head positions and contents of the $k+1$ tapes. Each of k work tapes is initially blank. A move of the ATM consists of reading one symbol from each of $k+1$ tapes, and then writing one symbol on each work tape and moving the heads as allowed by δ, along with a state-change of the ATM. If C is a configuration of M, let $Next_M(C)$ denote the set of possible configurations after one move of M. We say a configuration is *existential* (respectively *universal, initial, accepting*) if the state of the ATM in that configuration is an existential (respectively universal, initial, accepting) state.

Let C be a configuration of an ATM. A *value* $v(C)$ of C is either *true* or *false* defined by followings:

(1) If C is an accepting configuration, $v(C)$ is true.

(2) If C is an existential configuration but not an accepting configuration, and there is $C' \in Next_M(C)$ such that $v(C')$ is true, then $v(C)$ is true.

(3) If C is a universal configuration but not an accepting configuration, and for every configuration $C' \in Next_M(C)$, $v(C')$ is true, then $v(C)$ is true.

M *accepts* the input w if and only if $v(C_0)$ is true where C_0 is the initial configuration.

We say an ATM is *standard* if (1) M has only one work tape with the head initialized to the first cell of the tape, (2) if a configuration C of M is existential (universal), then every configuration $C' \in Next_M(C)$ is universal (existential), (3) the initial state is existential and the accepting state is universal, and (4) $Next_M(C) = \varnothing$ if and only if C is an accepting configuration.

We let ATIME($T(n)$) and ASPACE($S(n)$) denote the classes of languages accepted by ATM's within time $T(n)$ and within space $S(n)$, respectively.

LEMMA 2.1 [2]. *Let* $S(n) \geqq \log n$ *and* $T(n) \geqq n$. *If* $L \in \mathrm{ASPACE}(S(n))$, *then* L *is accepted by a standard ATM within space* $S(n)$. *If* $L \in \mathrm{ATIME}(T(n))$, *then* L *is accepted by a standard ATM within time* $O(T^2(n))$.

THEOREM 2.1 [2].

$$\mathrm{EXP} = \bigcup_{i \geqq 0} \mathrm{ASPACE}(n^i),$$

$$\mathrm{PS} = \bigcup_{i \geqq 0} \mathrm{ATIME}(n^i),$$

$$\mathrm{P} = \bigcup_{i \geqq 0} \mathrm{ASPACE}(i \cdot \log n).$$

3. Pebble games.

DEFINITION. A *pebble game* is a quadruple $G = (X, R, S, t)$ where:

(1) X is a finite set of *nodes*; the number of nodes is called the *order* of G.

(2) $R \subseteq \{(x, y, z)\colon x, y, z \in X, x \neq y, y \neq z, z \neq x\}$ is called a set of *rules*. For $A, B \subseteq X$, we write $A \vdash B$ if $(x, y, z) \in R$, $x, y \in A$, $z \notin A$, and $B = (A - \{x\}) \cup \{z\}$. We say the move $A \vdash B$ is made by the rule (x, y, z). A symbol $\overset{*}{\vdash}$ denotes the reflexive and transitive closure of $\vdash$.

(3) S is a subset of X; the number of nodes in S is called the *rank* of G.

(4) t is a node in X, called the *terminal* node.

A pebble game is said to be *solvable* if there exists $A \subseteq X$ such that $S \overset{*}{\vdash} A$ and $t \in A$.

At the beginning of a pebble game, pebbles are placed on all nodes of S. If $(x, y, z) \in R$ and pebbles are placed on x, y and not on z, then we can move a pebble from x to z. The game is solvable if we can place a pebble on the terminal node t by moving pebbles according to rules.

A pebble game *played by two persons* is a game between two players, P_1 and P_2, who alternatively move pebbles on the pebble game, with P_1 playing first. The winner is the first player who can put a pebble on the terminal node, or who can make the other player unable to move.

By the term "one-person pebble game problem," we mean the problem to determine for a given pebble game G, whether G is solvable. By "two-person pebble game problem," we mean the problem when a pebble game is played by two persons to determine whether the first player has a winning strategy, i.e., a way to win the game.

THEOREM 3.1. *A two-person pebble game problem is EXP complete.*

Proof. It should be clear that this problem is in EXP in the size of the representation of the game.

Let $M = (Q, \Sigma, \Gamma, \delta, b, q_1, q_a, U)$ be a standard ATM such that only $p(n)$ cells are available on the work tape for some polynomial p in n, where n is the length of input w of M. Since $\mathrm{EXP} = \bigcup_{i \geqq 0} \mathrm{ASPACE}(n^i)$ by Theorem 2.1, it suffices to construct a pebble game G' such that the construction is performed within log-space and M accepts w if and only if the first player P_1 has a winning strategy in G'. Let $Q = \{q_1, \cdots, q_s\}$ and $w = w_1 w_2 \cdots w_n$ $(w_i \in \Sigma, i = 1, 2, \cdots, n)$.

Let $G' = (X', R', S', t')$ where:

(1) $X' = \{[q, i, l]: q \in Q, 1 \leqq i \leqq n, 1 \leqq l \leqq p(n)\}$
$\cup \{[l, a]: 1 \leqq l \leqq p(n), a \in \Gamma\}$
$\cup \{[q, i, l, a, a']: q \in Q, 1 \leqq i \leqq n, 1 \leqq l \leqq p(n), a, a' \in \Gamma\}$
$\cup \{s_1, s_2, t'\}$.

(2) R' is defined as follows:

(2.1) for each $q \in Q$, $a \in \Gamma$, i $(1 \leqq i \leqq n)$, l $(1 \leqq l \leqq p(n))$,

(2.1.1) if $\delta(q, w_i, a)$ contains $(q', a', (d', d''))$, $a \neq a'$ then let

$$([q, i, l], [l, a], [q, i, l, a, a']),$$

$$([l, a], [q, i, l, a, a'], [l, a']),$$

$$([q, i, l, a, a'], [l, a'], [q', i + d', l + d''])$$

be elements of R';

(2.1.2) if $\delta(q, w_i, a)$ contains $(q', a, (d', d''))$ then let

$$([q, i, l], [l, a], [q', i + d', l + d''])$$

be an element of R';

(2.2) for each i $(1 \leqq i \leqq n)$, l $(1 \leqq l \leqq p(n))$, let

$$([q_a, i, l], s_1, s_2)$$

be elements of R';

(2.3) let (s_2, s_1, t') be an element of R'.

(3) $S' = \{[q_1, 1, 1], s_1\} \cup \{[l, b]: 1 \leqq l \leqq p(n)\}$.

A pebble on a node of the form $[q, i, l]$ represents that the current state of the ATM M is q and that the current head positions of the input tape and the work tape are on the ith cell and on the lth cell respectively. A pebble on a node $[l, a]$ represents that symbol a is written on the lth cell of the work tape and a pebble on a node of the form

$[q, i, l, a, a']$ means that M is to change symbol a to a' on the lth cell of the work tape and that M is in state q at the head position i on the input tape. Two nodes s_1 and s_2 are added to enable the player to put a pebble on t' who first put a pebble on a node of the form $[q_a, i, l]$. Thus pebbles on all nodes of S' imply the initial configuration of M. It is clear that the construction can be performed within log-space. We now show that M accepts w if and only if the player P_1 wins the game G'.

Suppose that M accepts w. Then the value of the initial configuration of M is true. Thus, for every true-valued existential configuration C_1, there is a true-valued universal configuration C_1' such that $C_1' \in Next_M(C_1)$, and for every true-valued universal configuration C_2 except the accepting configurations, $C_2' \in Next_M(C_2)$ implies that C_2' is a true-valued existential configuration. Each move of M corresponds either to three consecutive moves of G' induced by the rules in (2.1.1) or to one move induced by (2.1.2). In case a move of M changes a symbol on the work tape, player P_1 moves a pebble by the first rule of (2.1.1) corresponding to a move of M from a true-valued existential configuration to a true-valued universal configuration. Then player P_2 has to move a pebble by the second rule of (2.1.1). After that, P_1 moves a pebble by the third rule of (2.1.1). In case the move of M does not change the symbol on the work tape, P_1 moves a pebble by a rule of the form (2.1.2). Then it is P_2's turn. Whatever rules P_2 may choose, the moves correspond to moves of M from a true-valued universal configuration to a true-valued existential configuration.

Since the initial configuration is existential and the accepting configurations are universal, P_1 can first place a pebble on a node of the form $[q_a, i, l]$, then P_2 has to place a pebble on s_2 and P_1 can place a pebble on the terminal node t' of G'.

Therefore, P_1 has a winning strategy.

Suppose that M does not accept w. Then the value of the initial configuration of M is false. In this case, player P_2 can always move pebbles to nodes in G' which correspond to false-valued configurations of M. Thus, P_1 cannot win.

COROLLARY. *A one-person pebble game problem is PS complete.*

Proof. It should be clear that the problem is in PS. Let M be a nondeterministic Turing machine which accepts an input w, $|w| = n$, within polynomial space. We construct a pebble game G such that M accepts w if and only if G is solvable. Note that if all universal states of an ATM are treated as existential states, then the ATM behaves as a nondeterministic Turing machine. Hence we can treat the ATM in the proof of Theorem 3.1 as a nondeterministic Turing machine. Now let G be the pebble game constructed in the proof of Theorem 3.1; then it is clear that M accepts w if and only if G is solvable.

DEFINITION. A pebble game $G = (X, R, S, t)$ is *acyclic* if the digraph (X, E) is acyclic, where

$$E = \{(x, z), (y, z) : (x, y, z) \in R\}.$$

THEOREM 3.2. *A two-person acyclic pebble game problem is PS complete.*

Proof. The maximum number of moves made in an acyclic pebble game $G = (X, R, S, t)$ is less than $|X| \cdot |S|$, since each pebble can move at most $|X|$ times. Thus this problem is in PS.

Let M be a $p_0(n)$-time bounded standard ATM, where n is the length of an input w of M, and p_0 is a polynomial in n. Then we can construct a pebble game $G_1 = (X_1, R_1, S_1, t_1)$ as in the proof of Theorem 3.1 such that M accepts w if and only if the first player can win in G_1 within $p_1(n) = 3 \cdot p_0(n)$ moves. Now we construct an acyclic pebble game $G_2 = (X_2, R_2, S_2, t_2)$ such that the first player wins in G_2 if and only if the

first player wins in G_1 within $p_1(n)$ moves. Let $G_2 = (X_2, R_2, S_2, t_2)$ where:

$$X_2 = \{[x, i]: x \in X_1, 0 \leqq i \leqq p_1(n)\},$$

$$R_2 = \{([x, i], [y, j], [z, \max(i, j)+1]): (x, y, z) \in R_1, z \neq t_1,$$
$$0 \leqq i, j < p_1(n)\} \cup \{([x, i], [y, j], [z, p_1(n)]): (x, y, z) \in R_1,$$
$$z = t_1, 0 \leqq i, j < p_1(n)\},$$

$$S_2 = \{[x, 0]: x \in S_1\},$$

$$t_2 = [t_1, p_1(n)].$$

It is obvious that the pebble game G_2 is acyclic. It is also obvious that the first player has a winning strategy in G_2 if and only if the first player has a winning strategy in G_1 within $p_1(n)$ moves. Thus M accepts w if and only if the first player has a winning strategy in G_2. Note that the construction of G_2 from M is performed within log-space. Since $\mathrm{PS} = \cup_{i \geqq 0} \mathrm{ATIME}(n^i)$, the problem is PS complete.

COROLLARY. *A one-person acyclic pebble game problem is NP complete.*

Proof. Since the maximum number of moves made in an acyclic pebble game $G = (X, R, S, t)$ is less than $|X| \cdot |S|$, the solvability problem is in NP. We can show that for any nondeterministic Turing machine M, there is an acyclic pebble game G_2 such that M accepts input w within polynomial time in $|w|$ if and only if G_2 is solvable by the same construction method as in the proof of Theorem 3.2.

DEFINITION. Let $G = (X, R, S, t)$ be a pebble game. G is called a pebble game of *fixed rank* if the number of nodes in S is fixed.

THEOREM 3.3. *A two-person pebble game problem of fixed rank is P complete.*

Proof. It is clear that the problem is in P.

Let $M = (Q, \Sigma, \Gamma, \delta, b, q_1, q_a, U)$ be a log n space bounded standard ATM, where n is the length of the input $w = w_1 w_2 \cdots w_n$ of M. Now let $G = (X, R, S, t)$ be a pebble game of rank 3 where:

(1) $X = \{[q, i, l]: q \in Q, 1 \leqq i \leqq n, 1 \leqq l \leqq \lceil \log n \rceil\}$

$$\cup \{[\alpha]: |\alpha| = \lceil \log n \rceil, \alpha \in \Gamma^*\}$$

$$\cup \{[q, i, l, a, a']: q \in Q, 1 \leqq i \leqq n, 1 \leqq l \leqq \lceil \log n \rceil, a, a' \in \Gamma\}$$

$$\cup \{s_1, s_2, t\}.$$

(2) R is defined as follows:

(2.1) for each $q \in Q$, i $(1 \leqq i \leqq n)$, $a \in \Gamma$, l $(1 \leqq l \leqq \lceil \log n \rceil)$, β, $\gamma \in \Gamma^*$ such that $|\beta a \gamma| = \lceil \log n \rceil$, $|\beta| = l - 1$,

(2.1.1) if $\delta(q, w_i, a)$ contains $(q', a', (d', d''))$, $a \neq a'$, then let

$$([q, i, l], [\beta a \gamma], [q, i, l, a, a']),$$

$$([\beta a \gamma], [q, i, l, a, a'], [\beta a' \gamma]),$$

$$([q, i, l, a, a'], [\beta a' \gamma], [q', i + d', l + d''])$$

be elements of R;

(2.1.2) if $\delta(q, w_i, a)$ contains $(q', a, (d', d''))$ then let

$$([q, i, l], [\beta a \gamma], [q', i + d', l + d''])$$

be an element of R;

(2.2) for each i $(1 \leqq i \leqq n)$, l $(1 \leqq l \leqq \lceil \log n \rceil)$, let $([q_a, i, l], s_1, s_2)$ be elements of R;

(2.3) let (s_2, s_1, t) be an element of R.

(3) $S = \{[q_1, 1, 1], [bb \cdots b], s_1\}$.

It can be shown that M accepts w if and only if the first player wins the game G by a similar argument as in the proof of Theorem 3.1. Note that the construction of G is performed within log-space and that the rank of G is 3. Since $\mathrm{P} = \bigcup_{i \geq 0} \mathrm{ASPACE}(i \cdot \log n)$, the problem is P complete.

COROLLARY. *A one-person pebble game problem of fixed rank is NLOGSPACE complete.*

Proof. Clearly this problem is in NLOGSPACE. We can construct a pebble game G of rank 3 as in the proof of Theorem 3.3 such that a $\log n$ space bounded nondeterministic Turing machine accepts input if and only if G is solvable. Note that the construction of G is performed within log-space and an ATM would behave as a nondeterministic Turing machine if each universal state is treated as an existential state.

4. Applications. In this section, the basic results are applied to show that certain games are EXP complete.

DEFINITION. A *Chinese checkers game* is $G = (N, E, W, B, t)$, where N is a finite set of *nodes*, $E \subseteq N^2$ is the set of *edges*, W and B are subsets of N such that $W \cap B = \varnothing$, and t is an element of N.

A Chinese checkers game G is a game played on the graph (N, E) between two players, White and Black. White moves first. Initially, white stones are placed on each node of W and black stones are placed on each node of B. Suppose that (x, y) and (y, z) are edges of E. If there are a white stone on x, a black stone on y and no stone on z, then White in his turn can move the stone from x to z. Similarly, if a black stone is on x, a white stone is on y and no stone is on z, then Black can move the black stone from x to z. The player wins if after his move he has a stone on his color on the node t or the other player cannot move any stone of his color.

THEOREM 4.1. *The problem to determine whether there is a winning strategy in a Chinese checkers game is EXP complete.*

Proof. Since it is easily shown that the problem is in EXP, it suffices to show that two-person pebble game problem is log-space reducible to this problem.

Let $G = (X, R, S, t)$ be a pebble game. We construct a Chinese checkers game G' such that the first player has a winning strategy in G if and only if White has a winning strategy in G'. For each node x in X, we introduce 8 nodes x, $\bar{x}$, $x1$, $\bar{x}1$, $x2$, $\bar{x}2$, $x3$ and $\bar{x}3$, and 8 edges shown in Fig. 4.1. Let $N_X(x) = \{x, \bar{x}, x1, \bar{x}1, x2, \bar{x}2, x3, \bar{x}3\}$, let $E_X(x)$

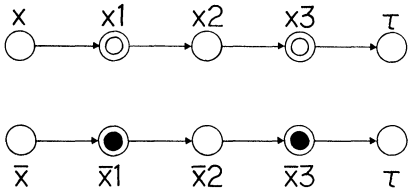


FIGURE 4.1

be a set of 8 edges shown in Fig. 4.1, let $W_X(x) = \{x1, x3\}$ and let $B_X(x) = \{\bar{x}1, \bar{x}3\}$. For each rule $r = (x, y, z)$ in R, we introduce nodes and edges shown in Fig. 4.2, where nodes $x, \bar{x}, y, \bar{y}, z, \bar{z}$ are defined previously and τ is the target node. In Fig. 4.2 the box with a white stone and the box with a black stone stand for a graph shown in Fig. 4.3(1) and a graph in Fig. 4.3(2) respectively. Let $N_R(r)$ be a set of nodes which appear in Fig. 4.2 except $x, \bar{x}, y, \bar{y}, z, \bar{z}$ and τ, and let $E_R(r)$ be a set of edges which appear in Fig. 4.2. $E_R(r)$

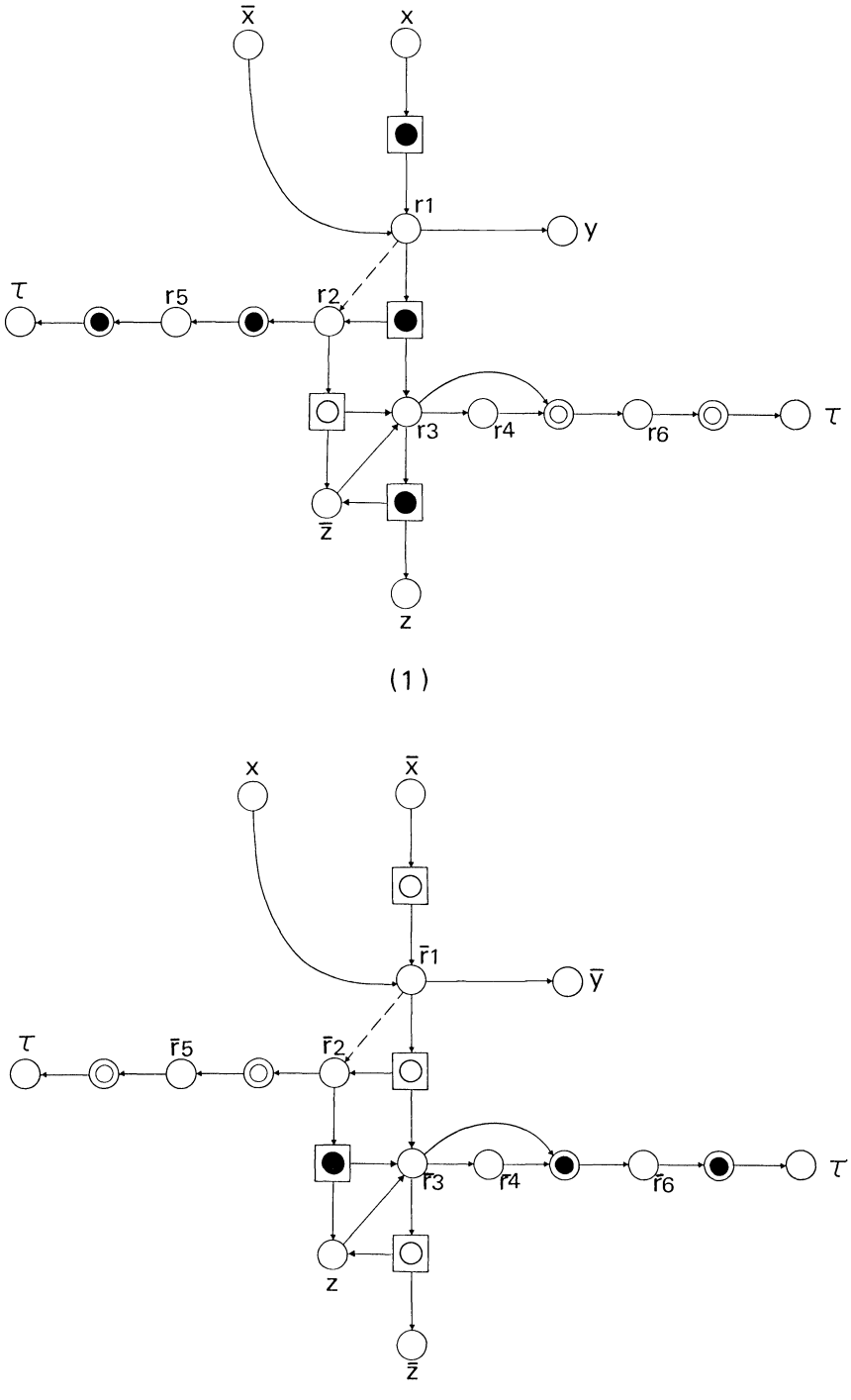


FIGURE 4.2

contains edges of the form $(r1, r2)$ and $(\bar{r}1, \bar{r}2)$, dotted edges shown in Fig. 4.2, if and only if $z \neq t$. Let $W_R(r)$ and $B_R(r)$ be sets of nodes in Fig. 4.2 which are marked nodes

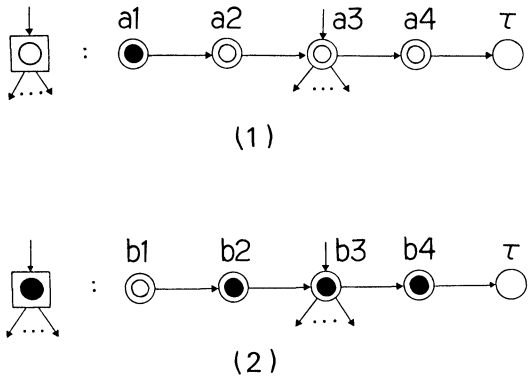


FIGURE 4.3

with white and black respectively. Now let $G' = (N, E, W, B, \tau)$, where:

$$N = \Big(\bigcup_{x \in X} N_X(x)\Big) \cup \Big(\bigcup_{r \in R} N_R(r)\Big) \cup \{\tau\},$$

$$E = \Big(\bigcup_{x \in X} E_X(x)\Big) \cup \Big(\bigcup_{r \in R} E_R(r)\Big),$$

$$W = \{x : x \in S\} \cup \Big(\bigcup_{x \in X} W_X(x)\Big) \cup \Big(\bigcup_{r \in R} W_R(r)\Big),$$

$$B = \{\bar{x} : x \in S) \cup \Big(\bigcup_{x \in X} B_X(x)\Big) \cup \Big(\bigcup_{r \in R} B_R(r)\Big).$$

If the player moves the stone from the box, then he loses the game: if White moves his stone from a3 in Fig. 4.3(1) then Black moves his stone from a1 to τ through a3 hence Black wins; and if Black moves his stone from b3 in Fig. 4.3(2) then White wins by the same argument. When White places his stone on $\bar{x}$, $x \in X$, then he wins, since in Fig. 4.1, White can place his stone to τ through $\bar{x}2$. Analogously, when Black places his stone on x, $x \in$ X, then Black wins.

Now we show that the first player moves a pebble from x to z by the rule $r = (x, y, z) \in R$ in G if and only if White moves his stone from x to z in G', and that the second player moves a pebble from x to z in G if and only if Black moves his stone from $\bar{x}$ to $\bar{z}$ in G'. Before and after this simulation of a move of G in G' the following (i), (ii) and (iii) hold.

(i) If a stone is on x, $x \in X$, then it is white and if a stone is on $\bar{x}$, $x \in X$, then it is black.

(ii) For each $x \in X$, a white stone is on x if and only if a black stone is on $\bar{x}$.

(iii) All stones other than the above are as in Fig. 4.1 and Fig. 4.2.

Note that the initial stage of the simulation satisfies (i), (ii) and (iii) above. Suppose that white stones are on x and y and no stone on z. (Thus, black stones are on $\bar{x}$ and $\bar{y}$ and no stone on $\bar{z}$.) We show that White in his turn can move the white stone from x to z by the following (1)–(5) procedures (black stone on $\bar{x}$ thus moves to $\bar{z}$), which corresponds that the first player moves a pebble from x to z by the rule (x, y, z) in G. Fig. 4.2(1) is used for White's turn.

(1) White moves his stone from x to r1. (If there is no stone on y, then White loses since Black can move the stone on $\bar{x}$ to y.)

(2) In case $z \neq t$ Black must move his stone from $\bar{x}$ to r2. Otherwise, White moves his stone from r1 to τ through r2 and r5, then he wins. In case $z = t$, White wins since (r1, r2) is not in E.

(3) White must move his stone from r1 to r3. Otherwise, Black moves the stone from r2 to τ through r3 and r6, then Black wins. If there has been a black stone on $\bar{z}$, then White loses since Black can move his stone from $\bar{z}$ to r4, and then r6 and τ.

(4) Black must move his stone from r2 to $\bar{z}$. Otherwise White moves the stone from r3 to $\bar{z}$, then White wins.

(5) White must move his stone from r3 to z. Otherwise, Black moves the stone from $\bar{z}$ to r4 and r6, τ, then Black wins.

We note that conditions (i), (ii) and (iii) above still hold after these procedures if $z \neq t$. Analogously, it can be shown that Black in his turn can move his stone from $\bar{x}$ to $\bar{z}$ and also move white stone from x to z. (See Fig. 4.2(2).) If $z = t$ and it is White's turn, then White wins since (r1, r2) is not in E and Black cannot move his stone from $\bar{x}$ to r2 after White moves his stone from x to r1 in Fig. 4.2(1). Similarly, if $z = t$ and it is Black's turn then Black wins.

Thus the first player in G has a winning strategy if and only if White has a winning strategy in G'.

Now we consider a game similar to the well-known game called the Towers of Hanoi.

DEFINITION. Let Z be the set of integers, and let N be the set of natural numbers. An *n-dimensional vector addition system* [10], [12] is a finite subset of Z^n. Let V be an n-dimensional vector addition system. Then the relation $\vdash_V$ over N^n is defined as follows. We write $v \vdash_V w$ if and only if there exists $z \in Z^n$ such that

$$w(i) = v(i) + z(i) \quad \text{for all } i,\ 1 \leqq i \leqq n,$$

where $w(i)$, $v(i)$ and $z(i)$ denote the ith component of w, v and z respectively. Let $\vdash_V^*$ denote the reflexive and transitive closure of $\vdash_V$. The *reachability problem* for vector addition systems is the problem to decide whether $x \vdash_V^* y$ for given V, x and y.

A *conservative vector addition system* is a vector addition system V such that $(v_1, \cdots, v_n) \in V$ implies $v_1 + v_2 + \cdots + v_n = 0$.

DEFINITION. A *peg game* is $G = (V, m, n)$, where V is an n-dimensional conservative vector addition system, and $m, n \in N$. Elements of V are called rules. We say that G is *solvable* if

$$(m, 0, 0, \cdots, 0) \vdash_V^* (0, 0, \cdots, 0, m).$$

A peg game can be considered as the game described as follows. There are n pegs fixed upright on a board, and m disks. Each disk has a hole it its center. An element $y = (y_1, \cdots, y_n)$ of N^n represents that y_i disks are threaded on the ith peg, $i = 1, 2, \cdots, n$. A rule $v = (v_1, \cdots, v_n) \in V$ means that for each i, if $v_i \geqq 0$ then we put v_i disks on the ith peg, and if $v_i < 0$ we remove v_i disks from the ith peg. (See Fig. 4.4.) Initially, all disks are threaded on the first peg. The object of the game is to transfer all disks to the

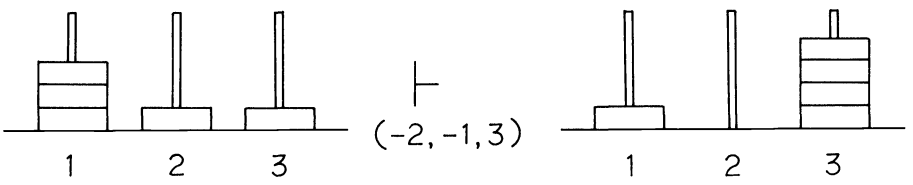


FIGURE 4.4

nth peg. In the two-person game, when two players alternatively move disks by the rules, the player wins if after his move all disks are on the nth peg.

THEOREM 4.2. *A two-person peg game problem is EXP complete.*

Proof. Since it is easily shown that the problem is in EXP, it suffices to show that two-person pebble game problem is log-space reducible to a two-person peg game problem.

Let $G=(\{x_1, \cdots, x_n\}, R, S, x_n)$ be a pebble game. We construct a peg game $G'=(V, n+1, 3n+4)$ as follows. Disks on the first $3n$ pegs are used to represent positions of pebbles in G and the last 4 pegs are auxiliary ones in the simulation of G. Note that the $(3n+4)$th peg is the target peg. Let $V=V_1 \cup V_2 \cup V_3$.

V_1 consists of the vectors (rules) u and v defined as follows.

(i)
$$u(l)=\begin{cases} -n-1 & \text{if } l=1, \\ n+1 & \text{if } l=3n+1, \\ 0 & \text{otherwise.} \end{cases}$$

(ii)
$$v(l)=\begin{cases} 1 & \text{if } 1 \leqq l \leqq n,\ x_l \in S, \text{ or } 2n<l \leqq 3n,\ x_l \notin S, \\ -n & \text{if } l=3n+1, \\ 0 & \text{otherwise.} \end{cases}$$

V_1 is the set of rules for the initial set of the simulation. At the beginning, the first player P_1 must apply the rule (i) above. No other rule can be applied. Then, the other player P_2 can apply only the rule (ii). After these applications, there is a disk on the $(3n+1)$st peg, and for each l, $1 \leqq l \leqq n$, a disk is on the lth peg if $x_l \in S$, and on the $(2n+l)$th peg is $x_l \notin S$.

V_2 is the set of rules for the simulation of the rules in G. At any stage of the simulation, the following conditions (a), (b) and (c) hold.

(a) For each l, $1 \leqq l \leqq n$, exactly one disk is on either the lth, the $(n+l)$th or the $(2n+l)$th peg;

(a-1) if there is a pebble on the node x_l in the pebble game G, then a disk is on either the lth or $(n+l)$th peg; and

(a-2) if there is no pebble on x_l in G, then a disk is on the $(2n+l)$th peg.

(b) Exactly one disk is on either the $(3n+1)$st or the $(3n+2)$nd peg.

(c) There is no disk on either the $(3n+3)$rd or the $(3n+4)$th peg.

For each $(x_i, x_j, x_k) \in R$, V_2 contains the vectors v which satisfy conditions (1) to (5).

(1) $v(i)=-1$, $v(n+i)=0$, $v(2n+i)=1$, or $v(i)=0$, $v(n+i)=-1$, $v(2n+i)=1$.

(2) $v(j)=-1$, $v(n+j)=1$, $v(2n+j)=0$, or $v(j)=1$, $v(n+j)=-1$, $v(2n+j)=0$.

(3) $v(k)=1$, $v(n+k)=0$, $v(2n+k)=-1$.

(4) $v(l)=v(n+l)=v(2n+l)=0$ for all $l(1 \leqq l \leqq n)$, $l \notin \{i, j, k\}$.

(5) if $k \neq n$, then

$$v(3n+1)=-1, \quad v(3n+2)=1, \quad v(3n+3)=v(3n+4)=0,$$

or

$$v(3n+1)=1, \quad v(3n+2)=-1, \quad v(3n+3)=v(3n+4)=0;$$

if $k=n$, then

$$v(3n+1)=-1, \quad v(3n+2)=0, \quad v(3n+3)=0, \quad v(3n+4)=1,$$

or

$$v(3n+1)=0, \quad v(3n+2)=-1, \quad v(3n+3)=0, \quad v(3n+4)=1.$$

Suppose that there are pebbles both on x_i and x_j, and no pebble on x_k. Then for $l = i, j$, the condition (a-1) holds, and for $l = k$, the condition (a-2) holds. Suppose that a rule (x_i, x_j, x_k) is applied in the pebble game. Then by the rule (1) above, a disk either on the ith or the $(n+i)$th peg moves to the $(2n+i)$th peg; by (2), a disk either on the jth or the $(n+j)$th peg moves to the $(n+j)$th or the jth peg respectively; and by (3), a disk on the $(2n+k)$th peg moves to the kth peg, and condition (a) still holds. If $k \neq n$, by (5), a disk on either the $(3n+1)$st or the $(3n+2)$nd moves to either the $(3n+2)$nd or the $(3n+1)$st peg, respectively. Conditions (b) and (c) hold in this case. If $k = n$, by (5), a disk either on the $(3n+1)$st or the $(3n+2)$nd peg moves to the $(3n+4)$th peg. Note that a rule in V_2 cannot be applied if there is not any disk either on the $(3n+1)$st or the $(3n+2)$nd peg. For example, if there are disks on the ith, the $(n+j)$th, the $(2n+k)$th and the $(3n+1)$st pegs, and the rule corresponding to the rule (x_i, x_j, x_k) of the pebble game is applied, then these disks are moved to the $(2n+i)$th, the jth, the kth and the $(3n+2)$nd pegs. (See Fig. 4.5.)

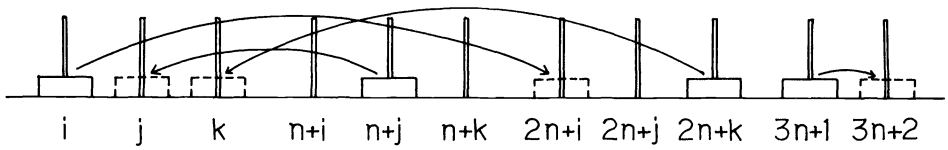


FIGURE 4.5

We note that the first player has a winning strategy in the pebble game if and only if, in the two-person peg game, the first player can first apply the rule which corresponds to the rule of the form (x_i, x_j, x_n), that is, the first player can move the disk on either the $(3n+1)$st or the $(3n+2)$nd peg to the $(3n+4)$th peg.

V_3 is the set of rules for the collection of all disks to the $(3n+4)$th peg. These rules enable the first player who put a disk on the $(3n+4)$th peg to collect all disks on the $(3n+4)$th peg. V_3 consists of two kinds of vectors.

(i)
$$v(l) = \begin{cases} +1 & \text{if } l = 3n+3, \\ -1 & \text{if } l = 3n+4, \\ 0 & \text{otherwise.} \end{cases}$$

(ii) for each i $(1 \leqq i \leqq 3n)$,
$$v(l) = \begin{cases} -1 & \text{if } l = i \text{ or } l = 3n+3, \\ 2 & \text{if } l = 3n+4, \\ 0 & \text{otherwise.} \end{cases}$$

We note that after an application of the vector shown in (i) above, the only applicable rules are in (ii), and that after an application of a vector in (ii), the only applicable rule is the vector (i).

It is straightforward to show that in the pebble game G, the first player has a winning strategy if and only if the first player has a winning strategy in the two-person peg game.

COROLLARY. *A one-person peg game problem is PS complete.*

Proof. Let G' be the peg game constructed from a pebble game G as in the previous proof. Then, it is clear that G' is solvable if and only if G is solvable. Since a one-person pebble game problem is PS complete, the corollary is proved.

COROLLARY. *The reachability problem of conservative vector addition systems is PS complete.*

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] A. K. CHANDRA AND L. J. STOCKMEYER, *Alternation*, Proc. 17th Ann. IEEE Symp. on Foundation of Computer Sciences, 1976, pp. 98–108.

[3] S. A. COOK, *The Complexity of theorem-proving procedures*, Proc. 3rd ACM Symp. on Theory of Computing, 1971, pp. 151–158.

[4] ———, *An observation on time-storage tradeoff*, J. Comput. System Sci., 9 (1974), pp. 213–229.

[5] S. A. COOK AND E. SETHI, *Storage requirements for deterministic polynomial time recognizable languages*, Ibid., 13 (1976), pp. 25–37.

[6] S. EVEN AND R. E. TARJAN, *A combinatorial problem which is complete in polynomial space*, J. Assoc. Comput. Mach., 23 (1976), pp. 710–719.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[8] J. E. HOPCROFT, W. PAUL AND L. VALIANT, *On time versus space*, J. Assoc. Comput. Mach., 24 (1977), pp. 332–337.

[9] W. D. JONES AND W. T. LAASER, *Complete problems for deterministic polynomial time*, Theoretical Comput. Sci., 3 (1977), pp. 105–117.

[10] R. M. KARP AND R. E. MILLER, *Parallel program schemata*, J. Comput. System Sci., 3 (1969), pp. 147–195.

[11] R. M. KARP, *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–104.

[12] R. E. MILLER, *Mathematical studies of parallel computation*, Proc. First IBM Symp. on Mathematical Foundation of Computer Science, IBM Japan, 1976.

[13] T. J. SCHAEFER, *Complexity of decision problems based on finite two-person perfect-information games*, Proc. 8th Ann. ACM Symp. on Theory of Computing, 1976, pp. 41–49.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0009 $01.00/0

TRANSLATABILITY AND DECIDABILITY QUESTIONS FOR RESTRICTED CLASSES OF PROGRAM SCHEMAS*

ELAINE J. WEYUKER†

Abstract. Two new classes of schemas are introduced: the reachable schemas and the semifree schemas. A schema is reachable if every statement in the schema is executed under some interpretation. A schema is semifree if every test in the schema is necessary in the sense that each exit of the test is taken under some interpretation.

It is shown that most of the standard decision problems are unsolvable for schemas in these two classes, and that there can be no algorithm which effectively translates an arbitrary schema into an equivalent reachable or semifree schema, even though such equivalent schemas always exist. These classes are also compared to the free and liberal schemas, and interclass translatability questions are investigated. It is demonstrated that every reachable schema can be effectively translated into a semifree schema, even though it is not decidable whether a reachable schema is semifree.

Key words. program schema, flowchart schema, abstract program, translatability, decision problems

1. Introduction and definitions. For several years, people have studied abstractions of computer programs known as program schemas. A great deal of work has been done comparing the relative computational power of classes of schemas with additional features [1], [2], [8]. We are interested in considering classes of schemas whose members fulfill certain semantic requirements. We introduce two such classes of schemas, the semifree schemas and the reachable schemas, and consider various decision problems for these classes. We also consider the relative power of these classes. We compare them to the class of all schemas as well as to other well-known semantically restricted classes. We use a program schema model based largely on the one formulated by Luckham, Park, and Paterson [6].

We have a formal language whose alphabet consists of the following disjoint sets of symbols:

(i) *Variable or Location Symbols*, denoted by the letters u, v, w, x, y, z. The set of variables is divided into three disjoint subsets X, Y, and Z. The set X contains the *input variables*. Y is the set of *program variables*, and Z is the set of *output variables*. A variable in X may be referenced but never changed, whereas an element of Z may be assigned a value, but may never be referenced. An element of Y may either be referenced or assigned a value provided it has been assigned a value before it is referenced.

(ii) *Function Symbols*, denoted by the letters f, g, h.

(iii) *Predicate Symbols*, denoted by the letters p, q, r, s, t.

(iv) *Distinguished Symbols*: *START*, *HALT*, (,), $\leftarrow$, numerals, comma.

Each of the symbols in (i), (ii), and (iii) may appear with or without a subscript.

The language has four types of statements, known as the *legal statements*:

(i) *Start Statement*: *START*.

(ii) *Assignment Statement*: $y \leftarrow f(y_1, \cdots, y_n)$ where $y_1, \cdots, y_n$ are elements of $X \cup Y$, y is an element of $Y \cup Z$, and f is an n-ary function symbol.

(iii) *Test Statement*: $p(y_1, \cdots, y_n)$ where $y_1, \cdots, y_n$ are elements of $X \cup Y$ and p is an n-ary predicate symbol.

(iv) *Halt Statement*: *HALT*.

* Received by the editors October 14, 1977, and in final revised form November 1, 1978.

† Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, New York 10012.

A *flow diagram* is a labeled directed graph (see [5] for basic graph theory definitions) each of whose vertices is labeled by a legal statement and such that:

(i) A vertex labeled with a start statement has no edges entering it and one edge exiting from it.

(ii) A vertex labeled with an assignment statement has at least one edge entering it and exactly one edge exiting from it.

(iii) A vertex labeled with a test statement has at least one edge entering it and exactly two edges exiting from it which are labeled 0 and 1. Occasionally we will extend the notation to include n-exit tests. This is simply a matter of notational convenience and could equally be represented by a series of $(n-1)$ two-exit tests.

(iv) A vertex labeled with a halt statement has at least one edge entering it and no edges exiting from it.

A *program schema* P is a finite flow diagram with the following restrictions:

(i) There is exactly one vertex labeled *START*.

(ii) Each vertex lies on a path from the vertex labeled *START*.

(iii) On every path from the start statement, if u is a variable in $Y \cup Z$, then u is assigned a value before it is referenced. That is, on every path, u must appear on the left-hand side of an assignment statement before it may appear on the right-hand side of an assignment or test statement. We note that this is a purely syntactic requirement.

The semantics of a schema is provided by an *interpretation* which specifies a domain, assigns actual functions and predicates to the function and predicate symbols of the schema, and also assigns initial values from the domain to each input variable. Throughout this paper we use $I(p(\bar{x}))$ and $I(f(\bar{x}))$ respectively to stand for the value of $I(p)$ and $I(f)$ applied to $I(\bar{x})$.

A *Herbrand interpretation* of a schema P is any interpretation I with the following properties. The domain D is the set of syntactically well-formed strings over the input variables and function symbols of the schema, $I(x) = x$ for every input variable x, and $I(f(u_1, \cdots, u_n)) = fu_1, \cdots, u_n$ for every n-ary function symbol f of P and $u_1, \cdots, u_n \in D$. Note that the interpretation of P's predicate symbols is not restricted in any way.

The *execution sequence* for schema P under interpretation I consists of the sequence of statements of P executed under I. Note that in defining both the syntactic notion of a path, and the semantic notion of an execution sequence, care must be taken in two circumstances. If either more than one exit of a test statement enter the same vertex, or enter distinct vertices labeled with the same statement, then an indication of which exit was selected is necessary to completely specify the path or execution sequence.

For each interpretation I, the computation of the schema P either terminates (i.e. reaches a halt statement), or diverges. In the former case the value, denoted val(P, I), is the n-tuple of current values of P's n output variables. If P diverges under I or if any output variable has never been assigned a value, then val(P, I) is undefined.

We say two schemas, P and Q, are *strongly equivalent*, denoted $P \equiv Q$, if for every interpretation I, either both val(P, I) and val(Q, I) are defined and val$(P, I) =$ val(Q, I), or both values are undefined.

A class of schemas $\mathscr{C}_1$ is *translatable* into a class $\mathscr{C}_2$, if for every $P_1 \in \mathscr{C}_1$, there is a strongly equivalent $P_2 \in \mathscr{C}_2$.

2. Semantically restricted classes of schemas. Paterson [7] introduced the notions of freeness and liberality as semantic restrictions on the class of schemas. He felt that such restricted classes of schema might have solvable decision problems.

A schema is *free* if every finite path through its flow diagram from the start statement is an initial segment of some execution sequence. This property has been shown [7] to be equivalent to the restriction that under no Herbrand interpretation is any predicate symbol of rank n ever applied to an n-tuple of elements of the universe more than once.

A schema is *liberal* if for every Herbrand interpretation, no element of the universe is computed more than once, i.e. each particular n-tuple of elements of the universe is an argument of each function symbol at most once.

A *statement* in a schema is *reachable* if there is an interpretation under which that statement is executed. A *schema* P is *reachable* if every statement in P is reachable.

A schema P is *semifree* if for every edge in the flow diagram of P, there is some interpretation under which that edge is traversed.

We use $\mathscr{F}$, $\mathscr{L}$, $\mathscr{R}$, $\mathscr{S}$, and $\mathscr{P}$ to represent the classes of free, liberal, reachable, semifree, and all schemas, respectively. It follows easily from our definitions that $\mathscr{F} \subseteq \mathscr{S} \subseteq \mathscr{R} \subseteq \mathscr{P}$.

The schema of Fig. 1 is an example of a nonreachable schema, as the *HALT* can never be executed. If we modify the schema and have the 0-exit of the second $p(y)$ test (labeled 2) enter the halt statement rather than returning to point @, the schema is reachable but not semifree, as the 1-exit of this test may never be taken. If in addition to the above modification of the schema of Fig. 1, we have the 1-exit of the first $p(y)$ test (labeled 1) enter the second $p(y)$ test and delete the assignment statement $y \leftarrow f(y)$, the resulting schema is semifree but not free. Thus it follows that each of the inclusions is strict.

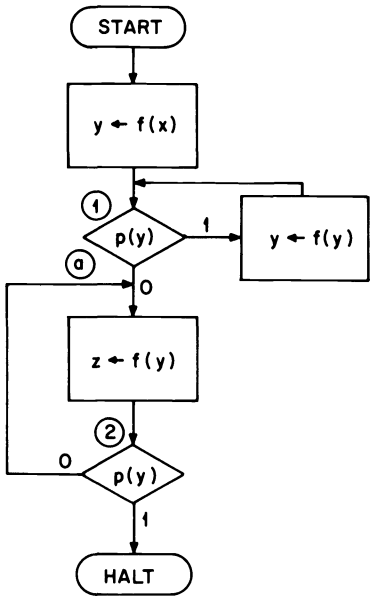


FIG. 1

2.1. Reachable schemas. It has been shown [7] that $\mathscr{P}$ is not translatable into $\mathscr{F}$ or $\mathscr{L}$. In contrast to these results, it is interesting to note that corresponding to every schema P, there is a semifree (and hence reachable) version Q. Unnecessary tests of P must be removed, as well as any unreachable code. To remove an unnecessary test, its direct predecessors must first be connected to the statement at the exit which is always

taken, and then the test may be deleted. Pieces of code which become syntactically inaccessible as a result of such modifications must then also be removed. Thus we have immediately:

PROPOSITION 1. *$\mathcal{P}$ is translatable into $\mathcal{S}$, and hence into $\mathcal{R}$.*

We see, however, that this translation is not effective.

LEMMA 2. *It is decidable whether a reachable schema P halts under some interpretation.*

Proof. P halts under some interpretation iff it contains a halt statement. □

THEOREM 3. *There is no algorithm which, given an arbitrary schema P, constructs a reachable schema Q, such that $P \equiv Q$.*

Proof. Assume such an algorithm existed. Then by Lemma 2, it would be decidable whether Q, and hence P, halted under some interpretation. But this is a well-known [7] undecidable property for arbitrary schemas. □

Thus although for every schema there is a strongly equivalent reachable schema, we cannot in general effectively obtain it. We next see that reachability is an undecidable property for schemas.

The notion of reachability in schemas is analogous to state accessibility in Turing machines, and as the next results indicate, reachability is an undecidable property of schemas for precisely the same reasons that state accessibility is an undecidable property of Turing machines.

LEMMA 4. *It is not decidable whether an arbitrary assignment statement of a schema is reachable.*

Proof. Let P be an arbitrary schema. We assume without loss of generality that P contains a single halt statement. We construct a schema Q from P by replacing the halt statement of P by an assignment statement s, followed by a halt statement. Then if it were decidable whether s is reachable in Q, it would be decidable whether P halted under some interpretation. □

THEOREM 5. *It is not decidable whether a schema is reachable.*

Proof. We prove this theorem by showing that there is an algorithm which, given an arbitrary schema P and assignment statement s_k of P, constructs a schema Q such that Q is reachable iff s_k is reachable in P. Let P be an arbitrary schema with statements $s_1, \cdots, s_n$. Let s_k be an assignment statement of P. Let p be an n-exit predicate symbol which does not appear in P. We construct schema Q from P by inserting an initializing assignment statement, $\bar{y} \leftarrow f(x)$, immediately following the start statement. The notation indicates that we are assigning the value $f(x)$, where x is some input variable, to every program variable y. This is done simply to gurantee that every program variable is assigned a value before it is referenced. We also insert the test $p(x)$ after instruction s_k. The branch from the mth exit of p enters statement s_m. The construction of Q is outlined in Fig. 2. Clearly s_k is reachable in P iff Q is a reachable schema. □

Before considering similar questions for $\mathcal{S}$, we consider two other decision problems for $\mathcal{R}$.

LEMMA A (Paterson [7]). *It is not decidable whether an arbitrary schema P halts under every interpretation.*

THEOREM 6. *It is not decidable whether a reachable schema halts under every interpretation.*

Proof. We present an algorithm which for a given schema P, produces a reachable schema Q which halts under every interpretation iff P does.

Assume P contains $n+1$ statements labeled $0, 1, \cdots, n$ with 0 the start statement and 1 the unique successor of the start statement. We assume without loss of generality, that P contains exactly one halt statement, and that it is labeled n.

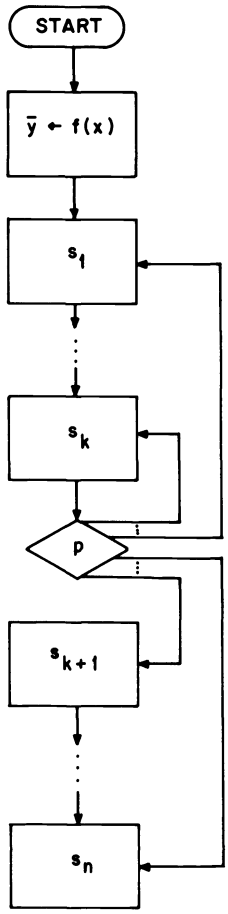


FIG. 2

Let $p, q_1, \cdots, q_{n-1}, r$ be predicate symbols which do not appear in P. Intuitively, Q can be in one of two distinct modes, depending on the interpretation of the test $p(x)$. If $I(p(x)) = 1$, then the schema is in "reachability mode." This mode guarantees that every instruction of Q can be reached. If $I(p(x)) = 0$, then the schema is in "simulation mode," and is effectively simulating the computation of P under interpretation I. Furthermore, since x is an input variable, and thus can never be assigned a new value, once a mode is determined, it cannot be changed. We construct Q as shown in Figs. 3a–d.

We call the subschema of Fig. 3a the initial subschema. $\bar{y} \leftarrow f(x)$ is an initializing assignment statement, as described in Theorem 5. We let $s(i)$ denote the instruction in P labeled i.

Case 1. If $s(i)$, $i = 1, \cdots, n-1$, is an assignment statement in P such that the successor of $s(i)$ is $s(j)$, then in Q we have the subschema shown in Fig. 3b.

Case 2. If $s(i)$, $i = 1, \cdots, n-1$, is a test statement in P with 0-successor $s(k)$, and 1-successor $s(j)$, then in Q we have the subschema shown in Fig. 3c.

Case 3. $s(n)$ is the halt statement in P. Then in Q we have the subschema shown in Fig. 3d.

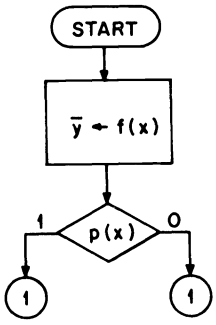


FIG. 3(a)

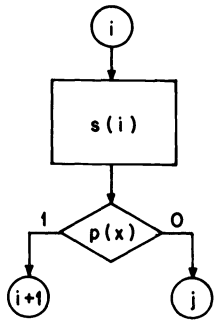


FIG. 3(b)

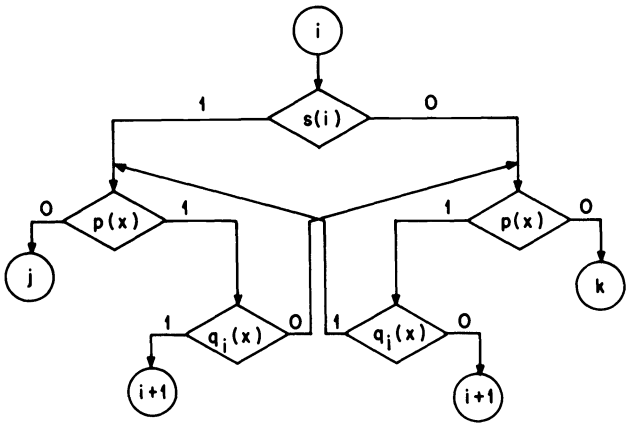


FIG. 3(c)

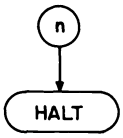


FIG. 3(d)

Note that a schema Q does not in general contain all of the predicate symbols $q_1, \cdots, q_{n-1}$ but contains one q_i for each test statement in P. The extra predicate symbols are included simply to facilitate the notation.

In either mode, we are not interested in the value calculated by Q. If $I(p(x)) = 0$, then P halts under interpretation I if and only if Q halts under interpretation I. If $I(p(x)) = 1$ then Q halts regardless of P's behavior under interpretation I.

We also point out here the reason for the test statements q_i and r. They are only encountered under interpretations which put the schema into reachability mode and are in fact used to guarantee that every statement is reachable. In the case that $s(i)$ is a test statement, we cannot guarantee that there is some interpretation which makes the value of $s(i)$ under that interpretation 1, and some other interpretation such that the value of $s(i)$ is 0. Hence it is possible that one of the two test statements $p(x)$ might not be reachable. By adding the tests $q_i(x)$, where q_i does not appear anywhere else in the schema Q, we can guarantee that both $p(x)$ tests are reachable.

To see that Q is reachable, note that subschema 1 is reachable, that subschema $i+1$ is always reachable from subschema i when $I(p(x)) = 1$, and that every statement within each subschema is reachable when $I(p(x)) = 1$.

Our final task is to show that Q halts under every interpretation if and only if P does. If Q halts under every interpretation, then P halts under every interpretation for which $I(p(x)) = 0$. But p is not a predicate symbol in P's language, and hence P's

behavior under an interpretation must be independent of the interpreted value of p. Thus P halts under every interpretation.

If P halts under every interpretation, then Q halts under every interpretation I such that $I(p(x)) = 0$. Furthermore, Q has been constructed so that under an interpretation I for which $I(p(x)) = 1$, the subschemas $1, 2, \cdots, n$ are executed in order and then the schema halts. Thus Q halts under every interpretation.

Thus we have shown that Q is a reachable schema which is constructed effectively from an arbitrary schema P and such that Q halts under every interpretation if and only if P halts under every interpretation. Therefore, if we could decide whether a reachable schema halts under every interpretation, we could decide whether an arbitrary schema halts under every interpretation, contradicting Lemma A. □

COROLLARY 7. *It is not decidable whether two reachable schemas are strongly equivalent.*

Proof. Let P be a reachable schema with input variable x, and output variables $z_1, \cdots, z_n$. Let Q be the reachable schema shown in Fig. 4a. Note that Q halts under every interpretation.

We construct schema R by replacing each halt statement of P by the sequence of instructions of Fig. 4b. Clearly R is reachable iff P is reachable. Also,

$$Q \equiv R \text{ iff } R \text{ halts under every interpretation}$$
$$\text{iff } P \text{ halts under every interpretation.}$$

Thus if strong equivalence were decidable for reachable schemas, we could decide whether a reachable schema halts under every interpretation, contradicting Theorem 6. □

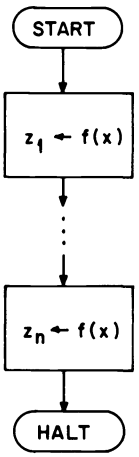


FIG. 4(a)

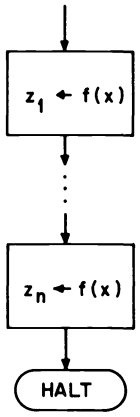


FIG. 4(b)

2.2. Semifree schemas. The semifree schemas represent a restriction on the class of reachable schemas. Not only must we be able to reach every statement in the schema, but we must also be able to leave via any exit. Paterson [7] showed that a schema is free iff it does not contain any repeated tests. Furthermore, there are schemas which are inherently nonfree in the sense that they are not strongly equivalent to any free schema, and thus must repeat some tests in order to do the desired computation. In contrast to this, Proposition 1 stated that there are no inherently nonsemifree schemas.

We say that a test t is *necessary* if there are interpretations I_0 and I_1 such that the 0-exit of t is taken under I_0 and the 1-exit is taken under I_1.

PROPOSITION 8. *A schema P is semifree iff every test in P is necessary.*

THEOREM 9. *There is no algorithm which given an arbitrary schema P, constructs a semifree schema Q, such that* $P \equiv Q$.

Proof. This is a direct consequence of Theorem 3 and the observation that $\mathcal{S} \subseteq \mathcal{R}$. □

We now discuss an interesting situation which underscores the necessity of considering both the letter and the spirit of a result. Our next theorem demonstrates that for any reachable schema, we can construct a strongly equivalent semifree schema. At first glance, that seems like a very desirable situation. It seems to say that if we know that we do not have any unreachable code, then we can effectively get rid of any unnecessary tests. However, the theorem does not really say that at all. The construction used in the proof below will cause just the opposite to happen. Instead of deleting unnecessary tests, we will add additional tests and code in order to force the newly constructed schema to be semifree.

We would really like to begin with a reachable schema, and if it is not semifree, delete exactly the unnecessary tests and be left with a "reduced" strongly equivalent semifree schema. Theorem 14 shall demonstrate that such a procedure is not possible.

THEOREM 10. *There is an algorithm which given an arbitrary schema P, produces a strongly equivalent schema Q, such that Q is semifree iff P is reachable.*

Proof. Assume P contains n test statements, designated $t_1, \cdots, t_n$. Let k_0 and k_1 denote test t_k's 0- and 1-successor respectively, $k = 1, \cdots, n$. Let p, q, and s be predicate symbols not appearing in P.

We build Q, shown in Fig. 5, as follows. It contains two slightly modified copies of schema P, which we call P_L and P_R. If a is the first statement following the start statement of P, then the corresponding statements in P_L and P_R are called a_L and a_R respectively.

For a given interpretation I, Q is entered and one of four conditions holds. If $I(s(x)) = 0$ and $I(p(x)) = 0$ the computation proceeds through P_L exactly as it would through P with the addition of repeated testing of $p(x)$ and $s(x)$. Similarly, if $I(s(x)) = 1$ and $I(p(x)) = 1$, the computation proceeds through P_R. If $I(s(x)) = 0$ and $I(p(x)) = 1$, then q is tested and one of the n tests of P_L is selected and applied. Next $p(x)$ and $s(x)$ are retested and then P_R is entered at a_R and the computation proceeds as it would through P under I. Similarly, if $I(s(x)) = 1$ and $I(p(x)) = 0$ some test of P_R is selected, P_L is entered at a_L and the computation proceeds as through P under I. Thus $P \equiv Q$.

A straightforward, but tedious argument verifies that Q has the required properties, and is omitted here. □

As we mentioned previously, this theorem is interesting since it points out that a semifree schema is not necessarily "optimized," or even an improvement over an equivalent schema which is not semifree. It is also useful as several results follow as corollaries to it.

COROLLARY 11. *It is not decidable whether an arbitrary schema is semifree.*

COROLLARY 12. *It is not decidable whether a semifree schema halts under every interpretation.*

Proof. The proof follows immediately from Theorem 10 and Theorem 6. □

COROLLARY 13. *It is not decidable whether two semifree schemas are strongly equivalent.*

Proof. The proof follows immediately from Theorem 10 and Corollary 7. □

3. Relationships between $\mathcal{R}$ and $\mathcal{S}$ and other semantically restricted classes of schemas. We have seen in the previous section that many properties are not decidable for schemas in $\mathcal{R}$ and $\mathcal{S}$, and in fact even membership in these two classes is

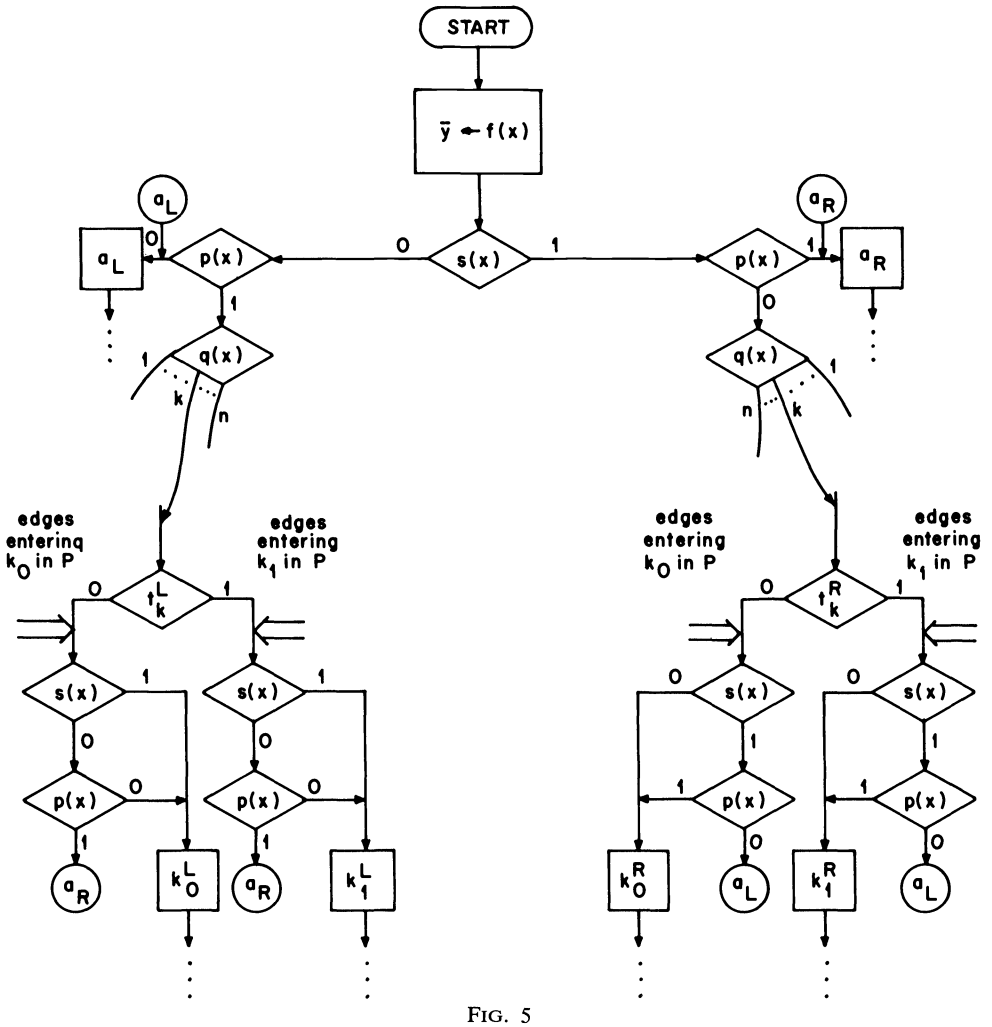


FIG. 5

undecidable. In this section we investigate under what circumstances knowing that a schema possesses certain desirable properties, enables one to decide whether it has other desirable properties.

THEOREM 14. *It is not decidable whether a reachable schema is semifree.*

Proof. Let P be an arbitrary schema. We construct a reachable schema Q which is semifree if and only if P is semifree. Since semifreeness is undecidable for arbitrary schemas, we cannot decide whether a reachable schema is semifree.

Assume P contains n test statements designated $t_1, \cdots, t_n$. The 0-successor of statement t_k is denoted by k_0, and the 1-successor is denoted by k_1. Let t_1 be the first test statement encountered under every interpretation (i.e. t_1 is the test statement nearest to *START*). Then 1_0 and 1_1 denote the 0- and 1-sucessors of t_1, respectively.

Let p be a predicate symbol not appearing in P. Let q be a $2n$-exit predicate symbol not appearing in P. The exits of q are labeled $t_1 0, t_2 0, \cdots, t_n 0, t_1 1, \cdots, t_n 1$. We construct Q from P by inserting $2n+1$ copies of the test statement $p(x)$ and one copy of the test statement $q(x)$ in the flow diagram of P as follows:

(1) Immediately after the start statement of P, insert an initializing statement $\bar{y} \leftarrow f(x)$ followed by the statement $p(x)$. The 0-successor of this test is the statement

which is the successor of the start statement in P; we designate that statement a. The 1-successor of the $p(x)$ statement is the test statement $q(x)$.

(2) Insert one copy of the test statement $p(x)$ as the 0-successor of each test statement t_k, $k = 1, \cdots, n$. Both the 0- and 1-successors of $p(x)$ will be the statement k_0 of P. Similarly, we insert one copy of $p(x)$ as the 1-successor of t_k. Both the 0- and 1-successors of this copy of the test statement $p(x)$ will be statement k_1 of P.

(3) The $2n$ successors of the test statement $q(x)$ are the $2n$ copies of $p(x)$ inserted in step 2 above.

Thus we have constructed the schema Q whose outline is shown in Fig. 6. Intuitively we can think of Q as being divided into $2n + 1$ segments of code, the entrance to each segment being controlled by the interpretation of the initial $p(x)$ test and the $q(x)$ test.

A straightforward argument can be used to verify that the schema Q, so constructed, has the required properties. The complete proof appears in [9]. □

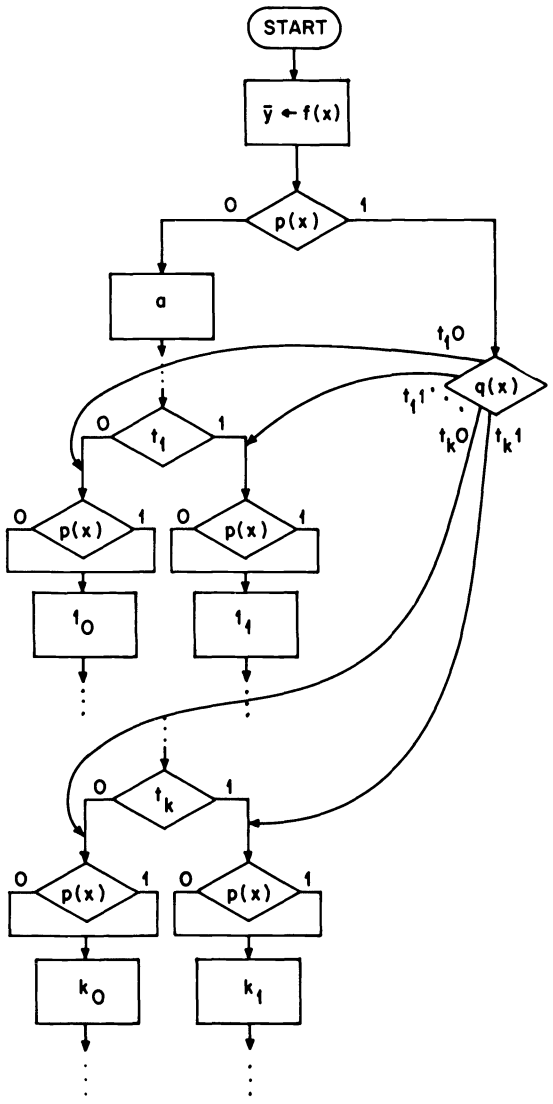


FIG. 6

Since every free schema is semifree, one might hope that semifreeness would be sufficient to allow us to decide freeness. The next theorem tells us that this is not the case.

THEOREM 15. *It is not partially decidable whether a semifree schema is free.*

Proof. Assume such a partial decision procedure existed. It has been demonstrated [9] that semifreeness is partially decidable. Thus we can apply the hypothesized partial decision procedure and select the semifree, free schemas. But $\mathscr{F} \subseteq \mathscr{S}$ and hence we would have a partial decision procedure for freeness, which Paterson [7] showed was not partially decidable. □

Although the decidability of the equivalence problem for free schemas remains an open problem, it is easy to demonstrate that the following related question is unsolvable. The proof uses an argument similar to that of Corollary 7.

THEOREM 16. *It is not decidable whether a semifree schema and a free schema are strongly equivalent.*

THEOREM 17. *There is no algorithm which, given a semifree schema, produces a strongly equivalent free schema.*

Proof. The proof follows from Corollary 12 and the observation that it is decidable whether a free schema halts under every interpretation. □

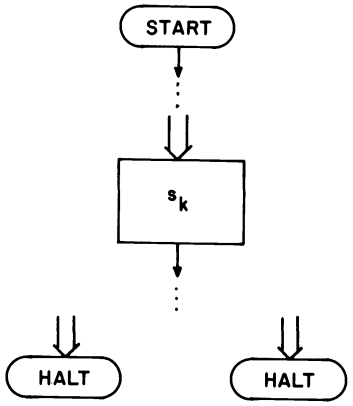
We next consider how $\mathscr{L}$ is related to $\mathscr{R}$ and $\mathscr{S}$. Paterson [7] showed that $\mathscr{L}$ is effectively translatable into $\mathscr{L} \cap \mathscr{F}$. Thus $\mathscr{L}$ is effectively translatable into $\mathscr{R}$ and $\mathscr{S}$. The schema shown in Fig. 1 is a simple example of a liberal schema which is not reachable, and hence neither semifree nor free.

LEMMA 18. *It is decidable whether a liberal schema halts under some interpretation.*

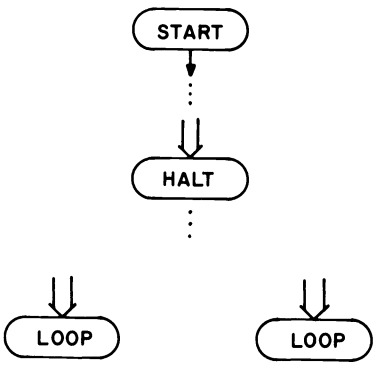
Proof. The proof follows from Paterson's construction, Lemma 2, and the fact that $\mathscr{F} \subseteq \mathscr{R}$. □

THEOREM 19. *It is decidable whether an arbitrary statement in a liberal schema is reachable.*

Proof. Let P be a liberal schema with statements $s_1, \cdots, s_n$. For each k, $1 \leqq k \leqq n$, we construct a liberal schema P_k from P, such that P_k halts under some interpretation, iff s_k is reachable in P. To construct P_k, we replace each halt statement of P by a *LOOP* statement, and statement s_k by a halt statement. If statement s_k was anything other than a halt statement, we remove all edges leaving s_k, and delete any portions of the flow diagram which are disconnected as a result of this replacement. Thus we have the schemas outlined in Figs. 7a and 7b.



SCHEMA P
FIG. 7(a)



SCHEMA P_k
FIG. 7(b)

Since P is liberal, and the construction of P_k from P does not add any new calculations, P_k is also liberal. Furthermore, since P_k contains only the single halt statement, in place of P's s_k, it is clear that P_k halts under some interpretation if and only if s_k is reachable in P. From Lemma 18 it is decidable whether P_k halts under some interpretation, and hence whether s_k is reachable in P. □

COROLLARY 20. *It is decidable whether a liberal schema is reachable.*

Since reachability of individual statements is decidable for liberal schemas, we can remove the unreachable ones. Thus we have:

COROLLARY 21. *There is a procedure which given a liberal schema, effectively constructs a strongly equivalent schema which is liberal and reachable.*

THEOREM 22. *It is decidable whether an arbitrary test statement of a liberal schema is necessary.*

Proof. Let P be a liberal schema containing a test statement t_k. We construct a liberal schema P_k from P by adding new assignment statements at each exit of t_k. It follows that t_k is necessary iff both new assignment statements are reachable. □

COROLLARY 23. *It is decidable whether a liberal schema is semifree.*

COROLLARY 24. *There is a procedure which, given a liberal schema, effectively constructs a strongly equivalent schema which is liberal and semifree.*

We note that both Corollaries 21 and 24 follow from Paterson's result, cited earlier, that $\mathscr{L}$ is effectively translatable into $\mathscr{L} \cap \mathscr{F}$. The results in this paper to which they are corollaries demonstrate that the translation procedure fulfills both the letter and the spirit of the definitions. This is in contrast to the result in Theorem 10.

Acknowledgment. I am grateful to Tom Ostrand and Ann Yasuhara for all their encouragement, helpful discussions, and suggestions. I am also very grateful to Emily Friedman for carefully reading the paper and making many good suggestions, particularly that I look for a construction such as the one which appears in Theorem 10.

REFERENCES

[1] A. K. CHANDRA, *On the properties and applications of program schemas*, Ph.D. thesis, Stanford University, Stanford, CA, 1973.

[2] R. L. CONSTABLE AND D. GRIES, *On classes of program schemata*, this Journal, 1 (1972), pp. 66–118.

[3] S. J. GARLAND AND D. C. LUCKHAM, *Program schemes, recursion schemes and formal languages*, J. Comput. System Sci., 7 (1973), pp. 119–160.

[4] S. A. GREIBACH, *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science, Vol. 36, Springer-Verlag, New York, 1975.

[5] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[6] D. C. LUCKHAM, D. M. PARK AND M. S. PATERSON, *On formalized computer programs*, J. Comput. System Sci., 4 (1970), pp. 220–249.

[7] M. S. PATERSON, *Equivalence problems in a model of computation*, Ph.D. thesis, Cambridge University, Cambridge, England, 1967.

[8] M. S. PATERSON AND C. E. HEWITT, *Comparative schematology*, Record Project MAC Conference on Concurrent Systems and Parallel Computation, 1970, pp. 119–128.

[9] E. J. WEYUKER, *program schemas with semantic restrictions*, Ph.D. thesis, Dept. Comp. Sci. Tech. Report DCS-TR-60, Rutgers University, New Brunswick, NJ, 1977.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0012 $01.00/0

AN EFFICIENT METHOD FOR STORING ANCESTOR INFORMATION IN TREES*

DAVID MAIER†

Abstract. We present a space efficient method for computing ancestor information in trees, specifically, whether one node is an ancestor of another and the lowest common ancestor of two nodes. We show the method is tunable to specific applications, and compare it to other methods. Finally, we apply our procedures to the problem of finding negative cycles in sparse graphs.

Key words. lowest common ancestor, tree algorithms, negative cycles

1. Introduction. We present a method of storing information in a tree that assists in quickly finding the lowest common ancestor of two nodes in the tree. Section 2 presents the general method, giving algorithms for the operations of inserting a node in the tree, grafting a subtree from one part of the tree to another, finding the ancestor of a node at a specified depth, and finding the lowest common ancestor of two nodes. The structure imposed upon the tree is a generalization of that used by Van Emde Boas, Kaas and Zijlstra [14]. We derive time and space bounds for these operations in terms of a parameter G. Section 3 shows how the complexity of the procedures can be tailored by letting $G = G(n)$, where n is the number of nodes in the tree, and gives a hypothetical example on which we perform the tailoring.

We compare our procedures to those of Aho, Hopcroft and Ullman [1] and see that the major improvement is in space complexity. On a tree of depth $D(n)$, the Aho, Hopcroft and Ullman method requires $O(n \log_2 D(n))$ space, while we can use as little as $O(n(\log_2 D(n)/\log_2 \log_2 D(n))^{1/2})$. Both of the LCA procedures run in $O(\log_2 D(n))$. Their LINK and our GRAFT procedures are seen not to be comparable, but in the special case of building a tree a node at a time, we take $O(n(\log_2 D(n)/\log_2 \log_2 D(n))^{1/2})$ time as compared to $O(n \log_2 D(n))$ for their method. Section 4 presents another example, finding negative cycles in a graph, that motivated the general case. We can find a negative cycle in a graph in space linear in the input, and time proportional to (number of nodes) · (number of edges) for not too sparse graphs. Finally, § 5 outlines some additional savings in time and space that can be made.

In what follows, we refer to nodes in a tree or in a graph by lower case letters. Names composed of upper case letters represent variables and functions used in the algorithms. We also assume the names of nodes are integers, so that we can easily store pointers to them. All time and space bounds are for the RAM model of Aho, Hopcroft and Ullman [2].

2. Storing information in the tree.

2.1. Given a tree T with root r, assume that each node u in the tree contains a pointer to its parent, except for the root. Call this pointer PARENT[u]. Pointers to children and siblings become necessary later for certain operations, but are not needed for all. Our method involves storing, at each node, a certain amount of additional information that can be computed quickly when the node is added to the tree and that

* Received by the editors September 23, 1977, and in revised form November 13, 1978.

† Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey. Currently at Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794. This work was partially supported by the National Science Foundation under Grant DCR-74-21939.

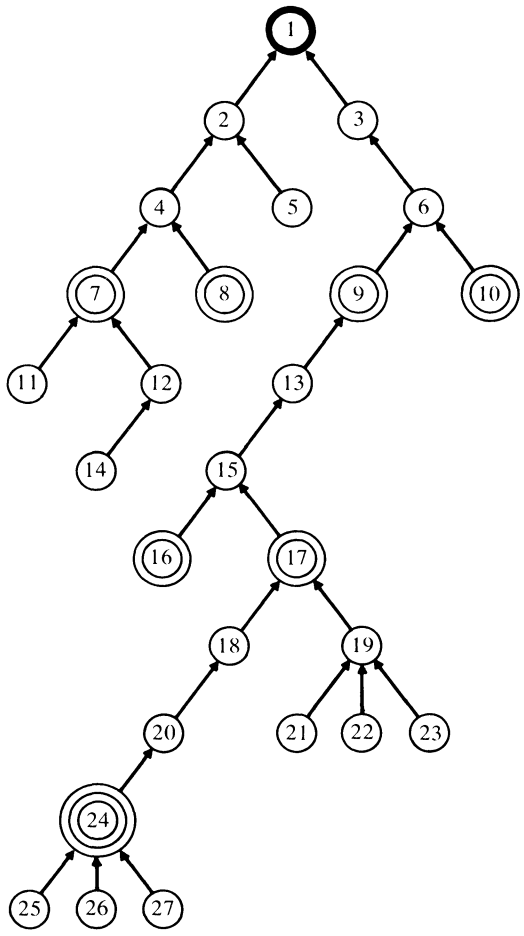


FIG. 1

enables us to make long jumps up the tree. For each node u in T, the *depth* of u is the number of ancestors of u in T, and is denoted DEPTH[u]. In Fig. 1, DEPTH[19] = 7, while the depth of the root, node 1, is 0.

Suppose we are given the maximum depth, say D, for any node in T. Pick an integer $2 < G \leqq D$ that we call the *base*. The method of selecting G is explored later.

DEFINITION. The *level* of a node u in T is the largest integer i such that G^i divides DEPTH[u]. The notation is LEVEL[u] = i. The root thus has infinite level, which is denoted LEVEL[r] = -1.

In Fig. 1, $G = 3$. All the nodes with a single circle are of level 0, those with two circles are of level 1, and the one node, 24, with three circles is of level 2. In the figures, the convention is that nodes of level i have $i+1$ circles, while the root has one heavy circle. In tree T, there is a maximum level among the nodes that are not the root. Call this value L. Note that $L = \lfloor \log_G D \rfloor$. In Fig. 1, $L = \lfloor \log_3 10 \rfloor = 2$.

DEFINITION. For any node u in T, the *chief* of u is the closest ancestor v of u such that the level of v is greater than the level of u. This relationship is denoted CHIEF[u] = v. The chief of the root is the root itself.

In Fig. 1, CHIEF[13] = 9, CHIEF[17] = 1 and CHIEF[26] = 24. Clearly, nodes of level 0 cannot be chiefs.

DEFINITION. Let node v be a chief, and let LEVEL[v] = i. The *k-clan* of v, for any $k < i$, is the set of all nodes u such that LEVEL[u] = k and CHIEF[u] = v. Also, v is the *k-chief* of its k-clan.

It is easily ascertained that every $(k+1)$-chief must also be a k-chief. In Fig. 1, the 0-clan of node 17 is {18, 19, 20, 21, 22, 23}; the 1-clan of node 1 is {7, 8, 9, 10, 16, 17} and the 2-clan of node 1 is {24}. Hence node 17 is a 0-chief, and node 1, the root, is a 1-chief and a 2-chief (and also a 0-chief). When we refer simply to the *clan* of a node u, where LEVEL[u] = k, we mean the k-clan of the chief of u. Referring again to Fig. 1, the clan of node 5 is the 0-clan of node 1 which is {2, 3, 4, 5, 6}. The chief of a clan is not considered a part of the clan.

We have defined three new pieces of information for each node u, namely DEPTH[u], LEVEL[u] and CHIEF[u]. We now present two arrays of information that are associated with the nodes of T. The first is the *clan-pointer vector*, denoted CLAN_PTR[u], that contains pointers to certain members of the clan of u. Specifically, CLAN_PTR[u][j] is the jth nearest ancestor of u in the clan of u, with the length of CLAN_PTR[u] being the largest i such that u has an ith nearest ancestor in its clan. Note that the root of T, among other nodes, has no clan-pointer vector. Figure 2 diagrams CLAN_PTR[5], with length 2, and CLAN_PTR[16], with length 1. The base $G = 4$. Node 11 has no clan-pointer vector. Note that the largest clan-pointer vector has $G - 2$ entries.

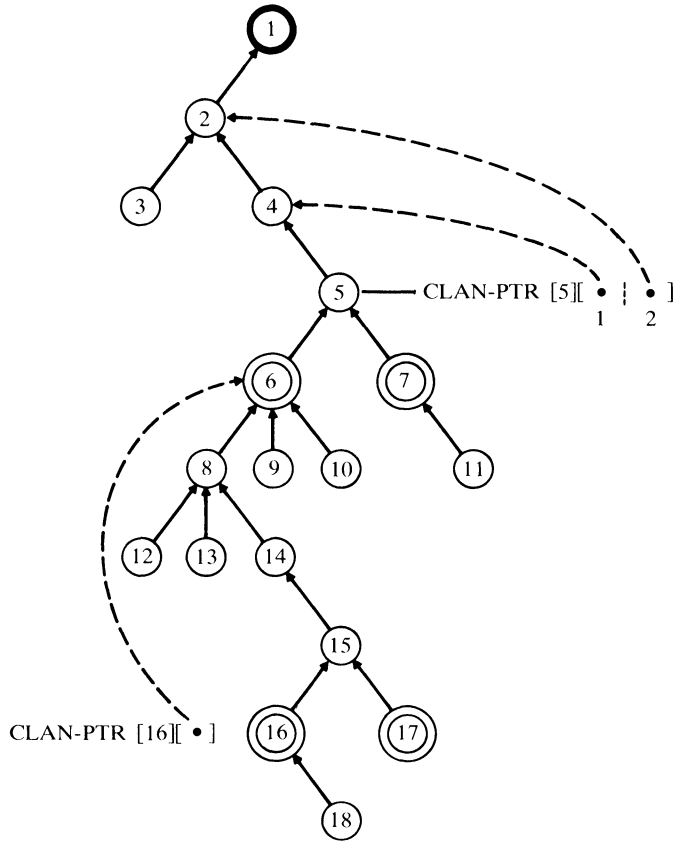


FIG. 2

DEFINITION. For a node u in T, a *superchief* of u, recursively, is

1. the chief of u, or
2. the superchief of a chief of u.

In Fig. 2, nodes 16 and 1 are superchiefs of node 18.

DEFINITION. Node v is the *k-superchief* of u if

1. v is a superchief of u,
2. LEVEL$[v] > k$, and
3. there is no closer superchief w of u with LEVEL$[w] > k$.

In Fig. 2, node 16 is the 0-superchief of node 18, and node 1 is the 1-superchief of node 18. What the k-superchief of u amounts to is the nearest ancestor of u of level $k+1$ or greater. Note that not all k-superchiefs are k-chiefs. In Fig. 1, node 24 is the 1-superchief of node 27, but node 24 is not the 1-chief of any node, since it has no descendents of level 1.

The purpose of clans and clan-pointer vectors is to allow large jumps up the tree T. When traveling up the graph, we attempt to do so within a clan of highest possible level. With the information stored in the tree so far, traveling from a 0-level node to its k-superchief takes $O(k)$ steps in the worst case. If we store information about superchiefs at each node, we can make the traversal in constant time. Unfortunately, this requires $O(L)$ additional storage at each node. The first step in getting around this problem is noting that all the nodes in a clan have the same superchiefs. Second, any node v that is a k-chief is also a 0-chief, and for all nodes u in the 0-clan of v, the k'-superchief of v, for $k' >$ LEVEL$[v]$, is the k'-superchief of u. The solution is to spread the superchief information among the members of each 0-clan, thereby saving space while having the information quickly accessible by every node of T. The exact method follows.

For each 0-level node u, define a vector SCHIEF$[u]$, where the length of SCHIEF$[u]$ plus the length of CLAN_PTR$[u]$ equals some fixed H, in order that there is the same amount of storage at each node. We determine the exact value of H shortly, but first let us work an example. Figure 3 shows a fragment of a tree T where the base G is 5 and the maximum level L is 14. We show a portion of the 0-clan of node 1: nodes 2, 3, 4 and 5. Among these four nodes we want to apportion the information about superchiefs of the members of this 0-clan. We need pointers to the 1-superchief through the 13-superchief. (The 0-superchief of a 0-level node is the same as the chief of the node, and the 14-superchief of any node is the root.) Hence 13 slots are needed among these four nodes. The lengths of CLAN_PTR for nodes 2, 3, 4 and 5 are 0, 1, 2 and 3. We need an H where

$$(H-0)+(H-1)+(H-2)+(H-3) \geqq 13,$$

which implies

$$4H-6 \geqq 13,$$

so $H \geqq 5$ suffices. Hence the lengths of SCHIEF for nodes 2, 3, 4 and 5 are 5, 4, 3 and 2. Figure 3 shows the way the information is apportioned.

In the general case, with base G and maximum level L, we choose the smallest H such that

$$L-1 \leqq \sum_{i=1}^{G-1} H-i+1 = (H+1)(G-1) - \sum_{i=1}^{G-1} i$$

$$= (H+1)(G-1) - (G-1)G/2 = (2H+2-G)(G-1)/2.$$

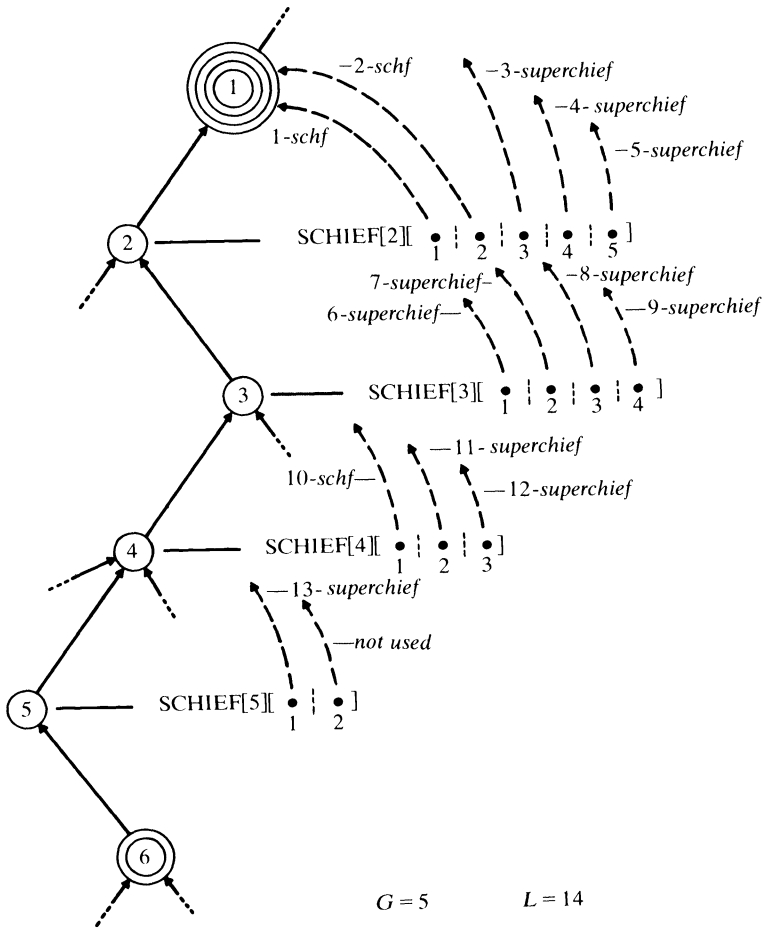


FIG. 3

Solving for H, we obtain

$$\frac{2(L-1)}{G-1} \leqq 2H+2-G \quad \text{or} \quad \frac{L-1}{G-1}+\frac{G-2}{2} \leqq H.$$

To determine the contents of SCHIEF[u][j], let $m = \text{DEPTH}[u] \bmod G$, and let

$$b = \sum_{i=1}^{m-1} H - i + 1 = (2H+2-m)(m-1)/2.$$

Then SCHIEF[u][j] points to the $(b+j)$-superchief of u.

At this point we present several functions to help us access the SCHIEF vectors of the 0-level nodes.

1. CONTENT(u, j) returns k such that SCHIEF[u][j] points to the k-superchief of u. This function can be computed as shown above.

2. VLENGTH(u) returns the length of SCHIEF[u]. VLENGTH(u) = $H+1-(\text{DEPTH}[u] \bmod G)$.

3. DISTANCE(k) and POSITION(k) are used to find where within a 0-clan a pointer to the k-superchief may be found. DISTANCE(k) gives the distance from a 0-chief to any node u in the corresponding 0-clan that contains a pointer to the

k-superchief in SCHIEF[u]. For the tree in Fig. 3, DISTANCE(11) = 3, since any node 3 nodes below a 0-chief contain a pointer to its 11-superchief.

POSITION(k) tells which entry in the appropriate SCHIEF vector points to the k-superchief. In Fig. 3, POSITION(11) = 2. DISTANCE and POSITION are almost partial right inverses to CONTENT: CONTENT(u, POSITION(k)) = k for any node u DISTANCE(k) below a 0-chief. It is possible to define explicit formulae for DISTANCE and POSITION, but Claim 1 shows they are unnecessary.

We also define one other useful function, MLEVEL(m), that returns the largest integer j such that G^j divides m.

CLAIM 1. CONTENT, VLENGTH, DISTANCE *and* POSITION *can be implemented to run in* $O(1)$ *time, given* $O(L)$ *precomputation time and* $O(L)$ *additional space.*

Proof. First note that the mod function is just the remainder after integer division. The claim is clear for CONTENT and VLENGTH. DISTANCE and POSITION have domains of only $L-1$ values, so they are best implemented as tables of length $L-1$. These tables can be computed in $O(L)$ time. □

CLAIM 2. MLEVEL *can be computed in* $O(\log_2 L)$ *time, with* $O(L)$ *precomputation and* $O(L)$ *additional space.*

Proof. Precompute the powers of G up to G^L in $O(L)$ time. Given these powers, we can find the desired value of MLEVEL by bisection search among them, looking for a j such that G^j divides m but G^{j-1} does not. Thus we get the $O(\log_2 L)$ bound once the precomputation is done, since we are searching over $O(L)$ values. □

We now use some of these functions to define another function, FIND_CHIEF(v, k), that finds the k-superchief of v if $k \geqq$ LEVEL[v], and otherwise finds the k-superchief of PARENT[v], which then must be of level 0.

Input. A node v and a nonnegative integer K. In this and all subsequent algorithms in this section, H, G, L, and the tree T are implicitly assumed to be parameters and all parameters except T are assumed to be passed by value.

Output. The next node above v of level $K+1$.

Explanation. In FIND_CHIEF, v is moved up the tree until it points to the desired node. Lines 2 and 3 move v to be the next node up the tree that is a chief. Note if LEVEL[v] $\neq 0$, then LEVEL[PARENT[v]] = 0. Line 4 checks if v points to the desired node. If not, in lines 5–9 an SCHIEF vector in the 0-clan above v is examined to find the proper superchief. In lines 5–7 the algorithm determines, via DISTANCE, which node above v contains the appropriate information in its SCHIEF vector, and v moves to this node. POSITION is used in line 8 to select the correct entry in the SCHIEF vector, and v is set to this new value, which points to the next k-superchief. Thus, at line 10, v points to the desired node and is returned.

```
1.  algorithm FIND_CHIEF(v, K)
2.  if LEVEL[v] = 0 then v ← CHIEF[v];
3.  else v ← CHIEF[PARENT[v]];
4.  if LEVEL[v] ≦ K and LEVEL[v] ≠ −1 then do;
5.      M ← DISTANCE(K);
6.      if M = G − 1 then v ← PARENT[v];
7.      else v ← CLAN_PTR[PARENT[v]][G − M − 1];
8.      v ← SCHIEF[v][POSITION(K)];
9.      end;
10. return (v);
11. end FIND_CHIEF;
```

This system of storing information on superchiefs still allows much duplication of information among members of a 0-clan, but it consumes less space than storing all the

pointers at each node. Section 5 gives ways to eliminate some duplication. As the system stands, if T has n nodes, the amount of additional space required is $O(nH)$.

2.2. Building the tree. Our next question is, if we are adding a new node u to tree T, how long does it take to compute DEPTH[u], LEVEL[u], CHIEF[u], CLAN_PTR[u], and SCHIEF[u]? Let us look at the following algorithm, ADD, that inserts a new node into the tree.

Input. A node v in tree T and a new node u.

Output. An updated tree T having u as a child of v and all necessary information stored at u.

Explanation. In lines 2–4 ADD fills in the parent, level and depth for u. In lines 5–7 ADD finds the chief of u when the level of u is 0. In line 8 it finds the chief of u in all other cases. (Line 8 will not work when the level of u is 0, since in the case LEVEL[u] = 0, FIND_CHIEF uses CHIEF[u].) In lines 9–10 the next ancestor of u of level greater than or equal to level u is found, and assigned to NEXT. In line 11 ADD uses NEXT to check if u has any ancestors in its clan; if so, the clan-pointer vector is filled in by lines 12–16. Finally, in lines 17–20, SCHIEF is computed if u is of level 0.

```
 1. algorithm ADD(u, v)
 2. PARENT[u] ← v;
 3. DEPTH[u] ← DEPTH[v] + 1;
 4. LEVEL[u] ← MLEVEL(DEPTH[u]);
 5. if LEVEL[u] = 0 then
 6.       if LEVEL[v] = 0 then CHIEF[u] ← CHIEF[v];
 7.       else CHIEF[u] ← v;
 8. else CHIEF[u] ← FIND_CHIEF(u, LEVEL[u]);
 9. if LEVEL[u] = 0 then NEXT ← v;
10. else NEXT ← FIND_CHIEF(u, LEVEL[u] − 1);
11. if CHIEF[u] ≠ NEXT then
12.       do I from 1 to H − VLENGTH(u) − 1;
13.            CLAN_PTR[u][I + 1] ← CLAN_PTR[NEXT][I];
14.            end;
15.       CLAN_PTR[u][1] ← NEXT;
16.       end;
17. if LEVEL[u] = 0 then
18.       do I from 1 to VLENGTH[u] while CONTENT(u, I) exists;
19.            SCHIEF[u][I] ← FIND_CHIEF(u, CONTENT(u, I));
20.            end;
21. end ADD;
```

When implementing this algorithm, CLAN_PTR and SCHIEF share a single vector of length H, since their lengths were chosen to sum to this amount.

THEOREM 1. *The algorithm* ADD *runs in* $O(H + \log_2 L)$ *time, given an initial precomputation of* $O(L)$ *time.*

Proof. Referring to Claim 1, the only parts of ADD that do not run in constant time are line 4 and the loops 12–14 and 18–20. Line 4 takes $O(\log_2 L)$ time, as shown by Claim 2. The worst case for the loops occurs when LEVEL[u] = 0 and both loops are executed. The loop 12–14 runs $H - \text{VLENGTH}(u) - 1$ times, while the loop 18–20 runs VLENGTH[u] times. Since both loops have a constant amount of time expended in their bodies, they run in $O(H)$ time. So the time for the entire algorithm is $O(H + \log_2 L)$. □

To build an entire tree from its nodes, the root of the tree must be initialized. We assume a procedure INIT(r) that initializes r as the root and runs in constant time. After calling this procedure, we just add the nodes one at a time until the tree is complete.

2.3. Grafting. A *graft* of a subtree of T headed by node u is simply changing the parent of u to any other node in T that is not u or a descendent of u, and then updating all the ancestor information in the subtree. In order to graft, we need pointers at each node to enable us to travel down the tree. Assume in this subsection that such pointers exist, and that there is a function PRE such that PRE(v) is the next node after v in a preorder traversal of T. Further assume the time PRE consumes in traversing an entire subtree of s nodes is $O(s)$ and that PRE uses a global variable, PEND, that causes PRE to return -1 instead if DEPTH[PRE(v)]$\leqq$PEND. Finally, assume ADD is modified to set the additional pointers. With these assumptions, the following algorithm grafts node u onto node v. Note that the parent and descendent information in the subtree below u is the same before and after GRAFT is applied.

Input. Nodes u and v in T, with u not v or a descendent of v.

Output. An updated tree, with u a child of v and all information in the subtree headed by u properly modified.

```
algorithm GRAFT(u, v) global PEND;
    ADD(u, v);
    PEND ← DEPTH[u];
    w ← PRE(u);
    do until w = −1;
        ADD(w, PARENT[w]);
        w ← PRE(w);
        end;
    end GRAFT;
```

If u heads a subtree of s nodes, then GRAFT(u, v) will perform s ADD operations. The time expended in calls to PRE is $O(s)$ total, and the $O(L)$ precomputation time for the ADD procedure becomes negligible as s grows. Hence we have

THEOREM 2. *The time to graft a subtree of size s is* $O(s(H+\log_2 L))$.

2.4. Finding an ancestor at a given depth. The next algorithm finds the ancestor of a node u at depth R. This result allows us to answer the question "Is v an ancestor of u?" by finding the ancestor of u of depth DEPTH[v] and comparing it to v.

Input. A node u in the tree and a nonnegative integer R.

Output. The ancestor of u at depth R.

Explanation. The variable u is used as a pointer that is moved up the tree until it hits depth R. Let u' denote the value of u when ANS is called. In the loop 3–5 we move up the tree by going from node to the chief of the node. At line 6, the ancestor of u' at depth R must be between u and the chief of u. This condition holds whenever we return to line 6. In lines 7–8 we attempt to jump along the clan of u, with the length of the jump being therefore a multiple of $G^{\text{LEVEL}[u]}$. In line 9 we see if this jump brings us to the desired depth. If not, lines 10–12 check if the desired node is among the next $G-1$ nodes. Otherwise, lines 13–21 find the node y above u of highest level such that DEPTH[y]$\geqq R$ and LEVEL[y]$<$LEVEL[u]. In step 20 we set u equal to this y. We now return to line 6, once again having the desired node between u and CHIEF[u], and also knowing that the level of u is less than the level during the previous pass through loop 6–22. Eventually u reaches the right depth, we drop out of the loop and return u.

```
1. algorithm ANS(u, R);
2. if R > DEPTH[u] then return ("error");
3. do while DEPTH[CHIEF[u]] > R and DEPTH[u] ≠ 0;
4.       u ← CHIEF[u];
5.       end;
6. do until DEPTH[u] = R;
7.       QUOT ← ⌊(DEPTH[u] − R)/G^LEVEL[u]⌋;
8.       if QUOT ≠ 0 then u ← CLAN_PTR[u][QUOT];
9.       if DEPTH[u] ≠ R then
10.          if DEPTH[u] − R < G then
11.              if DEPTH[u] − R = 1 then u ← PARENT[u];
12.              else u ← CLAN_PTR[PARENT[u]][DEPTH[u] − R − 1];
13.          else do;
14.              y ← u;
15.              LEV ← LEVEL[u];
16.              do until DEPTH[y] > R and LEV < LEVEL[u];
17.                  y ← FIND_CHIEF(u, LEV − 2);
18.                  LEV ← LEV − 1;
19.                  end;
20.              u ← y;
21.              end;
22.      end;
23. return (u);
24. end ANS;
```

THEOREM 3. *The algorithm* ANS *take* $O(L)$ *time.*

Proof. The precomputation for the functions used in FIND_CHIEF and for the powers of G in line 7 takes $O(L)$ time. The loop 3–5 iterates at most L times, since the level of u increases each time through. To figure the amount of time in loop 16–22, note that if on one pass we iterate loop 16–19 i times, the level of u is decreased by i when we return to line 6. So in all the iterations of loop 6–22, loop 16–19 iterates at most L times. Also, since the level of u decreases each time we return to line 6, the whole loop 6–22 can run at most L times. Thus the loop 6–22 takes $O(L)$ time. Summing the time for the different parts of ANS gives a total running time of $O(L)$. □

2.5. Finding the lowest common ancestor. Finally, we are ready to solve the lowest common ancestor (lca) problem. The lca of two nodes in tree T is the node of greatest depth which is an ancestor of the two nodes. We allow the lca to be one of the given nodes, if that node is an ancestor of the other, or the two nodes are the same. The predicate DONE(u, v) is true if $u = v$ or u and v have the same parent.

Input. Two nodes, u and v, in tree T.

Output. The lca of u and v.

Explanation. LCA uses u and v as pointers to work its way up the tree from their original locations to their lowest common ancestor. Let u' and v' be the original values. In lines 2–3 we bring u and v to the same depth, using ANS. In lines 4–7 u and v move up levels from node to chief, until u and v have the same chief. This situation is diagrammed in Fig. 4a, disregarding the solid arrows. The dashed arrows indicate strings of nodes. Portions of the tree immaterial to this discussion are not shown. The strategy is to have CHIEF[u] = CHIEF[v] every time line 8 is executed, while decreasing the level of u and v, all the time bracketing the lca of u' and v' between u and v and CHIEF[u]. Loop 8–37 insures this condition. The first part of the process is contained in

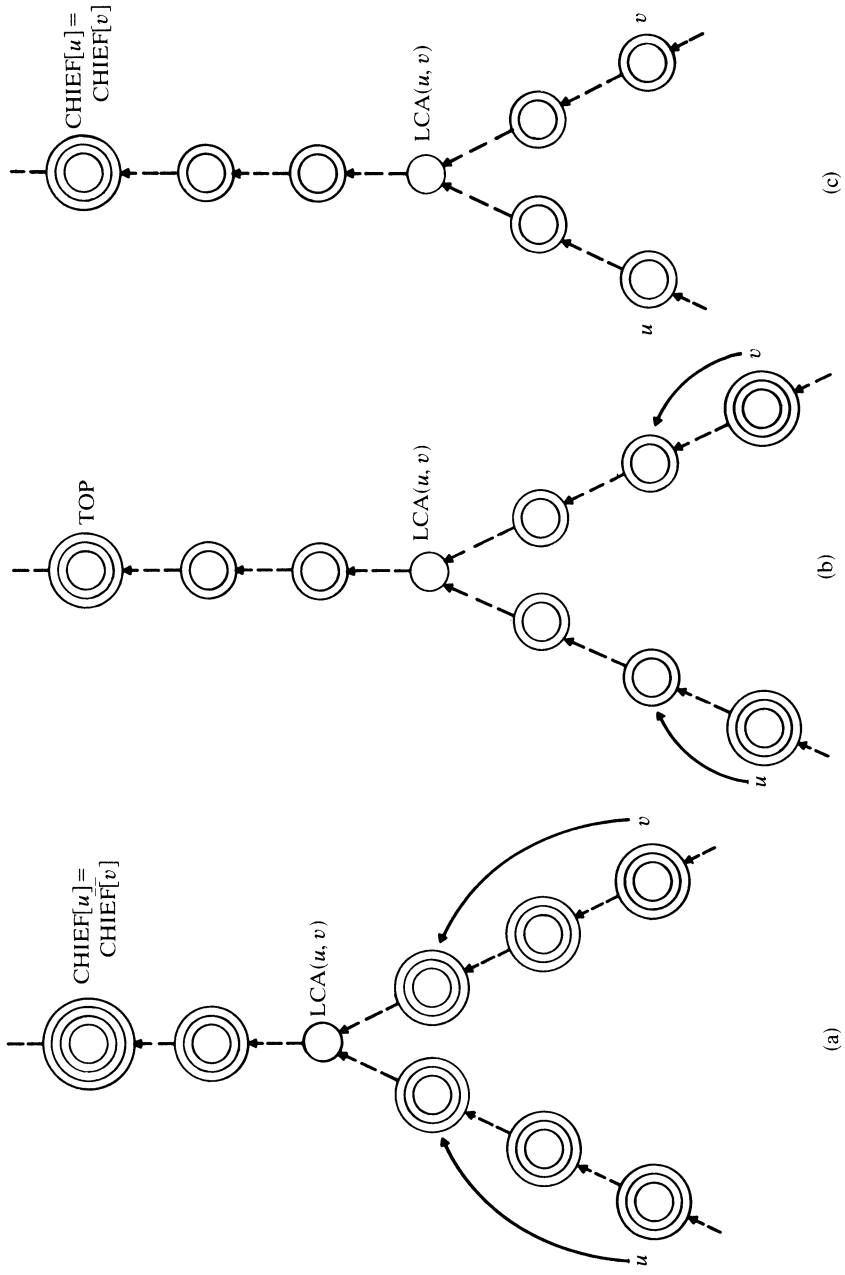

CHIEF[u] = CHIEF[v]
LCA(u, v)
u
v
(a)
TOP
LCA(u, v)
u
v
(b)
CHIEF[u] = CHIEF[v]
LCA(u, v)
u
v
(c)

FIG. 4

lines 9–21, which are a bisection search in the clan of u (= the clan of v) for the least depth to which u and v can be moved and still bracket the lca of u' and v'. TOP is the depth, *within the clan*, of a common ancestor of u and v. (TOP = 0 originally, indicating the chief of the clan, which is always a common ancestor.) M is the depth, within the clan, of u and v. The search continues until the next ancestor of u and v within the clan is the same. (M = TOP + 1.) The movement of u and v is given by the solid arrows in Fig. 4a. Figure 4b depicts the new situation, disregarding solid arrows. Note that when u and v are of level 0, the result of lines 9–21 is to give u and v the same parent.

Lines 22–36 return us to the state where CHIEF[u] = CHIEF[v], while decreasing the level of u and v. Lines 23–26 take care of the case where the lca of u and v lies within a 0-clan directly above x and v. In this case LCA moves u and v into the 0-clan. Otherwise, in lines 27–36 LCA searches for nodes above u and v of highest level less than LEVEL[u] that are distinct but have the same chief. In lines 32–33 we set u and v to these new values at level LEV − 1. The solid arrows in Fig. 4b show the movement of u and v. Figure 4c shows the situation after the move, which is analogous to the one in Fig. 4a with the level of u and v decreased. LCA now returns to line 8. Once we drop out of the loop 8–37, u and v are the same, or share the same parent. Lines 38–39 determine which is the case.

```
 1. algorithm LCA(u, v);
 2. if DEPTH[v] > DEPTH[u] then v ← ANS(v, DEPTH[u]);
 3. else u ← ANS(u, DEPTH[v]);
 4. do while not DONE(u, v) and CHIEF[u] ≠ CHIEF[v];
5, 6.    u ← CHIEF[u]; v ← CHIEF[v];
 7.       end;
 8. do until DONE(u, v);
 9.       TOP ← 0;
10.       M ← (DEPTH[u] mod G^LEVEL[u+1]) / G^LEVEL[u];
11.       MOVE ← M;
12.       do until M = TOP + 1;
13.            MOVE ← ⌈MOVE/2⌉;
14.            TRY ← TOP + MOVE;
15.            if CLAN_PTR[u][M − TRY] = CLAN_PTR[v][M − TRY] then
                    TOP ← TRY;
16.            else do;
17.                 u ← CLAN_PTR[u][M − TRY];
18.                 v ← CLAN_PTR[v][M − TRY];
19.                 M ← TRY;
20.                 end;
21.            end;
22.       if not DONE(u, v) then
23.            if CHIEF[PARENT[u]] = CHIEF[PARENT[v]] then do;
24.                 u ← PARENT[u];
25.                 v ← PARENT[v];
26.                 end;
27.            else do;
28.                 FLAG ← 0;
29.                 do LEV from LEVEL[u] to 2 by −1
                         while FLAG = 0;
30.                      if FIND_CHIEF(u, LEV − 2)
                              ≠ FIND_CHIEF(v, LEV − 2) then do;
```

```
31.                         FLAG ← 1;
32.                         u ← FIND_CHIEF(u, LEV − 2);
33.                         v ← FIND_CHIEF(v, LEV − 2);
34.                         end;
35.                     end;
36.                 end;
37.     end;
38. if u = v then return (u);
39. else return (PARENT[u]);
40. end LCA;
```

THEOREM 4. *The algorithm* LCA *takes* $O(L \log_2 G)$ *time.*

Proof. Lines 2–3 involve one call to ANS, which requires $O(L)$ time. In loop 4–7, the level of u and v is always increasing, so this loop iterates at most L times and hence takes $O(L)$ time. We know that each time we return to line 8 the level of u and v decreases. Hence the loop 8–37 runs at most L times. Within the loop, lines 9–11 execute once per iteration, while the subloop 12–21 is a bisection search within a single clan. Since a clan has depth at most G, this subloop requires time $O(\log_2 G)$ per iteration of the outer loop.

For the lines 22–36 we either execute lines 22–25 in constant time, or we execute the loop 29–36. For the loop 29–36 we have a situation similar to Theorem 3 where the inner loop 29–36 executes a total of at most L times during all the iterations of loop 8–37, since the level of u and v decreases for each iteration of the inner loop. Hence the major cost in loop 8–37 is the bisection search, so we can execute the loop in $O(L \log_2 G)$ time. This term dominates the run time for the rest of the algorithm, so the entire algorithm runs in time $O(L \log_2 G)$. □

3. Results and tuning.

3.1. Recall the relations between G, H and L: $L = \log_G D$ and $H = \lceil (L-1)/(G-1) + (G-2)/2 \rceil$. Let the number of nodes in tree T be n, and assume that the maximal depth of T is a function of n, $D(n)$. We shall express our previous results in terms of G, $D(n)$ and n. The space required for any of our features is $O(nH) + O(L)$, the $O(L)$ being for tables for computing DISTANCE and POSITION. This space requirement becomes

$$O(n[(\log_G D(n))/G + G] + \log_G D(n)) = O(n((\log_G D(n))/G + G)).$$

The time to add a node is

$$O(H + \log_2 L) + O(L) \text{ precomputation time,}$$

which is

$$O((\log_G D(n))/G + G) + O(\log_G D(n)).$$

The $\log_2 L$ term vanishes, since $O(\log_G D(n)/G + G)$ is at least $O((\log_G D(n))^{1/2})$. The time for grafting is just s, the size of the subtree, times the first term in the complexity of the add operation, or

$$O(s((\log_G D(n))/G + G)).$$

The time for finding an ancestor at a given level is

$$O(L) = O(\log_G D(n)).$$

Finally, the time for finding the lca of two nodes is

$$O(L \log_2 G) = O(\log_G D(n) \cdot \log_2 G) = O(\log_2 D(n)).$$

Let us examine a hypothetical case where we use some of these procedures, and see how to pick G. We are developing an algorithm that builds a binary tree T from n nodes, knowing $D(n) \leqq (n)^{1/2}$. During the process of building the tree we will make at most $n/2$ grafts of average size $\log_2 n$. While building the tree we will compute $n^2/2$ lca's. These values correspond to no specific real life situation, but are picked only to show how to choose G. Using the bounds just derived, the times we spend on each operation are as follows.

1. Adding:

$$O(n((\log_G n^{1/2})/G + G)) + O(\log_G n^{1/2}) = O(n((\log_G n)/G + G)).$$

2. Grafting:

$$O(n/2 \cdot \log_2 n \cdot ((\log_G n^{1/2})/G + G)) = O(n \log_2 n \cdot ((\log_G n)/G + G)).$$

3. Finding the lca's:

$$O(n^2/2 \log_2 n^{1/2}) = O(n^2 \log_2 n).$$

The add time is subsumed by the graft time and the time for the lca's does not depend on G. If we choose G such that the time for grafting is less than that for finding the lca's, the time complexity of the whole task will be $O(n^2 \log_2 n)$. So we want

$$O(n^2 \log_2 n) \geqq O(n \log_2 n \cdot ((\log_G n)/G + G)).$$

If we pick G as a function of n, specifically,

$$G(n) = (\log_2 n / \log_2 \log_2 n)^{1/2},$$

we satisfy the constraint on $G(n)$, our total time complexity is $O(n^2 \log_2 n)$ and our space complexity is $O(n(\log_2 n/\log_2 \log_2 n)^{1/2})$. There are other values for $G(n)$ that work for this hypothetical example. The choice $G(n) = (\log_2 n/\log_2 \log_2 n)^{1/2}$ minimizes the space requirements as well. Note that the complexity $O((\log_G n)/G + G)$ is minimized when $O((\log_G n)/G) = O(G)$ or $O(\log_G n) = O(G^2)$, which means

$$O(\log_2 n) = (G^2 \log_2 G).$$

Substituting our choice of $G(n)$ in the right side gives

$$O(\log_2 n - ((\log_2 \log_2 \log_2 n)/\log_2 \log_2 n)) = O(\log_2 n).$$

3.2. Comparison to other methods. The data structure described herein is quite similar to that of Van Emde Boas, Kaas and Zijlstra [14]. Our concept of level corresponds to their concept of rank, except they use an explicit *ranking function* to assign ranks to nodes of various depths. This ranking function differs for trees of different heights, but resembles our level function when $G = 2$. For a given base G, our computation of the level of a node is uniform for trees of varying heights. Van Emde Boas, Kaas and Zijlstra must impose additional restrictions on their ranking function. They are using the function to decompose the tree for a divide and conquer strategy and the subtrees resulting from the decomposition must be close to the same height.

We now compare our operations to the LINK and LCA operations of Aho, Hopcroft and Ullman (AHU) [1]. The first place we see an improvement is in space complexity. For a tree with n nodes and ultimate depth $O(n)$, the AHU method requires $O(n \log_2 n)$ additional space, while ours can use as little as

$O(n(\log_2 n/\log_2 \log_2 n)^{1/2})$. (As we saw before, $G(n) = (\log_2 n/\log_2 \log_2 n)^{1/2}$ minimizes the space complexity.) In general, if the depth of the tree is known in advance to be $D(n)$, the AHU method requires $O(n \log_2 D(n))$ space, while ours can run in $O(n(\log_2 D(n)/\log_2 \log_2 D(n))^{1/2})$ space.

Comparing the two LCA procedures, for $f(n)$ lca's in a tree of depth $D(n)$, both procedures will need $O(f(n) \log_2 D(n))$ time. The AHU LINK procedure and our GRAFT procedure are not comparable in terms of time. The $O(n \log_2 n)$ time bound on LINK depends heavily on the restriction that once a node is given a parent, the parent never changes. The GRAFT procedure, on the other hand, can be used to change a node's parent after it has been added to the tree. In the special case where we build a tree one node at a time, ADD operations suffice. We have a total time of $O(n(\log_2 n/\log_2 \log_2 n)^{1/2})$, if we choose $G(n)$ as before, as compared to $O(n \log_2 n)$ for the AHU method.

Last, we include a procedure, ANS, for finding if one node is the ancestor of another, that the AHU method does not have.

4. Finding a negative cycle in a graph. We turn our attention to the problem of finding negative cycles in a sparse graph. This problem has application in the area of two dimensional package placement and was originally brought to our attention by Eric Cho. We are given a digraph (V, E) where for each edge e in E we are given a cost $c(e)$ that ranges over the real numbers. There is also a source node s in V, from which all other nodes in V are reachable. The problem is to find if graph G has a negative cycle: a path from a node to itself where the sum of the costs of the edges along the path is negative. If such a cycle exists, we also want to find one such cycle. We can not find all negative cycles, since if one exists, there are an infinite number. We further wish to constrain our space complexity to be linear in the size of the input, which we assume to be $O(|V|+|E|)$, where $|E|$ grows much more slowly than $|V|^2$.

In the literature, most methods of detecting negative cycles in a graph are modifications of the single-source shortest path problem [4], [6], [7], [8], [10], [15], [16]. These are iterative methods in which the number of iterations until convergence can be bounded in the absence of a negative cycle, but the algorithm runs forever in the presence of such a cycle. The procedures are modified to count steps and, after the bound is passed, report the presence of a negative cycle. For recent results and bibliographies, the reader is directed to Johnson [8], Pape [12], and Pierce [13]. Several methods [3], [5], [9] look for negative cycles directly, without computing information about shortest paths. However, no complexity analysis is given for any. Yen [16] shows a worst case for the method of Florian and Roberts [5] of worse than $O(|V|^3)$, and the method of Klein and Tibrewala [9] would seem to be $O(|V|^3)$.

Of the methods above that compute shortest paths, most exhibit two traits. First, they detect the presence of negative cycles, but do not find one. Second, they achieve their worst case behavior in the presence of negative cycles. The method we present, though another modification of the shortest paths problem, does not share these shortcomings. We are essentially using Yen's implementation of the Ford–Fulkerson algorithm [6], [16]. We do not always generate shortest paths, but at each stage i, we look for paths of length i from the source s to the other nodes, such that these paths are less costly than paths discovered so far of length $i-1$ or less. This strategy corresponds generally to a breadth-first search through the graph (V, E), looking for paths with lesser cost to the nodes visited. We keep track of the search in a tree T_g, in which source node s is the root and each path from node to root corresponds to the least costly path found so far in the graph from the node to s. Keeping T_g up to date requires add and

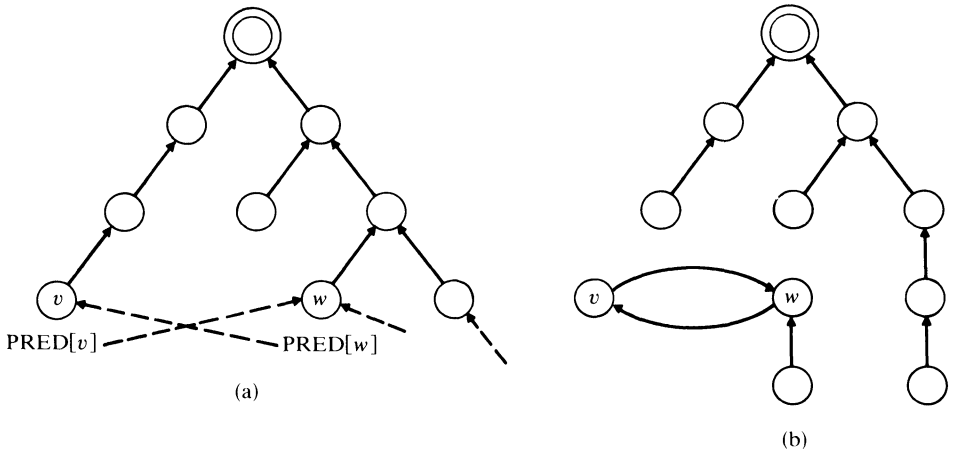


FIG. 5

graft operations as nodes are first reached and less costly paths are found. We are looking for the case where we can generate a less costly path from s to some node u by passing through u earlier on the path. This betrays the existence of a negative cycle.

In the algorithm CYCLE below, the vector DIST holds the current lowest costs for paths from s to the nodes of V; the vector PRED is used to temporarily hold predecessor information about tree T_g while we check for cycles; and the sets U and U' hold "active" nodes for the current and next iteration. We further assume the nodes and edges of the graph are presented in a way that we can form a list of adjacent nodes for all nodes in $O(|E|)$ time. The double arrow ($\Leftarrow$) is used for set or list assignment; lambda (Λ) is used for the null value.

Input. A graph (V, E) and a cost function c on the edges of the graph.

Output. The message "no negative cycles" or a list of nodes along a negative cycle in the graph.

Explanation. Steps 1–4 are straightforward. Step 1 initializes the lists of adjacent nodes for each node in V. Step 2 initializes s as the root of the tree T_g, as well as initializing the set U and the vector DIST. In step 4, where the iteration actually begins, we clear the vector PRED and the set U' which holds the nodes we consider during the next iteration. Step 5 is the meat of the distance finding procedure. We check, for each node u that received a lesser distance at the last stage, if we can establish a less costly path to any of the nodes adjacent to u using the path from s to u. We make one restriction: We do not consider a node u in U if u has already been added to U' during the current stage. The reason for this restriction is that we want the active nodes of the same depth at each stage. If all the nodes in U are currently at depth $i-1$ and we add a node w to U' where PRED[w] is already in U', then when we restructure T_g in step 7, node w would end up at depth $i+1$ rather than i. Another bad effect avoided by this restriction is depicted in Fig. 5, where two nodes end up in U' with their predecessors being each other. After step 7, these nodes would end up disconnected from T_g. The reason we can ignore paths through a node u already added to U' is that node u turns up in the set U during the next iteration and we consider it then. All that concerns us is that if there are less costly paths in the graph that we have not yet found, there is at least one node in U' at step 6, and tree T_g continues to grow in depth.

Step 7 would seem to be incorrect. Here, after checking to see if we are trying to make a node a descendent of itself—implying a negative cycle (see Figure 6)—it seems

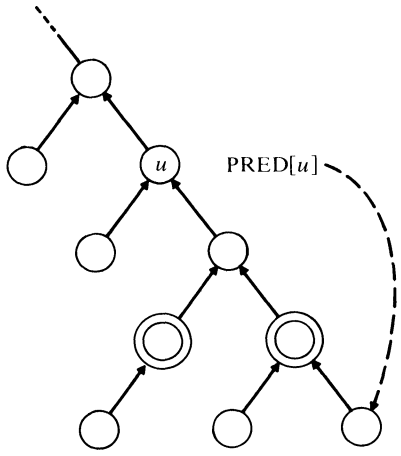


FIG. 6

we should graft node u to PRED[u], as Figure 7 shows. We perform only an add instead. We do not need to update the information in the subtree headed by u, since each of these nodes eventually become members of U' themselves, as the improved distance to u propagates down to them in subsequent stages, or they are added to other nodes in the tree. The out-of-date information they contain is not accessed before they become active. We only check the ancestor relation (using ANS) between a newly active node and a node above it in the tree T_g. Before any node in the subtree in question can have an active descendent, it must itself become active again. Hence we are able to get by with an ADD operation rather than the GRAFT alluded to before.

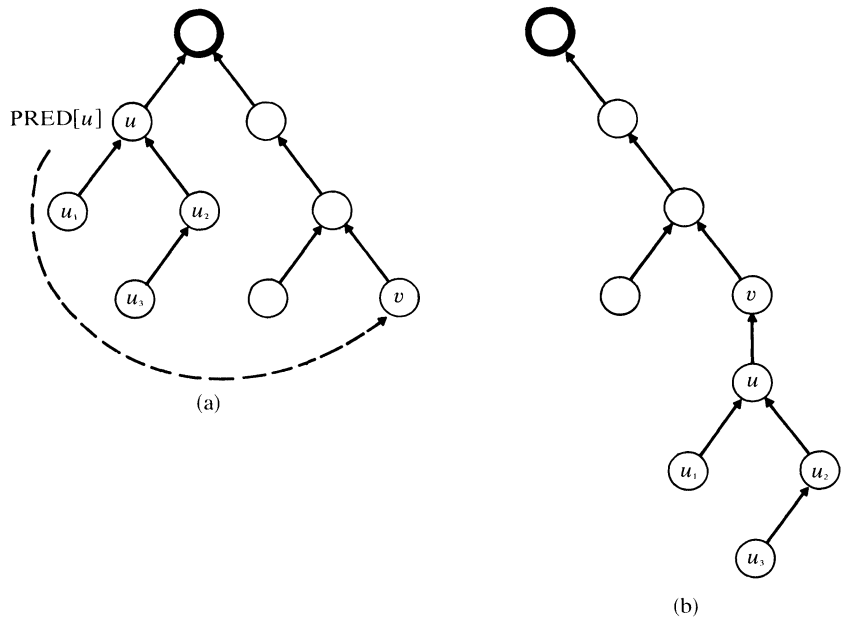


FIG. 7

algorithm CYCLE

1. **for** all v in V let LIST[v] be a list of all nodes u such that (v, u) is in E.
2. $U \Leftarrow \{s\}$; **call** INIT(s); DIST[s] $\leftarrow 0$;
   for all v in V, $v \neq s$, **do** DIST[v] $\leftarrow \Lambda$;
3. **do** steps 4–7 **while** "true";
4. PRED $\Leftarrow \Lambda$; $U' \Leftarrow \varnothing$;
5. **for** each u in U **do**;
   if u has not been added to U'
   then for each v in LIST[u] **do**;
   if DIST[v] < DIST[u] + $c((u, v))$ **then do**;
   PRED[v] $\leftarrow u$;
   DIST[v] $\leftarrow$ DIST[u] + $c((u, v))$;
   $U' \leftarrow U'U\{u\}$;
   end;
   end;
   end;
6. **if** $U' = \varnothing$ **then return** ("no negative cycles");
7. **for** all u in U' **do**;
   if ANS(PRED[u], DEPTH[u]) = u **then**
   construct NEG_CYCLE, a list of the nodes along the path in T_g from PRED[u] to u;
   return (NEG_CYCLE);
   else ADD(u, PRED[u]);
   end;
   $U \leftarrow U'$;

Complexity. We assumed earlier that the input allows step 1 to be executed in $O(|E|)$ time. Step 2 takes $O(|V|)$ time. Step 4–7 can be repeated at most $|V|$ times, since T_g gains depth during each iteration. Step 4 is $O(|V|)$, and step 5 is $O(|E|)$, since we can perform the body of the loop no more times than the total number of members of LIST[u] for all u in U, which is bounded by $|E|$. Step 7 does at most one ADD and ANS for each node in U', and hence it takes

$$O(|V|) \cdot (O(\log_G |V|) + O((\log_G |V|)/G + G)) = O(|V| \cdot (\log_G |V| + G))$$

which is minimized when

$$O(\log_G |V|) = O(G) \quad \text{or} \quad O(\log_2 |V|) = O(G \log_2 |V|).$$

This relationship is satisfied if

$$G = (\log_2 |V|)/\log_2 \log_2 |V|.$$

Since steps 5 and 7 are our major time consumers, the total time complexity is

$$O(|V| \cdot (|E| + |V|(\log_2 |V|)/\log_2 \log_2 |V|)).$$

Our space complexity is $O(|V| + |E|)$ to represent the graph, plus $|V| \cdot O((\log_G |V|)/G + G)$ for our tree T_G. Hence we have

THEOREM 5. *The algorithm CYCLE requires*

$$O(|V| \cdot (|E| + |V|(\log_2 |V|)/\log_2 \log_2 |V|))$$

time and

$$O(|E| + |V|(\log_2 |V|)/\log_2 \log_2 |V|)$$

space on the graph (V, E).

COROLLARY. *If, for a class of graphs,* $O(|E|) > O(|V|(\log_2 |V|)/\log_2 \log_2 |V|)$, *then the algorithm CYCLE runs in space linear in its input, that is,* $O(|V|+|E|)$.

5. Other economies and conclusions.

5.1. Perhaps the most obvious economy is to modify $G(n)$ to a function $G'(n)$ where the value of $G'(n)$ is the closest power of two to $G(n)$. Now all statements involving powers of $G(n)$ can utilize shift operations for exponentiation and division, and there is no need for a table of powers of $G(n)$.

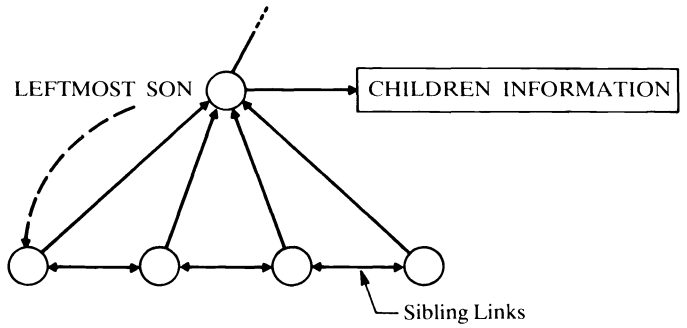


FIG. 8

If the tree includes child pointers, we can make both a time and a space savings by noting all the children of a node share the same information, except for their names. Thus, instead of keeping information for children of a node at each child, we store one copy of the information at the node itself, for all the children to access. Figure 8 shows

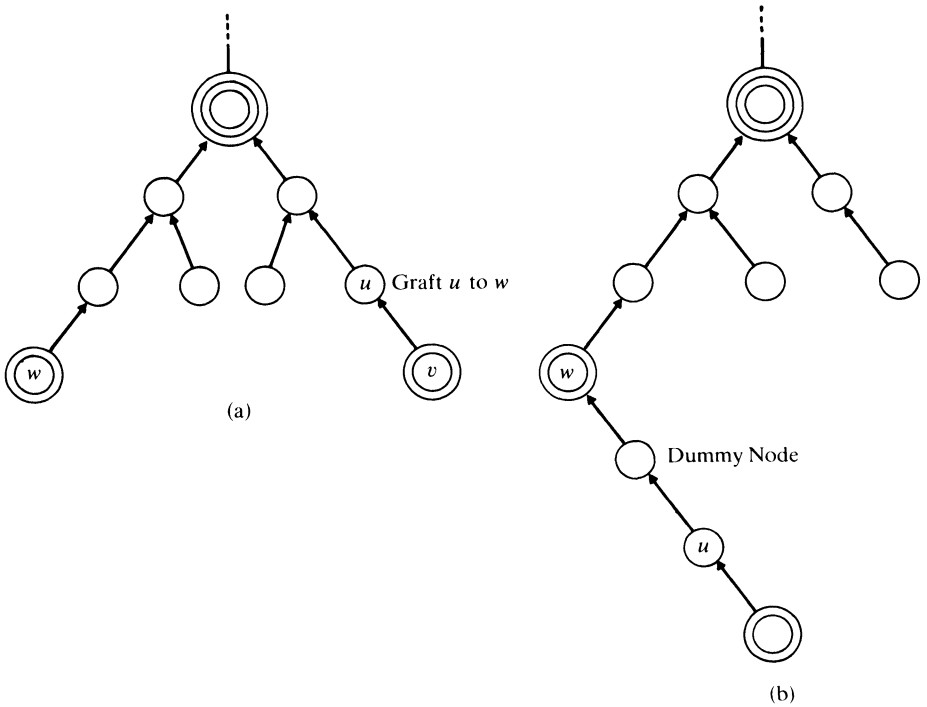


FIG. 9

one possible arrangement. This modification gives us a time and space savings, since we need assign space only to interior nodes and need compute the information only for the first child of the node. Thus, the time complexity for the GRAFT operation, for example, becomes

$$O(s + I_s \cdot ((\log_G D(n))/G + G))$$

where s is the number of nodes in the subtree being grafted, and I_s is the number of leftmost sons, which equals the number of interior nodes. This rearrangement of storage results in substantial savings when the ratio of leaves to interior nodes is large.

The reader may wonder why the same amount of space is allocated for each node in the tree when some nodes do not need it all and why much of this information is duplicated. The storage is apportioned in this way to accommodate grafting nodes, where the level—and hence the space requirements—for a node can change. If we know in advance that we will make no grafts, we can assign each node only the amount of space needed. For example, non-0-level nodes require no space for SCHIEF pointers. Or we could keep all the superchief information at the chief of each 0-clan, rather than apportioning it among the nodes of the clan. It may also be possible, even in the presence of some grafting, to keep each 0-level node at level 0 after a graft, by inserting dummy nodes to force alignment. (See Fig. 9.)

5.2. Conclusions. We have given a method that, by addition of some information to a tree, allows us to add nodes, graft nodes, and find if one node is an ancestor of another and the lowest common ancestor of two nodes within the tree. The time and space complexities are shown to be tunable to different applications. Our results are seen to compare favorably to the on-line results of Aho, Hopcroft and Ullman method, especially in regards to space complexity. The procedures are applied to finding negative cycles in graphs and yield a linear space algorithm when the graph is not too sparse. We have also shown a number of ways to make savings in time and space, in special cases.

Acknowledgment. I thank Jeff Ullman for reading and discussing an earlier version of this paper [11]; Eric Cho, Allen Korenjak and Henry Baird of RCA Laboratories for suggesting and discussing the negative cycle problem; and the referees for their suggestions on improving my presentation.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT AND J. D. ULLMAN, *On finding lowest common ancestors in trees*, this Journal, 5 (1976), 115–132.

[2] ———, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] W. DOMSCHKE, *Two algorithms to detect negative cycles in a valued graph*, Computing, 11 (1973), pp. 124–136. (In German.)

[4] S. E. DREYFUS, *An appraisal of some shortest-path algorithms*, OR, 17 (1969), pp. 395–411.

[5] M. FLORIAN AND P. ROBERTS, *A direct search method to locate negative cycles*, Management Sc., 17 (1971), pp. 307–310.

[6] L. R. FORD, JR. AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[7] T. C. HU, *A decomposition algorithm for shortest paths in a network*, Operations Res., 16 (1968), pp. 91–102.

[8] D. B. JOHNSON, *Efficient algorithms for shortest paths in a network*, J. Assoc. Comput. Mach., 24 (1977), pp. 1–13.

[9] M. KLEIN AND R. K. TIBREWALA, *Finding negative cycles*, INFOR—Canada J. Operational Res. and Information Processing, 11 (1973), pp. 59–65.

[10] S. R. KOSARAJU, private communication to J. D. Ullman, 1977.

[11] D. MAIER, *A space efficient method for the lowest common ancestor problem and an application to finding negative cycles*, TR #230, Computer Science Laboratory, Princeton University, Princeton, NJ, June 1977. Also in Proceedings of the 18th Annual Symposium on Foundations of Computer Science, October 1978, pp. 132–141.

[12] U. PAPE, *Eine Bibliographie zu "Kurzeste Weglangen und Wege in Graphen,"* Bericht 77-07, Technischen Universitat Berlin, Berlin, May, 1977.

[13] A. R. PIERCE, *Bibliography on algorithms for shortest path, shortest spanning tree, and related circuit routing problems* (1965–1974), Networks, 5 (1975), no. 2, pp. 129–149.

[14] P. VAN EMDE BOAS, R. KAAS AND E. ZIJLSTRA, *Design and implementation of an efficient priority queue*, Math. System Theory, 10 (1977), pp. 99–127.

[15] J. Y. YEN, *An algorithm for finding shortest routes from all source nodes to a given destination in general networks*, Quart. Appl. Math., 27 (1970), pp. 526–530.

[16] ———, *On the efficiency of a direct search method to locate negative cycles in a network*, Management Sci., 19, no. 3 (1972), pp. 333–335.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0013 $01.00/0

NODE-DELETION NP-COMPLETE PROBLEMS*

M. S. KRISHNAMOORTHY† AND NARSINGH DEO‡

Abstract. The entire class of node-deletion problems can be stated as follows: Given a graph G, find the minimum number of nodes to be deleted so that the remaining subgraph g satisfies a specified property π. For each π, a distinct node-deletion problem arises. The various deletion problems considered here are for the following properties: each component of g is (i) null, (ii) complete, (iii) a tree, (iv) nonseparable, (v) planar, (vi) acyclic, (vii) bipartite, (viii) transitive, (ix) Hamiltonian, (x) outerplanar, (xi) degree-constrained, (xii) line invertible, (xiii) without cycles of a specified length, (xiv) with a singleton K-basis, (xv) transitively orientable, (xvi) chordal, and (xvii) interval. In this paper, these 17 different node-deletion problems are shown to be NP-complete. A unified approach is taken for the transformations employed in the proofs.

Key words. Graph, node cover, node-deletion, NP-complete

1. Introduction. Only loopless, finite graphs without multiple edges will be considered here. We will denote a graph by $G = (V, E)$, where V is the set of nodes, and E, the set of edges of graph G. The more-or-less standard graph terminology and definitions used here can be found in most textbooks on graph theory, e.g., [4], [6]. By *deleting a node* v from a graph G, we mean that the node v is removed from G and all edges incident upon node v are also removed. A graph is said to be *null* if it has no edges. An *articulation point* in a connected graph is a node whose removal makes the given graph disconnected. A *nonseparable* graph is connected, has more than one node and has no articulation points. A graph G is said to be *Hamiltonian*, if there is a circuit in G, which passes through all nodes once and only once. A diagraph is said to be *transitive*, if for every pair of edges $\langle v_i, v_j \rangle$, and $\langle v_j, v_k \rangle$, there is an edge $\langle v_i, v_k \rangle$. An *outerplanar* graph is a planar graph with a planar mapping in which every node lies on the infinite region. A *line graph* (or an edge graph) L of a given graph G is a graph in which each node corresponds to a distinct edge of G, and two nodes in L are adjacent iff the corresponding edges in G are incident at a common node. Graph L is called *line invertible*, and G is referred to as an inverse line graph of L. An undirected graph is said to be *transitively* orientable, if by assigning appropriate directions to the edges, the resultant diagraph becomes transitive. A *chordal* graph is one in which for every circuit of length greater than 3, there is an edge (called a chord) joining two nonconsecutive nodes of the circuit. A graph G is called an *interval* graph if every circuit of length 4 in G has a diagonal and its complement $\bar{G}$ is transitively orientable.

Since Cook's discovery of the NP-complete class of problems [3], the list of problems in this class has been growing steadily. Karp [7], [8], Sahni [9], Garey, Johnson and Stockmeyer [5], Ullman [11], and other have have shown that a large number of combinatorial problems are NP-complete. Three of the NP-complete graph problems, namely, minimum feedback node-set, minimum node-cover, and maximum clique can be viewed as particular cases of a more general *node-deletion problem* stated as follows: Given an undirected or directed graph G, find a minimum number of nodes to be deleted from G, so that the remaining subgraph or subdigraph of G satisfies a specified property. Thus, the minimum node-cover problem is the same as the node-deletion problem, the specified property being that the remaining subgraph is a null graph.

* Received by the editors January 5, 1977, and in final revised form August 28, 1978. This work was partially supported by the National Science Foundation under Grant No. MCS-7825851.

† Computer Science programme, Indian Institute of Technology, Kanpur, India.

‡ Department of Computer Science, Washington State University, Pullman, Washington, 99163.

Our results depend on the property π being such that if G_1 and G_2 satisfy π, then G_1+G_2 also does. In other words we say that a property π is determined by the components of a graph, if whenever the components of a graph G satisfy π, then G also satisfies π.

For some node-deletion problems polynomial-time algorithms have been discovered. In this paper, we shall show that 17 node-deletion problems are NP-complete, by first transforming an NP-complete problem into the node-deletion problem and then proving that the latter is in NP. It can be verified that the transformations can be performed in polynomial time. We also examine these NP-complete problems, when the input graphs are restricted to be of a particular class.

2. Node-deletion problems. The following 17 node-deletion problems are stated as recognition problems. Each problem is specified by giving (under the heading "Input") as a generic element of its domain of definition and (under the heading "Property") the property that causes an input to be accepted.

(i) Node-deleted "null" subgraph (node-cover). *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is a null graph.

(ii) Node-deleted "complete" subgraph (maximum clique). *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, the remaining subgraph is the union of complete components.

(iii) Node-deleted "tree" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is a tree.

(iv) Node-deleted "nonseparable" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each of the connected components of the remaining subgraph is nonseparable.

(v) Node-deleted "planar" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is planar.

(vi) Node-deleted "acyclic" subdigraph (feedback node-set). *Input.* Directed graph $D=(V, E)$ and a positive integer k.

Property. When we delete k nodes from D, each component of the remaining subdigraph is acyclic.

(vii) Node-deleted "bipartite" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is bipartite.

(viii) Node-deleted "transitive" subdigraph. *Input.* Directed graph $D=(V, E)$ and a positive integer k.

Property. When we delete k nodes from D, the remaining subdigraph is the union of transitive digraphs.

(ix) Node-deleted "Hamiltonian" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, the remaining subgraph is the union of Hamiltonian graphs.

(x) Node-deleted "outerplanar" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is outerplanar.

(xi) Node-deleted "degree-constrained" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph contains no node with degree greater than any specified positive integer k.

(xii) Node-deleted "line-invertible" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is line-invertible.

(xiii) Node-deleted "without cycles of specified length" subdigraph. *Input.* Directed graph $D=(V, E)$ and a positive interger k.

Property. When we delete k nodes from G, each component of the remaining subdigraph contain no cycles of length (any specified positive integer).

(xii) Node-deleted "a singleton K-basis"[1] subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each of the connected components of the remaining subgraph has a singleton K-basis.

(xv) Node-deleted "transitively orientable" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G each component of the remaining subgraph is transitively orientable.

(xvi) Node-deleted "chordal" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is a chordal graph.

(xvii) Node-deleted "interval" subgraph. *Input.* Undirected graph $G=(V, E)$ and a positive integer k.

Property. When we delete k nodes from G, each component of the remaining subgraph is an interval graph.

Each such property denoted by π, (except "Hamiltonian," and "nonseparable") holds even when G is disconnected [6]. Thus, if each of the connected components of G satisfies property π, then the given graph G satisfies property π. For the node-deleted "nonseparable" subgraph property, we specify that each of the connected components be nonseparable. Otherwise it is fairly easy to find one; namely it is equal to the difference between the number of nodes and the number of nodes the largest biconnected component has.

3. Transformation steps. The entire class of the node-deletion problems stated in §2 is in NP. We prove here that the node cover problem is polynomially transformable to the node-deletion problems.

We use two general techniques for the transformation that applies to a variety of deletion problems. In the first of two lemmas, we substitute a graph, which satisfies some specified conditions, for every edge of G in the node cover problem. In the second lemma, we substitute a graph, which satisfies some specified conditions, for every node of G in the node cover problem. In the subsequent corollaries we draw graphs for each property, which when substituted provides the necessary transformation.

[1] The problem of finding a minimum K-basis [10] of graph G is that of selecting as small a set B of nodes as possible, such that every node of G is at a distance K or less from some node in B.

LEMMA 1 (edge substitution lemma). *Let π be a specified graph property that is determined by its components, and suppose there is a graph H with two nodes s and t such that the following hold*:

(1) *The graph H is a forbidden graph for property π.*

(2) *If we take one, two or three H's, delete either s or t from each, and join the graphs at the remaining s and t, then the resulting graph has property π.*

(3) *If we delete both s and t from H, then the resulting subgraph has property π.*

Then the node-cover problem in graphs of degree at most three is polynomially transformable to the node-deletion problem for property π.

Proof. Let G be a given graph of degree at most three in which we wish to find the node-cover. Substitute each edge (u, v) in the given graph by the whole graph H with the nodes s and t coinciding with u and v respectively.

Let the minimum number of nodes for the node-cover problem be k_1. That is, by deleting these k_1 nodes from the given graph G, we get a null subgraph. When we delete these k_1 nodes from the constructed graph, the resultant subgraph will satisfy property π by condition (1) and (2) as the set of k_1 nodes is either s or t from each of the H's.

Conversely, let the minimum number of nodes for the node-deletion problem for property π be k_2. These k_2 nodes are either s or t or other nodes from each of the graph H. When these nodes are other than s or t in a particular H, then the corresponding s and t cannot be included in the minimum set of nodes, for then the set of nodes chosen is not minimal. So whenever a node other than s or t from H is chosen, one might as well have chosen the nodes s or t from each of the graphs H, retaining the minimality. That is, when we delete these k_2 nodes, the resulting subgraph satisfies property π. When these k_2 nodes are deleted from the given graph G, we will get a null subgraph; otherwise at least one edge is not covered by any node thus contradicting the assumption; hence the lemma follows.

LEMMA 2 (node substitution lemma). *Let π be a specified graph property that is determined by its components, and suppose there is a graph F with a node "s" such that the following hold*:

(1) *The graph F and the subgraph resulting after deleting node "s" from F satisfy π.*

(2) *If a node x is added to the graph F and nodes x and "s" are joined by an edge, then the resulting graph is a forbidden induced subgraph for property π.*

Then the node-cover problem is polynomially transformable to the node deletion problem for property π.

Proof. Let G be a given graph in which we wish to find the node-cover. Replace each node u of G by the graph F, with u coinciding on the node "s". It can be seen (as in the proof of Lemma 1) that a node cover with k nodes exists in G iff k nodes can be deleted from G, so that the remaining subgraph satisfies the property π.

COROLLARY 1. *The node-cover problem in graphs of degree at most three is polynomially transformable to the node-deleted* (a) *tree*, (b) *planar*, (c) *acyclic*, (d) *bipartite*, (e) *transitive*, (f) *outerplanar*, (g) *without cycles of specified length*, (h) *a single K-basis*, (i) *transitively orientable*, (j) *chordal, and* (k) *interval subgraph problems.*

Proof: The graph H to be used in the edge substitution lemma is shown in Fig. 1.

COROLLARY 2. *The node-cover problem is polynomially transformable to the node-deleted* (a) *nonseparable* (b) *degree-constrained* (c) *line-invertible subgraph problems.*

Proof. The graph F to be used in the node substitution lemma is shown in Fig. 2. The node-deleted "null' subgraph problem and the node-deleted "complete" subgraph problem have already been shown to be NP-complete [7]. The Hamiltonian circuit

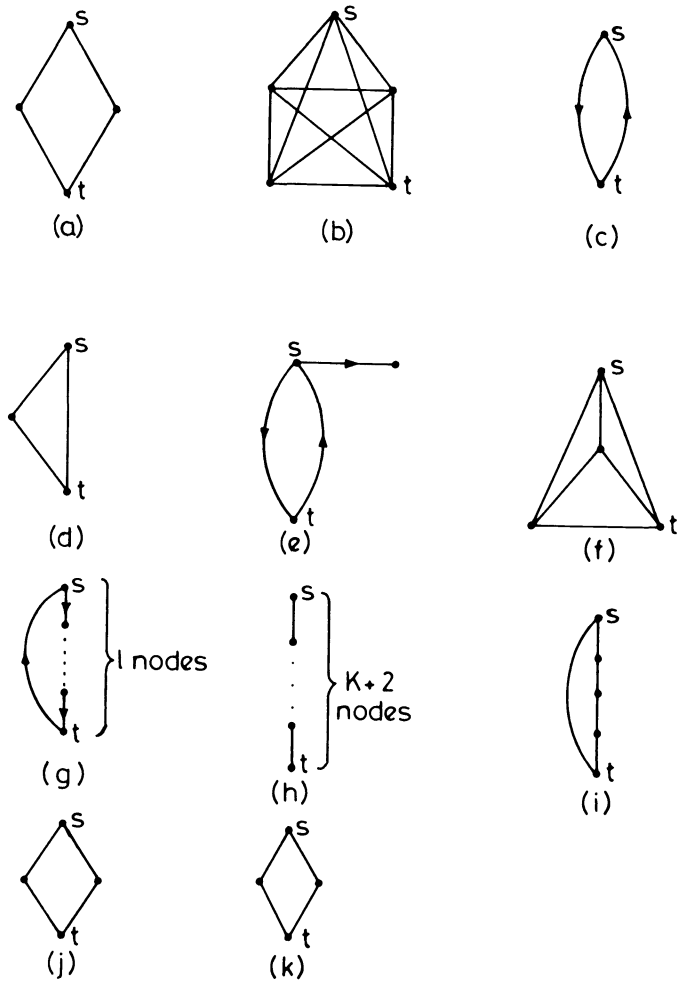


FIG. 1. *Graph H.*

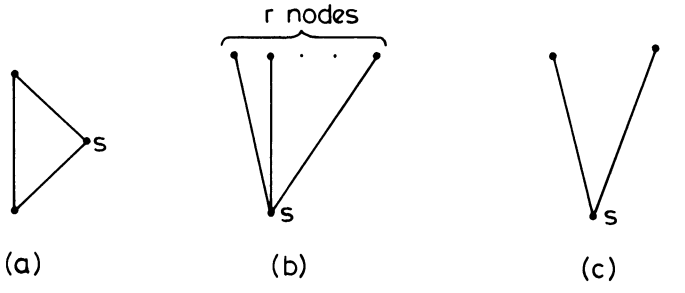


FIG. 2. *Graph F.*

problem is polynomially transformable to the node-deleted Hamiltonian subgraph problem.

As a result of corollaries 1 and 2 and the fact that the node-deletion problems are in NP, we have the following theorem.

THEOREM 1. *All the node-deletion problems in* § 2 *are NP-complete.*

4. Restricted NP-complete problems. When the input graphs of node-deletion problems are restricted to be planar or bipartite, the problems may be simplified. For example the node-deleted "complete" subgraph problem (§ 2 problem (ii)) can be solved in polynomial time when the input graph is constrained to be planar. This follows from an observation that solutions for the maximum clique problem can be obtained in polynomial time for planar graphs [3].

Another example of such reduction in complexity is the node-deleted 'acyclic' subdigraph problem (feedback node-set) in transitive digraphs. In a transitive digraph each strongly connected component is a complete subdigraph, and so using depth-first search we can find in $O(n, e)$ time the number t of the strongly connected components. Once the number t is known the cardinality of feedback node-set can be calculated as $\sum_{i=1}^{t}(n_i - 1) = n - t$. Thus we have an $O(n, e)$ algorithm for finding the feedback node-set in transitive digraphs.

On the other hand, restricting the domain of a problem does not always move a problem from NP to P class. For example, Garey, Johnson and Stockmeyer [5] have shown that the node-cover problem remains NP-complete for planar graphs and for graphs of degree at most 3. In this vein we will show that some of the restricted node-deletion problems are NP-complete. We state two general lemmas from which different restrictions become corollaries.

LEMMA 3. *Let π be a specified graph property that is determined by the components of the graph and let ρ be a restriction. Let H be a graph with two nodes s and t such that the following hold*:

(1) *The graph H is a forbidden graph for property π.*

(2) *If we take one or more H's, delete either s or t from each, and join them at the remaining s or t, then the resulting graph has property π.*

(3) *If H is substituted for all edges in a graph G with property ρ, the resulting graph has property ρ.*

(4) *The node-cover problem for G with restriction ρ is NP-complete.*

(5) *If we delete both s and t from H, then the resulting subgraph has property π.*

Then the node-deletion problem for property π with restriction ρ is NP-complete.

Proof. Transform the node-cover problem on a given graph G with restriction ρ to the node-deletion problem by substituting the graph H for every edge in G.

As the node-cover problem in planar graphs is NP-complete, the following theorem results from Lemma 3.

THEOREM 2. *All the node-deletion problems stated in* § 2 *(except "planar", "complete", and "Hamiltonian" (subgraph)) are NP-complete even when the given graph is restricted to be planar.*

In addition to this we have one more lemma regarding the restricted NP-complete problems.

LEMMA 4. *Let π be a specified graph property that is determined by the graph's components, let ρ be a restriction and suppose there is a graph H with two nodes s and t such that the following hold*:

(1) *The graph H is a forbidden graph for property π.*

(2) *If we take one, two or three H's, delete either s or t from each, and join them at the remaining s or t, then the resulting graph has property π.*

(3) *For any graph G, when H is substituted for all edges in G, the resulting graph satisfies property ρ. (This implies in particular that H satisfies ρ with G a single edge.)*

(4) *If we delete both s and t from H, then the resulting subgraph has property π.*

Then the node-deletion problem for property π with restriction ρ is NP-complete.

Proof. Transform the node-cover problem for graph of degree at most three to the node-deletion problem when every edge in G is substituted by the graph H.

As a result of the above lemma and Corollary 1, we have the following theorem:

THEOREM 3. *The node-deleted "tree" subgraph problem in bipartite graphs is NP-complete.*

As the node-cover problem in planar graphs is also NP-complete, we also have a stronger theorem.

THEOREM 4. *The node-deleted "tree" subgraph problem in planar bipartite graphs is NP-complete.*

5. Conclusions. In this paper, we have shown that 17 node-deletion problems are NP-complete. A unified approach for the transformation from the node-cover problem to the node-deletion problems using forbidden subgraphs has been developed. This may help in finding the complexities of new node-deletion problems. When the input digraph is restricted to transitive digraph, an $O(n, e)$ algorithm has been proposed for finding the feedback node-set (one of the 17 node-deletion problems). We have also proved that many (14 of them) node-deletion problems remain NP-complete for planar graphs. Further we have shown that the node-deleted "tree" subgraph problem remains NP-complete in planar bipartite graphs.

Acknowledgment. The authors wish to thank Professor J. D. Ullman for having suggested a number of improvements in Lemma 1. The authors wish to thank the two referees for their constructive criticisms.

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] G. CHARTRAND, D. GELLER and S. HEDETNIEMI, *Graphs with forbidden subgraphs*, J. Combinatorial Theory Ser. B, 10 (1971), pp. 12–41.

[3] S. A. COOK, *The complexity of theorem—proving procedures*, Proc. of Third Annual ACM Symp. on Theory of Computing, 1970, pp. 151–158.

[4] N. DEO, *Graph Theory: with Applications to Computer Science and Engineering*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[5] M. R. GAREY, D. S. JOHNSON AND L. STOCKMEYER, *Some simplified polynomial complete problems*, Proc. of Sixth Annual ACM Symp. on Theory of Computing, 1974, pp. 47–64.

[6] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[7] R. M. KARP, *Reducibility among cominatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, NY, 1972, pp. 85–104.

[8] ———, *On the computational complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–58.

[9] S. K. SAHNI, *Computationally related problems*, this Journal, 3 (1974), pp. 262–279.

[10] P. J. SLATER, *R-Domination in graphs*, J. Assoc. Comput. Mach., 23 (1976), pp. 446–450.

[11] J. D. ULLMAN, *Complexity of sequencing problems*, Computer and Job-Shop Scheduling Theory, E. G. Coffman, ed., John Wiley, New York, 1976.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0014 $01.00/0

A FAMILY OF ALGORITHMS FOR POWERING SPARSE POLYNOMIALS*

DAVID K. PROBST† AND VANGALUR S. ALAGAR†

Abstract. We discuss four new algorithms from a family of algorithms for computing integer powers of sparse polynomials. The four algorithms form a sequence of successively better algorithms; even the first member of the sequence shows an improvement in the leading term of the cost function in comparison with the best previously known binomial-expansion algorithm. To quote one result, if f is a 32-term sparse polynomial, computing $f * f^9$ takes 8.75×10^9 multiplications, while the best new algorithm computes f^{10} from scratch using 1.125×10^9 multiplications. The time and space analyses given support the conjecture that the best new algorithm is optimal for time and space within the family of sequential binomial-expansion algorithms for this problem. If the input polynomial has t terms, then the nth power may be computed by this algorithm with a time complexity of

$$\frac{t^n}{n!} + t^{n-1}\left[\frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!}\right] + O(t^{n-2}), \qquad n > 2,$$

and a space complexity of

$$\frac{t^{n-1}}{2^{2n-4}(n-1)!} + O(t^{n-2}), \qquad n > 2.$$

Key words. powers of polynomials, sparse polynomials, analysis of algorithms, symbolic computation

1. Introduction. In a previous paper [2] we made a comparative study of several sequential algorithms for computing integer powers of sparse polynomials. We emphasized the various design decisions which led to three new algorithms having significantly lower time complexities than the best algorithm proposed in [4]. We conjectured that the best of the new algorithms was optimal for time, in the sense that the leading terms of its time-complexity cost function could not be improved on, within the family of binomial-expansion algorithms. In this paper, we extend our previous results in several directions. In particular, we give a fourth new algorithm with low space complexity, and slightly improved time complexity. Our analysis shows why the newest sequential algorithm is difficult to improve on, and supports our conjecture that it is optimal for both time and space within the binomial-expansion family whenever the problems are of at least moderate size.

Several approaches to powering sparse polynomials are known, including repeated multiplication [5], binomial expansion [4] and multinomial expansion [6]. Reference [4] contains a comparison of these approaches. Our analysis supports the analysis given there which suggests that binomial expansion is by far the most promising general approach. Neither of the two binomial-expansion algorithms proposed in [4], viz., BINA and BINB, is optimal for time or space, although BINB is the better algorithm of the two. We discuss four successively-better superior algorithms from the binomial-expansion family, viz., BINC, BIND, BINE and BINF. We obtain, with BINC, an improvement in the coefficient of the leading term of the BINB time complexity and,

* Received by the editors December 29, 1977, and in final revised form November 29, 1978.

† Department of Computer Science, Concordia University, Montreal, Quebec, Canada H3G 1M8. This work was supported in part by the National Research Council under Grant A3552.

with BINE, an improvement in the leading term of the BINB space complexity. Selected values of the various cost functions are tabulated in the appendices.

In all cases we make an analysis of intrinsic time complexity and intrinsic space complexity rather than use the results of run-time tests. The computational model characterizes the problem domain; the cost model defines how intrinsic complexity is measured. We require that the input polynomials be completely or almost completely sparse to the power sought (definition follows). We measure the time complexity of a sequential algorithm by the number of coefficient multiplications which occur. We say that the space complexity of an algorithm is S if we can produce an implementation of that algorithm in which the central memory requirements for the storage of intermediate results do not exceed S terms. One term will require several central memory words.

We represent sparse polynomials in one or more variables with integer coefficients as a sum of monomials, where each monomial is of the form $c(\prod x_i^{\alpha_i})$. Let t be the number of nonzero terms in this representation. We say that f is *completely sparse to power n* if and only if f^i, when expanded, for all i from 1 to n, contains exactly the number of terms of the t-term multinomial expansion. This means that no collection of like terms is possible with respect to such an expansion. The model of sparse polynomial is due to Morven Gentleman [5]. The assumption of sparsity affects the design of the powering algorithms in two ways. First, it is reasonable to multiply sparse polynomials by multiplying each term of one by each term of the other. Second, the new algorithms have been designed so that, when the input polynomial is completely sparse, no like terms are ever formed during polynomial multiplication; hence, no sorting routines are required. When the input polynomial is almost sparse, the analysis remains correct, but the relatively few like terms are not collected. Johnson [7] shows the high cost of exponent comparisons in large polynomial multiplications when sorting is required.

We assume that multiplication and addition costs do not grow with the size of the result. In the new algorithms, the computation consists exclusively of multiplying polynomials by binomial coefficients, and multiplying polynomials by polynomials, with no collection of like terms. Multiplying a monomial by a monomial requires a coefficient multiplication and an exponent set addition; multiplying a monomial by a binomial coefficient requires a coefficient multiplication. The coefficient multiplication count is therefore an accurate reflection of the run-time cost. This cannot be guaranteed when an algorithm contains sorting routines.

This paper is organized as follows: we describe the various design decisions within the family of binomial-expansion algorithms and show, for sequential algorithms, which combination of decisions minimizes both time complexity and space complexity; briefly review the description and time analysis of algorithm BINC; describe and give time and space analyses of algorithms BINE and BINF; and, in the conclusion, briefly compare the new algorithms to some existing algorithms.

2. Design decisions. The discussion in [2] of the family of binomial-expansion algorithms did not include space considerations. Analysis now shows that the algorithm which had been preferred for its low time complexity can be adapted to yield an algorithm with low time and space complexity. We show the place of the new algorithm within the algorithm family.

Analysis suggests that binomial expansion is the best general approach for powering sparse polynomials. We choose therefore to compute f^n as

$$f^n = (f_1 + f_2)^n = \sum_{r=0}^{n} \binom{n}{r} f_1^r f_2^{n-r}.$$

We study various design decisions which, by successive refinements, ultimately yield fully-specified algorithms for the problem. It seems likely that it is the binomial-expansion family which contains the algorithms with the least time complexity and the least space complexity, and that these algorithms are in fact achievable provided that one chooses the correct means of achieving the various subgoals implicit in binomial expansion.

We represent the algorithm family as a tree. The root is the problem, and the terminal nodes are the algorithms. Each nonterminal node represents a decision point; the branches which exit from a node are the possible decisions. As a graphical convention we draw the branch corresponding to what we regard as the best decision on the left. A branch is better than another if it leads to an algorithm with lower time and space complexity. We follow Dijkstra [3]:

> A program should be conceived and understood as a member of a family · · · ·

When we split f, we must choose the relative sizes of f_1 and f_2. This choice affects the time complexity of generating and combining subpolynomial powers, and the space complexity of storing them. Analysis of many algorithms from within the binomial-expansion family shows that splitting which is as even as possible reduces both time and space complexity. Even splitting reduces generation costs more than it increases combining costs, and also limits the size of the largest subpolynomial.

Another splitting decision arises when one considers alternative means of generating subpolynomial powers. Prior to [2], binomial-expansion algorithms used (essentially) repeated multiplication for this task. In contrast, the new algorithms use binomial expansion. Therefore, the original polynomial is split recursively until the monomial level is reached; this is called "multilevel splitting." Multilevel splitting permits both lower time complexity and lower space complexity.

Multilevel, even splitting of a polynomial may be displayed graphically using a binary tree. A "term group" is one or more terms from the polynomial; the tree is called the "term group tree." The polynomial goes into the root; the two halves of the polynomial go into the left and right subnodes, and so on, recursively. If one half is larger, it goes into the left subnode.

The multilevel algorithms generate subpolynomial powers using binomial expansion in one of three distinct ways. In recursion, powers of subpolynomials are obtained by applying the original algorithm to the subnodes. Separate recursive application for distinct powers of a subpolynomial leads to some recomputation. In binary merge, which is a form of dynamic programming, one keeps track of subproblems and never solves the same problem twice. Starting from the terminal nodes, one first generates powers of subnodes, and then powers of father nodes, and so on, until ultimately one is able to generate the desired power of the root. In distribution, which is a modified form of binary merge, one minimizes the space required to store solutions to subproblems by finding the minimum set of such solutions from which the final answer can be computed using only simple loops. Up to the level of the subsubnodes of the root, distribution proceeds exactly as binary merge. One does not store powers of subnodes of the root. Rather one stores certain products of the form $\binom{n}{r} f_1^r$ which occur in the binomial expansion of the root. From these and powers of subsubnodes, the final answer is computed.

One reduces both time and space complexity by using the "smaller" technique for combining powers of subpolynomials. We assume that binomial coefficients are available free. The products $\binom{n}{r} f_1^r f_2^{n-r}$ may be computed either (a) left to right, or (b) by first

multiplying the smaller polynomial by $\binom{n}{r}$, and then this result by the other polynomial. In itself, the smaller technique reduces the number of coefficient multiplications. In combination with distribution, it also reduces space complexity by reducing the size of the $\binom{n}{r} f_1^r$. We call the classical technique left-to-right.

The whole set of what are apparently the optimal decisions for both space and time within the family of sequential binomial-expansion algorithms can be diagrammed in the following way:

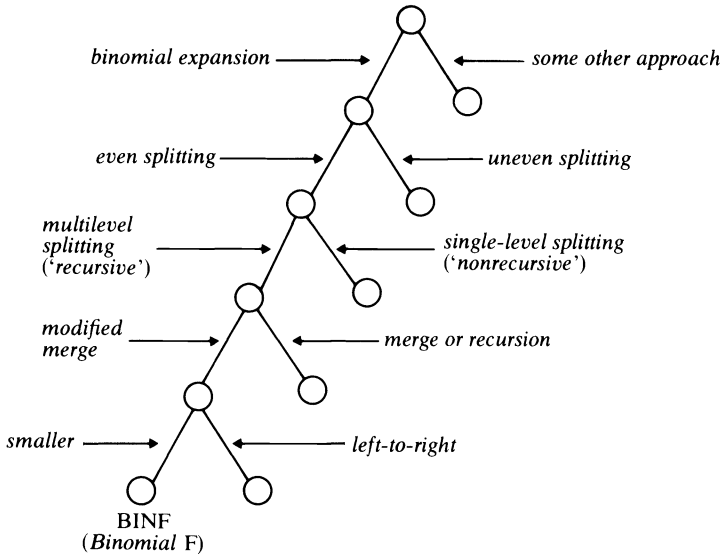


FIG. 2.1. *The program family.*

We have consistently drawn the apparently optimal choice on the left, and the other choice (or choices) on the right. Our analysis BINF is optimal for time and space within the entire family of sequential binomial-expansion algorithms for computing integer powers of sparse polynomials.

3. Mathematical preliminaries. The following results are either standard (e.g., Knuth [8]) or have appeared in Gentleman [5] or Fateman [4]. Except for slight differences, they have been previously collected in [4].

THEOREM 3.1. $\sum_{i=0}^{n} \binom{r+i}{r} = \binom{r+n+1}{r+1} = \binom{r+n+1}{n}$.

THEOREM 3.2. $\sum_{i=0}^{n} \binom{r+i}{r}\binom{r+n-i}{r} = \binom{2r+n+1}{n}$.

THEOREM 3.3. *Let f be a t-term polynomial which is completely sparse to power n. If* size (f) *gives the number of terms in f, then* size $(f^n) = \binom{t+n-1}{n}$, $t \geqq 2$.

THEOREM 3.4. *In combining powers in the binomial expansion of a completely sparse polynomial, the multiplication of a polynomial f by a polynomial g may be obtained with a cost of* size $(f) \cdot$ size (g).

Proof. By writing f^n as

$$f^n = f_1^n + f_2^n + \sum_{r=1}^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

and observing that

$$\binom{t+n-1}{n} = 2\binom{t/2+n-1}{n} + \sum_{r=1}^{n-1} \binom{t/2+r-1}{r}\binom{t/2+n-r-1}{n-r}$$

we see that if any of the terms in the cross products did combine, f^n would have fewer terms than is required for a completely sparse polynomial. As no terms combine, the cost of multiplying f by g is measured by the number of monomial products, which is size $(f) \cdot$ size (g). Q.E.D.

The function size (f) has already been introduced. We provide an alternate functional form: size (s, n) is size (f^n) when size $(f) = s$. The sparsity of f is assumed.

$$\text{size } (s, n) = \binom{s+n-1}{n} = \frac{1}{n!} \sum_{j=1}^{n} \begin{bmatrix} n \\ j \end{bmatrix} s^j \tag{3.1}$$

where the square brackets indicate Stirling numbers of the first kind; cf. [8]. We introduce a further notation.

$$\text{group } (s, n) \triangleq \sum_{j=1}^{n} \text{size } (s, j)$$

$$\text{group } (s, n) = \binom{s+n}{n} - 1 = \frac{1}{n!} \sum_{j=1}^{n} \begin{bmatrix} n+1 \\ j+1 \end{bmatrix} s^j. \tag{3.2}$$

Finally, we state

$$\binom{s+n}{n-1} - n = \frac{1}{(n-1)!} \sum_{j=1}^{n-1} \sum_{k=0}^{n-j-1} \begin{bmatrix} n-1 \\ j+1 \end{bmatrix} \binom{k+j}{k} 2^k s^j. \tag{3.3}$$

We should also mention [4]

THEOREM 3.5. *No algorithm can compute f^n, the n-th power of an arbitrary polynomial, in fewer than* size (f^n) − size (f) *multiplications.*

Suppose now that f is completely sparse to power n and that size $(f) = t$. If we use $L(t, n)$ to denote this theoretical lower limit, we have

$$L(t, n) = \binom{t+n-1}{n} - t = \frac{1}{n!} \sum_{j=1}^{n} \begin{bmatrix} n \\ j \end{bmatrix} t^j - t, \tag{3.4}$$

that is,

$$L(t, n) = \frac{t^n}{n!} + \frac{t^{n-1}}{2(n-2)!} + O(t^{n-2}). \tag{3.5}$$

4. Algorithms.

4.1. Algorithm BINC. We repeat the description and time analysis of BINC, the first of the new algorithms announced in [2]. The algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, recursion for subpolynomial powers, and left-to-right for cross products. It differs from BINB (the best algorithm in [4]) only in that repeated multiplication has been replaced by recursion. The analysis is given for $t = 2^k$. The original polynomial f is split into f_1 and f_2 such that size (f_1) = size $(f_2) = t/2$. Powers of subpolynomials other than monomials are obtained

by recursive application of BINC, powers of monomials by repeated squaring. Cross products are formed left-to-right.

Analysis. Let $C(t, n)$ be the number of coefficient multiplications to compute f^n when size $(f) = t$. If we expand f^n as

$$f^n = f_1^n + f_2^n + \sum_1^{n-1} \binom{n}{r} f_1^r f_2^{n-r}$$

we have

$$C(t, n) = 2C(t/2, n) + \sum_1^{n-1} Q_r \tag{4.1}$$

where Q_r is the sum of costs itemized as follows.

Step	Compute	Cost
1	f_1^r	$C(t/2, r)$
2	f_2^{n-r}	$C(t/2, n-r)$
3	$\binom{n}{r} f_1^r$	$\binom{t/2+r-1}{t/2-1}$
4	$\binom{n}{r} f_1^r f_2^{n-r}$	$\binom{t/2+r-1}{t/2-1}\binom{t/2+n-r-1}{t/2-1}$

Therefore

$$\begin{aligned} C(t, n) = 2C(t/2, n) + 2\sum_1^{n-1} C(t/2, r) + \sum_1^{n-1} \binom{t/2+r-1}{t/2-1} \\ + \sum_1^{n-1} \binom{t/2+r-1}{t/2-1}\binom{t/2+n-r-1}{t/2-1}. \end{aligned} \tag{4.2}$$

We use the summation formulas (3.1) and (3.2) to obtain

$$\begin{aligned} C(t, n) = \binom{t+n-1}{n} - 2\binom{t/2+n-1}{n} + \binom{t/2+n-1}{n-1} \\ -1 + 2\sum_1^n C(t/2, r), \quad \text{where } C(1, n) = \lceil \log_2 n \rceil. \end{aligned} \tag{4.3}$$

Given a closed form for $C(t, n)$, we may obtain one for $C(t, n+1)$ by using the formula:

$$\begin{aligned} C(t, n+1) = nt + \frac{t-1}{n+1}\binom{t+n-1}{n} + \sum_1^k 2^{r-1}\binom{t/2^r+n-1}{n} \\ + \sum_1^k 2^{r-1} C(t/2^{r-1}, n) \end{aligned} \tag{4.4}$$

obtained from (4.3) by changing n to $n+1$ and using recurrence on t. Using induction on n, we obtain the general form of $C(t, n)$.

$$C(t, n) = \sum_1^n a_i^{(n)} t^{n-i+1} + t \sum_{n+1}^{2n-1} a_i^{(n)} k^{i-n}. \tag{4.5}$$

We have at once that $a_1^{(n)} = 1/n!$. Substituting (4.5) into (4.4) and rearranging gives finally the coefficient of the second leading term. It is

$$a_2^{(n)} = \frac{1}{2(n-2)!} + \frac{3}{(n-1)!(2^{n-1}-2)}.$$

Let $B(t, n)$ be the coefficient-multiplication cost function for algorithm BINB (see [4]). We can show the superiority of algorithm BINC by subtracting cost functions. We have:

$$B(t, n) - C(t, n)$$

$$= \frac{1}{n(n-2)!2^{n-1}} t^n + \left[\frac{n-1}{(n-2)!2^{n-1}} - \frac{3}{(n-1)!(2^{n-1}-2)}\right] t^{n-1} + O(t^{n-2}). \tag{4.6}$$

It is also true, for large n, that:

$$C(t, n) - L(t, n) \cong \frac{3}{(n-1)!}\left(\frac{t}{2}\right)^{n-1} + O(t^{n-2}). \tag{4.7}$$

Both $B(t, n)$ and $C(t, n)$ approach $L(t, n)$ for large t and large n. However, the improvement in the coefficient of the leading term makes $C(t, n)$ approach $L(t, n)$ much more rapidly. Examination of tabulated values shows that BINC outperforms BINB for $n > 2$ and $t \geqq 9$.

4.2. Algorithm BIND. BIND improves on BINC by modifying just one of BINC's suboptimal decisions. Recursion is retained, but cross products are formed via the smaller technique rather than left-to-right. The analysis of the time complexity can be carried through in an exactly analogous fashion. The general form of $D(t, n)$ is

$$D(t, n) = \sum_{1}^{n} a_i^{(n)} t^{n-i+1} + t \sum_{n+1}^{2n-1} a_i^{(n)} k^{i-n}. \tag{4.8}$$

The coefficient of the leading term, i.e., $a_1^{(n)}$, is still $1/n!$. The coefficient of the second leading term, however, has been reduced. It is now

$$a_2^{(n)} = \frac{1}{2(n-2)!} + \frac{1}{(n-1)!(2^{n-2}-1)}.$$

We now describe algorithms BINE and BINF, and give new time and space analyses for both of them.

4.3. Algorithm BINE. This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, binary merge for computing subpolynomial powers, and smaller for forming cross products. More formally, we describe the algorithm as follows. As before, $f = f_1 + f_2$.

Description. (1) Create the binary term-group tree for the polynomial f:

(a) place f at the root;

(b) place f_1, size $(f_1) = \lceil \text{size } (f)/2 \rceil$, in the left subnode;

(c) place f_2 in the right subnode;

(d) repeat steps (b) and (c) until each term of f has been placed at one of the terminal nodes of the tree.

(2) For each term of the original polynomial f, compute all powers from 2 to n. This completes the processing of the terminal nodes.

(3) For each strictly interior node, both of whose subnodes have already been processed, compute all powers from 2 to n according to the following scheme:

$$(\text{node})^r = (\text{left subnode} + \text{right subnode})^r$$

expanded binomially.

(i) For $s = 1$ to $r-1$ do

(a) Multiply $\binom{r}{s}$ by whichever of $(\text{left subnode})^s$ and $(\text{right subnode})^{r-s}$ has fewer terms.

(b) Multiply the result in (a) by the remaining factor.

(ii) Collect $(\text{left subnode})^r + (\text{right subnode})^r +$ the products computed in (i).

(4) Compute the nth power of the root according to the previous scheme.

Analysis. We save all space-complexity analysis for a later section and discuss only time complexity here. We introduce the concept of a "power group triangle." The power group triangle of a node $a+b$ is the graphical representation in triangular form of all binomial expansions required to compute all powers from 1 to n of that node, given all powers from 1 to n of the two subnodes a and b, where all cross-products have been written in conformity with the "smaller" idea.

For example, when $n = 6$, the power group triangle is as shown in Fig. 4.1. It is understood here that a is the left subnode and b is the right subnode. Therefore, size $(a) \geqq$ size (b). When size (a) = size (b), powers alone determine the relative sizes of a^i and b^j. When $t = 2^k$, this will always be the case. When $t \neq 2^k$, there will be a few isolated instances where a larger power of the right subnode will have fewer terms than a smaller power of the left subnode. The power group triangle, therefore, is not always a strict embodiment of the "smaller" idea. For an actual computation, the difference between the two will be extremely small. Tabulated values of number of coefficient multiplications have been determined using a strict interpretation of "smaller."

$$a^6+b^6+a^5\cdot 6b+b^5\cdot 6a+a^4\cdot 15b^2+b^4\cdot 15a^2+a^3\cdot 20b^3$$
$$a^5+b^5+a^4\cdot 5b+b^4\cdot 5a+a^3\cdot 10b^2+b^3\cdot 10a^2$$
$$a^4+b^4+a^3\cdot 4b+b^3\cdot 4a+a^2\cdot 6b^2$$
$$a^3+b^3+a^2\cdot 3b+b^2\cdot 3a$$
$$a^2+b^2+a\cdot 2b$$
$$a+b$$

FIG. 4.1. *Power group triangle when* $n = 6$.

Our first task is to evaluate $BC(s, n)$, the total number of multiplications by binomial coefficients in a power group triangle when the subnodes a and b are of size s. This is just the sum of sizes of polynomials which are multiplied by binomial coefficients. A simple summation gives

$$BC(s,n) - BC(s,n-2) = 4\left[\binom{s+p-1}{p-1} - 1\right] + M_n\binom{s+p-1}{p} \tag{4.9}$$

where $p = \lfloor n/2 \rfloor$ and $M_n = 1(3)$ when n is even (odd).

Let x be 2(3) when n is even (odd). Therefore, $BC(s, x) = M_n\binom{s+0}{1}$. A difference scheme, counting down from n by twos, gives

$$BC(s, n) - BC(s, x) = \sum_{1}^{p-1} \left\{4\left[\binom{s+j}{j} - 1\right] + M_n\binom{s+j}{j+1}\right\}. \tag{4.10}$$

Using (3.1) for the summations and adding $BC(s, x)$ gives

$$BC(s, n) = 4\left[\binom{s+p}{p-1} - p\right] + M_n\left[\binom{s+p}{p} - 1\right]. \tag{4.11}$$

A closed form for $BC(s, n)$ is therefore

$$BC(s, n) = \sum_{j=1}^{p} \left\{\frac{M_n}{p!}\left[{p+1 \atop j+1}\right] + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \left[{p-1 \atop k+j}\right]\binom{k+j}{k} 2^k\right\} s^j \tag{4.12}$$

where we have used (3.3) and (3.2), and where it is understood that $\sum_{k=0}^{-1} \cdots$ is zero. The leading term of $BC(s, n)$ is $(M_n/p!)s^p$. As a power group is computed for each strictly interior node, by summing over all such nodes we obtain the "group binomial work" or number of multiplications by binomial coefficients involved in computing groups of powers. If the root is at level 0 and the terminal nodes at level k, then the required sum is

$$\frac{t}{2} \cdot BC(1, n) + \frac{t}{4} \cdot BC(2, n) + \cdots + 2 \cdot BC(t/4, n)$$

which is

$$\frac{t}{2} \cdot \sum_{0}^{k-2} 2^{-i} BC(2^i, n).$$

Making use of (4.12) gives the following for the group binomial work (GBW)

$$\begin{aligned} \mathrm{GBW} &= \frac{t}{2} \sum_{m=0}^{k-2} \sum_{j=1}^{p} a_j^{(n)} (2^m)^{j-1} \\ &= \frac{t}{2} \cdot a_1^{(n)}(k-1) + \frac{t}{2} \sum_{1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^j - 1}{2^j - 1} \end{aligned} \tag{4.13}$$

where

$$a_j^{(n)} = \frac{M_n}{p!}\left[{p+1 \atop j+2}\right] + \frac{4}{(p-1)!} \sum_{k=0}^{p-j-1} \left[{p-1 \atop k+j}\right]\binom{k+j}{k} 2^k. \tag{4.14}$$

In addition to group binomial work there is root binomial work (RBW). This is

$$\mathrm{RBW} = 2 \sum_{1}^{r-1} \binom{t/2+j-1}{j} + N_n\binom{t/2+r-1}{r} \tag{4.15}$$

where $r = \lceil n/2 \rceil$ and $N_n = 1(0)$ when n is even (odd). We have

$$\mathrm{RBW} = \frac{2}{(r-1)!} \sum_{1}^{r-1} \left[{r \atop j+1}\right]\left(\frac{t}{2}\right)^j + \frac{N_n}{r!} \sum_{1}^{r} \left[{r \atop j}\right]\left(\frac{t}{2}\right)^j. \tag{4.16}$$

Adding GBW and RBW gives the total number of multiplications by binomial coefficients in algorithm BINE. When $p > 1$, the leading term in that sum is

$$\frac{1}{p!}\left(\frac{t}{2}\right)^p \left[2 - N_n + \frac{M_n}{2^{p-1} - 1}\right].$$

The remaining or "nonbinomial" work is the number of coefficient multiplications required to process all terminal nodes and form all polynomial-polynomial products in the binomial expansions. As before, we distinguish group and root nonbinomial work. The group work at level k is $t(n-1)$, which is also $t \cdot \text{group}\,(1, n) - t$. If we look at the binomial expansion $(f_1+f_2)^n$, we see that the nonbinomial work involved in computing f^n given all powers of f_1 and f_2 is size $(f^n) - 2 \cdot \text{size}\,(f_1^n)$, as there is one monomial multiplication per term formed. After the group work at level k is completed, an additional nonbinomial work of $t/2 \cdot \text{group}\,(2, n) - t \cdot \text{group}\,(1, n)$ is required to compute the powers from 2 to n of all nodes at level $k-1$. The total nonbinomial work to level $k-1$ is their sum, namely, $t/2 \cdot \text{group}\,(2, n) - t$.

By continuing this argument, we see that the total nonbinomial work to level 1, i.e., for forming groups of powers, is given by $2 \cdot \text{group}\,(t/2, n) - t$. The root nonbinomial work is simply $\text{size}\,(t, n) - 2 \cdot \text{size}\,(t/2, n)$. The total nonbinomial work is their sum, which is

$$\text{size}\,(t, n) + 2 \cdot \text{group}\,(t/2, n-1) - t$$

or

$$\frac{1}{n!}\sum_{1}^{n}\begin{bmatrix} n \\ j \end{bmatrix} t^j + \frac{2}{(n-1)!}\sum_{1}^{n-1}\begin{bmatrix} n \\ j+1 \end{bmatrix}\left(\frac{t}{2}\right)^j - t. \tag{4.17}$$

This may also be written as

$$\frac{t^n}{n!} + t^{n-1}\left[\frac{1}{2(n-2)!} + \frac{1}{2^{n-2}(n-1)!}\right] + O(t^{n-2}). \tag{4.18}$$

Since the binomial work is $O(t^p)$, $p = \lfloor n/2 \rfloor$, these are in fact the leading terms of the BINE time-complexity cost function. Indeed, we may write $E(t, n)$ as

$$E(t, n) = L(t, n) + 2 \cdot \text{group}\,(t/2, n-1) + \text{BW},$$

where BW is the total binomial work and is $O(t^p)$. This last result shows that deviations of $E(t, n)$ from $L(t, n)$ in the leading terms result exclusively from nonbinomial work and would persist in exactly the same form even if the total binomial cost were reduced to zero.

The complete, closed-form expression for $E(t, n)$ is the sum of the closed-form expressions for total binomial work and total nonbinomial work. We have

$$\begin{aligned} E(t, n) = {} & \frac{1}{n!}\sum_{1}^{n}\begin{bmatrix} n \\ j \end{bmatrix} t^j + \frac{2}{(n-1)!}\sum_{1}^{n-1}\begin{bmatrix} n \\ j+1 \end{bmatrix}\left(\frac{t}{2}\right)^j - t \\ & + \frac{2}{(r-1)!}\sum_{1}^{r-1}\begin{bmatrix} r \\ j+1 \end{bmatrix}\left(\frac{t}{2}\right)^j + \frac{N_n}{r!}\sum_{1}^{r}\begin{bmatrix} r \\ j \end{bmatrix}\left(\frac{t}{2}\right)^j \\ & + \frac{t}{2} \cdot a_1^{(n)}(k-1) + \frac{t}{2} \cdot \sum_{1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^j - 1}{2^j - 1} \end{aligned} \tag{4.19}$$

where

$$a_j^{(n)} = \frac{M_n}{p!}\begin{bmatrix} p+1 \\ j+1 \end{bmatrix} + \frac{4}{(p-1)!}\sum_{k=0}^{p-j-1}\begin{bmatrix} p-1 \\ k+j \end{bmatrix}\binom{k+j}{k} 2^k.$$

We have:

$$E(t, 2) = t^2/2 + t/2 + kt/2, \tag{4.20}$$

$$E(t, 3) = t^3/6 + 3t^2/4 + t/3 + 3kt/2, \tag{4.21}$$

$$E(t, 4) = t^4/24 + 7t^3/24 + 29t^2/24 - 2t/3 + 11kt/4. \tag{4.22}$$

The improvement of BINE over BIND is given by

$$D(t, n) - E(t, n) = t^{n-1} \frac{1}{(n-1)!(2^{n-2}-1)2^{n-2}} + O(t^{n-2}). \tag{4.23}$$

4.4. Algorithm BINF. This algorithm is specified by the design decisions: binomial expansion, even splitting, multilevel splitting, modified merge for subpolynomial powers, and smaller. BINF differs from BINE in the handling of multiplication by binomial coefficients near the top of the term-group tree. The number of coefficient multiplications in the nonbinomial work, which is responsible for the leading terms of the time-complexity cost function, does not change. As before, $f = f_1 + f_2$.

Description. (1) Form the binary term-group tree for the polynomial f in the usual manner: place f at the root, split f as evenly as possible placing the slightly larger half (if the sizes are not identical) in the left subnode and the other half in the right subnode, and continue this process until only monomials are left.

(2) Process the terminal nodes (original monomials of f) by forming all powers from 2 to n.

(3) For all strictly interior nodes up to level 2, both of whose subnodes have already been processed, compute all powers from 2 to n by:

$$(\text{node})^r = (\text{left subnode} + \text{right subnode})^r$$

expanded binomially.

(i) For $s = 1$ to $r-1$ do

(a) Multiply $\binom{r}{s}$ by whichever of $(\text{left subnode})^s$ and $(\text{right subnode})^{r-s}$ has fewer terms.

(b) Multiply the result in (a) by the remaining factor.

(ii) Collect $(\text{left subnode})^r + (\text{right subnode})^r +$ the products computed in (i).

(4) For the left and right subnodes of the root compute all powers from 2 to n *except* for the following:

(a) For the left subnode, all powers from 2 to $\lfloor (n-1)/2 \rfloor$, if any.

(b) For the right subnode, all powers from 2 to $\lceil (n-1)/2 \rceil$, if any.

(5) Compute the nth power of the root according to the following scheme. Use binomial expansion in the manner of (3), that is, form each cross product of the expansion as (larger subpolynomial power · (binomial coefficient · smaller subpolynomial power)). If the smaller subpolynomial power has already been computed in (4), compute the inner parenthesis as indicated. Otherwise, compute the inner parenthesis by distributing the binomial coefficient over the binomial expansion of the smaller subpolynomial power. That is, if g^r is the smaller subpolynomial power, compute $\binom{n}{r} g^r$ as

$$\binom{n}{r} g_1^r + \binom{n}{r} g_2^r + \sum_1^{r-1} \binom{n}{r} \binom{r}{s} g_1^s g_2^{r-s}$$

where

$$f = g + h \quad \text{and} \quad g = g_1 + g_2.$$

Analysis. We are interested in evaluating $E(t, n) - F(t, n)$, the reduction in number of multiplications by binomial coefficients caused by not computing all powers from 2 to n of the two nodes at level 1. As the use of distribution to bypass the independent computation of some subpolynomial powers effectively reduces the amount of root binomial work, we reevaluate the function RBW(t, n).

Let the binomial expansion of the desired power of the root be written according to the smaller idea. Products of the form $\binom{n}{s} g^s$ will occur $n-1$ times. In each case, if $g = g_1 + g_2$, we compute $\binom{n}{s} g^s$ as

$$\binom{n}{s} g_1^s + \binom{n}{s} g_2^s + \sum_1^{s-1} \binom{n}{s}\binom{s}{j} g_1^j g_2^{s-j}.$$

This means that, if we allow for the cost of computing the $\binom{n}{s}\binom{s}{j}$, we avoid the cost of multiplying $\binom{n}{s}$ by the product terms in the binomial expansion of g^s. The cost of forming the $\binom{n}{s}\binom{s}{j}$ is $\lfloor s/2 \rfloor$, the number of distinct $\binom{s}{j}$, $1 \leqq j \leqq s-1$; the reduction is that the binomial cost of computing $\binom{n}{s} g^s$ drops from size (g^s) to $2 \cdot$ size (g_1^s). The cost of multiplying $\binom{n}{s}\binom{s}{j}$ by the appropriate subpolynomials is already included in the binomial cost of computing g^s, i.e., in GBW (t, n). When $s = 1$, there are no product terms in the binomial expansion of g^s, and hence no reduction. We see therefore that RBW (t, n) is now given by

$$\begin{aligned} \text{RBW} &= 4 \cdot \sum_1^{r-1} \binom{t/4 + j - 1}{j} + 2N_n \cdot \binom{t/4 + r - 1}{r} \\ &= 4\left[\binom{t/4 + r - 1}{r-1} - 1\right] + 2N_n \cdot \binom{t/4 + r - 1}{r} \end{aligned} \tag{4.24}$$

$$= 4 \cdot \text{group}\,(t/4, r-1) + 2N_n \cdot \text{size}\,(t/4, r) \tag{4.25}$$

$$= \frac{4}{(r-1)!} \sum_1^{r-1} \begin{bmatrix} r \\ j+1 \end{bmatrix} \left(\frac{t}{4}\right)^j + \frac{2N_n}{r!} \sum_1^r \begin{bmatrix} r \\ j \end{bmatrix} \left(\frac{t}{4}\right)^j. \tag{4.26}$$

Therefore, while the leading term of the total binomial work in BINE is given by

$$\frac{1}{p!}\left(\frac{t}{2}\right)^p \left[2 - N_n + \frac{M_n}{2^{p-1} - 1}\right], \qquad p > 1, \tag{4.27}$$

the leading term of the total binomial work in BINF is given by

$$\frac{1}{p!}\left(\frac{t}{2}\right)^p \cdot \left[\frac{2 - N_n}{2^{p-1}} + \frac{M_n}{2^{p-1} - 1}\right], \qquad p > 1. \tag{4.28}$$

We get an improvement, not in t^n, nor in t^{n-1}, but merely in t^p.

Adding the costs as before, but using the new value of RBW (t, n) gives the complete, closed-form expression for $F(t, n)$. However, we must also add $BB(n)$, the total binomial-coefficient-times-binomial-coefficient cost. As these products arise only for $2 \leqq s \leqq p$, $BB(n)$ is given by

$$BB(n) = \sum_{2}^{p} \lfloor j/2 \rfloor$$

$$= \tfrac{1}{4}(p^2 - \delta_p)$$

where $\delta_p = O(1)$ when p is even (odd). The final result is therefore

$$F(t, n) = \frac{1}{n!}\sum_{1}^{n} \begin{bmatrix} n \\ j \end{bmatrix} t^j + \frac{2}{(n-1)!}\sum_{1}^{n-1} \begin{bmatrix} n \\ j+1 \end{bmatrix}\left(\frac{t}{2}\right)^j - t$$

$$+ \frac{4}{(r-1)!}\sum_{1}^{r-1} \begin{bmatrix} r \\ j+1 \end{bmatrix}\left(\frac{t}{4}\right)^j + \frac{2N_n}{r!}\sum_{1}^{r} \begin{bmatrix} r \\ j \end{bmatrix}\left(\frac{t}{4}\right)^j \tag{4.29}$$

$$+ \tfrac{1}{4}(p^2 - \delta_p) + \frac{t}{2} \cdot a_1^{(n)}(k-1) + \frac{t}{2} \cdot \sum_{1}^{p-1} a_{j+1}^{(n)} \frac{(t/2)^j - 1}{2^j - 1}$$

where

$$a_j^{(n)} = \frac{M_n}{p!}\begin{bmatrix} p+1 \\ j+1 \end{bmatrix} + \frac{4}{(p-1)!}\sum_{k=0}^{p-j-1} \begin{bmatrix} p-1 \\ k+j \end{bmatrix}\binom{k+j}{k} 2^k$$

and p, r, N_n, and M_n have their usual meanings.

4.5. Space complexities. To obtain space-complexity functions for these algorithms, we need to be more specific about implementation strategies, and in particular to discuss data structures and storage management. The most interesting comparison is between the space complexity of BINB, a single-level algorithm which uses repeated multiplication to generate subpolynomial powers, and the space complexities of BINE and BINF, two new multilevel algorithms which use dynamic programming for the same task. We consider implementations of these algorithms, and attempt to minimize space complexity.

We restrict the analysis to in-core implementations. A term of a multivariate polynomial is specified by a coefficient and an exponent set. The former requires a single central-memory word; the latter requires E words, $E \geqq 1$. E can be 1: we represent an exponent set by a bit string, with portions of a central-memory word given over to each variable. A polynomial may be represented as a linked list by adding a link field to each term. In this case, if the link uses P words, $P \leqq 1$, one term will require $(1 + E + P)$ central-memory words.

As a lemma, we establish the space complexity of computing f^n by repeated multiplication when size $(f) = t$ and f is completely sparse to power n. We make a best-case analysis, and assign a lower bound to the space complexity. We adopt the merge approach to multiplying sorted sparse polynomials recommended by Aho, Hopcroft, and Ullman [1]. The final step is computing $f \cdot f^{n-1}$. Assuming f and f^{n-1} sorted, we form t subsequences of length size $(t, n-1)$, and then merge these subsequences to obtain f^n. At the very least, we require space to merge the last subsequence with the union of previous subsequences. We can sort in place by applying a list merge sort to polynomials represented as linked lists. The combined size of the last two lists to be merged is somewhat larger than size (t, n). As a lower limit, therefore, we assign a space complexity of size (t, n) provided a link field is attached to each term.

Indeed, we believe that a linked-list representation for polynomials makes good sense for all the sequential algorithms considered here.

4.6. Space analysis for BINB. We consider an in-core linked-list implementation, where the final answer is written to disc, and determine the central memory required to store intermediate results. We split f evenly into f_1 and f_2, and obtain powers of subpolynomials using repeated multiplication. A power of f_1 or f_2 not needed to generate higher powers, and already used in the binomial expansion, need not be retained in core. We form all powers from 2 to p of f_1 and f_2, where $p = \lfloor n/2 \rfloor$. If n is even, we form $\binom{n}{p} f_1^p f_2^p$. We then form the powers from $p+1$ to n of f, using each power in the expansion as soon as generated, releasing powers of f_2 whenever possible, and not retaining powers of f_1 beyond f_1^p. We proceed similarly for higher powers of f_2. After forming $\binom{n}{p} f_1^p f_2^p$ we need to retain the powers from 1 to $p-1$ of f_1, and f_2^p. We also require space to generate f_1^n. So, at best, the space complexity of BINB is size $(t/2, n)$ + group $(t/2, p)$. This may also be written as

$$S_B = \frac{1}{n!} \sum_1^n \left[n \atop j \right] \left(\frac{t}{2}\right)^j + \frac{1}{p!} \sum_1^p \left[p+1 \atop j+1 \right] \left(\frac{t}{2}\right)^j. \tag{4.30}$$

As a lemma, we establish that, if f_1^r and f_2^{n-r} are present in core, then $\binom{n}{r} f_1^r f_2^{n-r}$ may be obtained with space complexity not exceeding the space, if any, required for the result. That is, there is no working storage. We form the product by retrieving each term of the smaller polynomial, multiplying by the binomial coefficient, and then retrieving and multiplying by each term of the larger polynomial. A final lemma establishes the space complexity of dynamic programming: If all powers from 1 to n of all nodes at some level of the term-group tree are present in core, then the space complexity of forming the next higher level does not exceed the space, if any, required to store the higher level. For any g and any s, the terms of g^s are the terms from g_1^s, g_2^s, and all terms from the cross products in the binomial expansion of g^s. To form a higher level from a lower level, we need only add the terms from all cross products.

4.7. Space analysis for BINE and BINF. Again, we consider in-core linked-list implementations, and write the final answer to disc. The levels of the term-group tree are numbered from 0(root level) to k(terminal nodes). Storing all groups of powers at level 1 would require $2 \cdot \text{group } (t/2, n)$ terms. Yet there is no need to retain the nth power of any subpolynomial. If g_1^n and g_2^n have been written to disc, and the other powers of g_1 and g_2 are available in core, writing the appropriate cross products to disc writes g^n to disc. Always writing the nth power of every subpolynomial to disc reduces the space complexity of BINE to $2 \cdot \text{group } (t/2, n-1)$. This may also be written as

$$S_E = \frac{2}{(n-1)!} \sum_1^{n-1} \left[n \atop j+1 \right] \left(\frac{t}{2}\right)^j. \tag{4.31}$$

The tasks of computing the $\binom{n}{r} f_1^r f_2^{n-r}$ are independent subtasks. We decide, therefore, to generate the powers of nodes at level 1 only as they are needed in the binomial

expansion. When t is of at least moderate size, the most space required will be that to generate f_1^{n-1}. Taking into account the groups of powers at level 2, and using a bar to distinguish the new space complexity function, we have $\underline{S}_E =$ size $(t/2, n-1)+4 \cdot$ group $(t/4, n-1)$, which is also

$$\underline{S}_E = \frac{1}{(n-1)!}\sum_{1}^{n-1}\begin{bmatrix} n-1 \\ j \end{bmatrix}\left(\frac{t}{2}\right)^j + \frac{4}{(n-1)!}\sum_{1}^{n-1}\begin{bmatrix} n \\ j+1 \end{bmatrix}\left(\frac{t}{4}\right)^j. \tag{4.32}$$

It is difficult to avoid in-core storage for the powers from 1 to $n-1$ of the nodes at level 2. An improvement at level 1 with respect to storage is, however, possible if we make use of distribution to precompute directly products of the form $\binom{n}{r} f_1^r$; this is the characterizing idea of algorithm BINF. In each cross product of the binomial expansion of f^n, let the larger polynomial be called the "a-list", and the product of the smaller polynomial and the binomial coefficient be called the "b-list." By distribution (in the algebraic sense), any b-list may be represented in terms of constants and powers of nodes available at level 2. If we have storage for all of level 2 and for the largest b-list at level 1, we may compute f^n with no further central memory. This gives algorithm BINF a space complexity of size $(t/2, p)+4 \cdot$ group $(t/4, n-1)$, $p = \lfloor n/2 \rfloor$. This may also be written as

$$S_F = \frac{1}{p!}\sum_{1}^{p}\begin{bmatrix} p \\ j \end{bmatrix}\left(\frac{t}{2}\right)^j + \frac{4}{(n-1)!}\sum_{1}^{n-1}\begin{bmatrix} n \\ j+1 \end{bmatrix}\left(\frac{t}{4}\right)^j. \tag{4.33}$$

When $n > 2$, the leading term of S_F is given by

$$\frac{t^{n-1}}{2^{2n-4}(n-1)!}.$$

We give the computational procedure to form one cross-product. The b-list is precomputed and stored. Let the a-list be g^r, and its binomial expansion be

$$g^r = g_1^r + g_2^r + \sum \binom{r}{s} g_1^s g_2^{r-s}.$$

The rth powers g_1^r and g_2^r are easily retrieved and multiplied by the b-list. Each of the $r-1$ cross products has a larger polynomial (called "large") and a smaller polynomial (called "small"). We therefore execute the following nested loop, expressed in pseudo-Pascal, $r-1$ times.

```
for each term of small do
    temp := binomial coefficient * small [i];
    for each term of large do
        for each term of b-list do
            write (temp * large [j] * b-list [k])
```

4.8. Comparison of space complexities. The BINE and BINF complexities represent the space that would be used by those implementations; the BINB complexity is a generous lower limit. Asymptotic arguments, say for fixed n and large t, show the superiority of the BINE and BINF complexities. But there is a danger that the

asymptotically superior algorithm becomes superior just as we pass beyond the bounds of the practically computable. We make a more careful comparison of S_B (4.30) and S_F (4.33). These functions were tabulated for various values of $t = 2^k$ and n. When $t = 4$, BINB requires less space. When $t \geqq 8$, BINF requires less space; in the one case of equality ($t = 8$, $n = 3$), the BINB lower limit can be shown to be inapplicable. Examination of the tabulated results shows the superiority of BINF over a wide range of the practically computable. For a small problem $t = 8$, $n = 10$, $S_B = 411$, $S_F = 272$; for a large problem, $t = 32$, $n = 5$, $S_B = 15{,}656$, $S_F = 2{,}112$. One may conclude, therefore, that dynamic programming, coupled with intelligent memory management, leads to space improvements more dramatic even than those in the time domain.

5. Conclusion. We have analyzed four new algorithms, viz., BINC, BIND, BINE, and BINF, for computing integer powers of sparse polynomials. All four algorithms are based on using binomial expansion systematically for obtaining all powers of all polynomials other than monomials; they are successive improvements of the systematic binomial-expansion approach. Binomial expansion is an extremely powerful general approach to powering sparse polynomials. For example, if $R(t, n)$ is the cost for repeated multiplication, then the weak binomial-expansion algorithm BINA has a ratio $A(t, n)/R(t, n)$ which is roughly $t/(t+n-1)$ [4], while the strong binomial-expansion algorithm BINF has a ratio $F(t, n)/R(t, n)$ which is asymptotically $1/n$. This shows the degree of the superiority to repeated multiplication. Similarly, in the multinomial-expansion algorithm of Horowitz and Sahni [6], the time complexity is $4 \cdot \text{size}\ (t, n)$; the better binomial-expansion algorithms have time complexities which are asymptotically size (t, n). Values of $L(t, n) = \text{size}\ (t, n) - t$ are tabulated in an appendix. In a previous paper [2], we have reanalyzed the time complexity of algorithm BINB, and obtained a slightly larger coefficient for the second term in the cost function in comparison with that given in [4]. Tabulation shows the clear superiority of the new algorithms to BINB no matter which analysis is used. In our analysis, BINC and BIND outperform BINB for $n > 2$ and $t \geqq 9$, while BINE and BINF outperform BINB for $n > 2$ and $t \geqq 2$. The time complexities of the sequential algorithms BINA through BINE form a monotonically decreasing sequence, as measured by the leading two terms of the respective cost functions. In comparison with a generous lower limit on the space complexity of BINB, BINE has lower space complexity for large problems, while BINF has lower space complexity for all but small problems. We conjecture that BINF is optimal for both space and time among sequential binomial-expansion algorithms whenever the problems are of at least moderate size. We have implemented algorithm BINE in PASCAL 6000 to verify the cost function $E(t, n)$ and to examine the appearance of the output. One description of the output of the current implementation is to say that it is what would be produced by any systematic binomial expansion, e.g., by hand, provided that no reordering of terms took place.

The conclusions for the family of binomial-expansion algorithms are these: Both the time complexity and the space complexity depend on design decisions for polynomial splitting, subpolynomial powering, cross product formation, and intermediate result storage. When the polynomials are sparse, even splitting reduces powering costs and storage requirements. Repeated multiplication is not the best way to power subpolynomials. Recursion and dynamic programming are successively better from the time-complexity standpoint; a particular version of dynamic programming is significantly better from the space-complexity standpoint. In cross product formation, one lowers time and space complexity by always multiplying the binomial coefficient by the smaller polynomial first.

Appendix A. Values of $B(t, n)$, $C(t, n)$, $D(t, n)$, $E(t, n)$, and $L(t, n)$ for selected values of t and n.

T	N	B	C	D	E	L
4	4	79	94	92	68	31
8	4	585	600	578	458	322
16	4	6043	5100	4928	4400	3860
17	4	7499	6214	6048	5434	4828
18	4	9200	7554	7348	6648	5967
19	4	11181	9068	8846	8060	7296
20	4	13459	10858	10564	9692	8835
21	4	16078	12826	12523	11555	10605
22	4	19051	15125	14746	13682	12628
23	4	22432	17671	17254	16494	14927
24	4	26229	20608	20074	18818	17526
25	4	30507	23773	23240	21865	20450
26	4	35268	27394	26770	25276	23725
27	4	40589	31348	30688	29075	27378
28	4	46465	35828	35026	33294	31437
29	4	52986	40632	39810	37947	35931
30	4	60139	46037	45074	43080	40890
31	4	68028	51874	50845	48720	46345
32	4	76631	58392	57160	54904	52328
4	5	125	166	162	110	52
8	5	1324	1387	1332	1036	784
16	5	22365	18908	18332	16852	15488
17	5	29182	24260	23709	21948	20332
18	5	37379	30972	30226	28184	26316
19	5	47545	38944	38146	35823	33630
20	5	59559	48742	47606	45002	42484
21	5	74179	60111	58961	56034	53109
22	5	91211	73868	72340	69090	65758
23	5	111610	89772	88118	84545	80707
24	5	135090	108731	106496	102600	98256
25	5	162833	130218	128001	123688	118730
26	5	194440	155541	152782	148052	142480
27	5	231352	184243	181346	176199	169884
28	5	273034	217685	213970	208406	201348
29	5	321220	255052	251286	245243	237307
30	5	375214	298195	293562	287040	278226
31	5	437079	346351	341457	334456	324601
32	5	505931	401460	395336	387856	376960
4	6	183	262	256	164	80
8	6	2700	2835	2732	2140	1708
16	6	72749	62282	60828	57508	54248
17	6	99525	84074	82658	78617	74596
18	6	133022	112566	110518	105756	100929
19	6	176672	148154	145956	140473	134577
20	6	230201	193598	190304	184100	177080
21	6	298336	249067	245719	238660	230209
22	6	380511	318551	313850	305936	295988
23	6	483070	402405	397349	388580	376717
24	6	605026	505629	498476	488852	474996
25	6	754709	627987	620848	610067	593750
26	6	930552	776476	767062	755124	736255
27	6	1143290	951165	941307	928212	906165
28	6	1390584	1160379	1147122	1132870	1107540
29	6	1686046	1403244	1389822	1374181	1344875

Appendix A (*contd.*)

T	N	B	C	D	E	L
30	6	2026337	1690905	1673562	1656532	1623130
31	6	2428474	2022750	2004573	1986154	1947761
32	6	2887851	2411800	2388080	2368272	2324752
4	7	254	383	374	230	116
8	7	5104	5329	5144	4104	3424
16	7	214777	187504	184084	177596	170528
17	7	307400	265714	262410	254333	245140
18	7	428381	372260	367070	357404	346086
19	7	592998	511916	506412	495157	480681
20	7	803146	697079	688258	675414	657780
21	7	1081414	933854	924985	910192	888009
22	7	1429856	1240777	1227492	1210750	1184018
23	7	1880841	1626674	1612590	1593899	1560757
24	7	2436322	2117039	2095968	2075328	2035776
25	7	3141469	2722264	2701310	2677906	2629550
26	7	3997720	3478563	3449158	3422990	3365830
27	7	5066691	4400552	4370028	4341096	4272021
28	7	6348698	5535545	5492274	5460578	5379588
29	7	7926148	6900012	6856490	6821288	6724491
30	7	9797392	8558239	8498962	8460254	8347650
31	7	12070753	10532441	10470973	10428759	10295441
32	7	14741491	12903984	12819976	12774256	12620224

Appendix B. Values of $S_B(t, n)$, $S_E(t, n)$, $\underline{S}_E(t, n)$, and $S_F(t, n)$ for selected values of t and n.

T	N	$S_B(T, N)$	$S_E(T, N)$	$\underline{S}_E(T, N)$	$S_F(T, N)$
4	4	10	18	16	15
4	5	11	28	21	19
4	6	16	40	26	24
4	7	17	54	31	28
4	8	23	70	36	33
4	9	24	88	41	37
4	10	31	108	46	42
8	4	49	68	56	46
8	5	70	138	91	66
8	6	118	250	136	100
8	7	154	418	192	128
8	8	234	658	260	175
8	9	289	988	341	211
8	10	411	1428	436	272
16	4	374	328	256	172
16	5	836	988	606	312
16	6	1880	2572	1292	620
16	7	3596	6004	2552	956
16	8	6929	12868	4748	1646
16	9	11934	25738	8411	2306
16	10	20734	48618	14296	3648
32	4	4028	1936	1472	792
32	5	15656	9688	5852	2112
32	6	55232	40696	20648	5960
32	7	171512	149224	66272	12824

Appendix B (*contd.*)

T	N	$S_B(T, N)$	$S_E(T, N)$	$S_E(T, N)$	$S_F(T, N)$
32	8	495158	490312	196280	29612
32	9	1312348	1470940	541790	55352
32	10	3289108	4085948	1404740	112740
64	4	52920	13088	9856	4400
64	5	377552	117808	71736	19904
64	6	2331328	871792	458384	87376
64	7	12626800	5521360	2623232	304432
64	8	61582652	30761872	13600880	1032984
64	9	273497784	153809368	64465628	2994240
64	10	1121535304	700687128	281610776	8548888

Acknowledgment. We wish to thank the referees for their valuable comments.

REFERENCES

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

[2] V. ALAGAR AND D. PROBST, *Binomial-expansion algorithms for computing integer powers of sparse polynomials*, International Computing Symposium 1977, E. Morlet and D. Ribbens, eds., North-Holland, Amsterdam, 1977, pp. 395–402.

[3] E. W. DIJKSTRA, *Structured programming*, Software Engineering Techniques, J. N. Buxton and B. Randell, eds., NATO Science Committee, Brussels, 1970, pp. 84–88.

[4] R. J. FATEMAN, *On the computation of powers of sparse polynomials*, Studies Appl. Math., 53 (1974), pp. 145–155.

[5] W. M. GENTLEMAN, *Optimal multiplication chains for computing a power of a symbolic polynomial*, Math. Comput., 26 (1972), pp. 935–939.

[6] E. HOROWITZ AND S. SAHNI, *The computation of powers of symbolic polynomials*, SIAM J. Comput., 4 (1975), pp. 201–208.

[7] S. C. JOHNSON, *Sparse polynomial arithmetic*, Eurosam '74, SIGSAM Bull., 8 (1974), No. 3, pp. 63–71.

[8] D. E. KNUTH, *Fundamental Algorithms, The Art of Computer Programming*, Vol. 1, 2nd ed., Addison-Wesley, Reading, MA, 1975, pp. 65–67.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0010 $01.00/0

A LINEAR TIME ALGORITHM FOR FINDING MINIMUM CUTSETS IN REDUCIBLE GRAPHS*

ADI SHAMIR†

Abstract. The analysis of many processes modeled by directed graphs requires the selection of a subset of vertices which cut all the cycles in the graph. Reducing the size of such a cutset usually leads to a simpler and more efficient analysis, but the problem of finding minimum cutsets in general directed graphs is known to be *NP*-complete. In this paper we show that in reducible graphs (and thus in almost all the "practical" flowcharts of programs), minimum cutsets can be found in linear time. We further show that the linear algorithm can check its own applicability to a given graph, thus eliminating the need of prechecking (in nonlinear time) whether it is reducible or not. An immediate application of this result is in program verification systems based on Floyd's inductive assertions method.

Key words. reducible graph, cutset, cutpoint, inductive assertions, verification

1. Motivation. A directed graph is often used as a path-generating device, which models the succession of events (in the form of edge traversals) that can take place in some process. Two common examples are the graph representations of finite state machines (with edges labeled by symbols from some alphabet) and of flowcharts of computer programs (with edges labeled by instructions).

Finite directed graphs which do not contain cycles can describe only finitely many paths, each of which contains finitely many edges, and thus the path-analysis of these graphs is usually straightforward. The analysis becomes qualitatively different in the presence of cycles, since the number and length of the paths need not be finite any longer.

However, in many cases the path-analysis of arbitrary graphs can be reduced to that of cycle-free graphs by selecting an appropriate subset of vertices (called cutpoints) such that any cycle in the graph contains at least one cutpoint. These cutpoints dissect the graph in a natural way into cycle-free components, which can be analyzed separately. All that remains to be done is to relate the overall behavior of the original graph to that of its components, and this is usually done by some kind of induction.

An important concrete example of such an analysis is Floyd's method for proving the partial correctness of computer programs (Floyd (1967)). Since execution sequences of instructions may be arbitrarily long (or infinite), one uses the selected cutpoints in the flowchart in order to "chop" them into subsequences of bounded size. If the correctness of the specifications attached to the cutpoints is preserved along any such subsequence, one can infer the overall correctness of the program by induction of the number of subsequences.

Graphs may have many sets of cutpoints, all of which are useful in principle (one example is the set of all the vertices pointed to by backward edges in a depth-first search; the potential redundancy in this choice of cutpoints is demonstrated in Fig. 1). In many cases, the number of cutpoints selected has a strong influence on the complexity of the subsequent analysis. For example, if each cutpoint gives rise to an equation (where the equated quantities may be numbers, logical formulas, or sets of strings), and the time required in order to solve n such simultaneous equations is a rapidly growing function of n, then minimizing the number n of cutpoints can be very desirable.

*Received by the editors June 12, 1978.

† Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This research was supported by the Office of Naval Research under Grant N00014-76-C-0366.

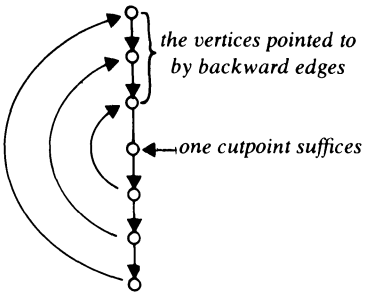


FIGURE 1

The problem of finding the smallest set of vertices which cut all the cycles in a given directed graph is *NP*-complete (Karp (1972)), and thus the optimization of the set of cutpoints is probably too expensive for big graphs. However, in this paper we show that for reducible graphs, the smallest set of cutpoints can be found in linear time.

Reducible graphs occur naturally in connection with flow-charts of computer programs. All the flowcharts which have a clear loop structure (with uniquely-defined loop entries) are reducible graphs, and as observed empirically, most programs used in practice have this property. Reducible graphs have been extensively analyzed in connection with problems of code-optimization (see Aho and Ullman (1973, vol. 2)).

As a typical application of the suggested algorithm, let us consider once more Floyd's method. An interactive implementation of this method needs a user-supplied set of inductive assertions, one for each cutpoint. Since assertions in nontrivial programs tend to be very long and very detailed, a small number of cutpoints can minimize the user's effort in finding the relations between program variables at each cutpoint, in formalizing them as inductive assertions, in "fine tuning" their pairwise power so that they may imply each other, and even in entering them into the computer. Furthermore, the number of verification conditions that the computer must prove is proportional to the number of cutpoints, and thus reducing the number of cutpoints can shorten the verification time considerably.

One possible fallacy in this argument is that some selections of cutpoints may be more natural than others, giving rise to shorter or simpler inductive assertions. However, a special feature of the suggested algorithm is that the selected cutpoints are always loop entries, and in most practical cases these places tend to be natural locations for inductive assertions.

2. Basic definitions. A *graph* is a pair (V, E) where V is the set of *vertices* and $E \subseteq V \times V$ is the set of (directed) *edges* (an edge from v to its *son* v' is denoted by $v \to v'$). A *path* from v to a *descendant* v' is a sequence of zero or more edges in E of the form $v = v_1 \to v_2 \to \cdots v_k = v'$ (which we sometimes shorten to $v \overset{*}{\to} v'$). A *cycle* is a nonempty path from a vertex to itself; it is *simple* if all the vertices along it (except the first and last) are distinct. A graph which does not contain cycles is called a *dag* (directed acyclic graph).

A *rooted graph* is a triple (V, E, r) such that (V, E) is a graph, the *root* r is in V, and for any $v \in V$ there is a path $r \overset{*}{\to} v$. A *depth first search* (DFS) of a rooted graph is a way of exploring a rooted graph, which can be implemented in linear time $O(|V|+|E|)$ on a pointer machine, using an auxiliary stack. A detailed description of DFS and its properties can be found in Tarjan (1972). A DFS defines two possible orders on the

vertices:

(i) *preorder*—the order in which vertices are pushed into the stack during the DFS.

(ii) *postorder*—the order in which vertices are popped from the stack during the DFS.

A DFS defines a partitioning of the edges into:

(i) *Backward edges* (or *fronds*): edges $v \to v'$ such that v' is already in the stack when v is pushed into the stack.

(ii) *Dag edges*: all the other edges. In the literature these edges are classified further into *tree* edges, *reverse fronds* and *cross links*; we shall not use this finer classification.

The classification of edges may depend both on the graph and on the order of search in the DFS. For a given $G = (V, E, r)$ and DFS α, we define the *dag of G defined by* α to be $G_d^\alpha = (V, E_d^\alpha, r)$ where E_d^α is the set of dag edges in E. G_d^α is always a rooted dag, and if any edge in $E - E_d^\alpha$ is added to it, a cycle is generated.

3. Cutsets in graphs.

DEFINITION. A vertex v *cuts* a path P if it is an endpoint of one of the edges in P. A set S of vertices in a graph G is a *cutset* if any cycle in G is cut by at least one vertex from S. A cutset S is *minimum* if for any other cutset S', $|S| \leqq |S'|$. The vertices in a cutset are called *cutpoints*.

Note that a minimum cutset of G need not be unique (e.g., when G is a single cycle of length $\geqq 2$), but all the minimum cutsets have the same size.

DEFINITION. The set of all cycles in a graph G is denoted by C_G. Given a set S of vertices, the set of all the cycles in G which are *not* cut by vertices in S is denoted by C_G^S.

Clearly, $C_G^\varnothing$ is the initial set of cycles C_G, and a set S is a cutset iff $C_G^S = \varnothing$.

A simple but important observation is that the cycle cutting problem is *monotonic* in the following sense:

LEMMA 1. *Let G be a graph and let S_1, S_2 be two sets of vertices such that $C_G^{S_1} \subseteq C_G^{S_2}$. Then the minimum number of vertices which should be added to S_2 to get a cutset is equal to or greater than the minimum number of vertices which should be added to S_1 to get a cutset.*

Proof. Let S_2' be a minimum set of vertices such that $S_2 \cup S_2'$ is a cutset. Any cycle in $C_G^{S_1}$ is a cycle in $C_G^{S_2}$, and thus must be cut by some vertex in S_2'. Consequently, $S_1 \cup S_2'$ is also a cutset in G, and the minimum number of vertices that should be added to S_1 to obtain a cutset cannot exceed $|S_2'|$. Q.E.D.

This lemma actually asserts that in order to solve a harder problem, more cutpoints are needed. We now use this property of the problem in order to describe an iterative process by which minimum cutsets can be monotonically "grown". The inductive hypothesis used at each stage is that the current set S of vertices is a subset of some minimum cutset. Initially $S = \varnothing$, which clearly satisfies this hypothesis. When the set S of vertices becomes a cutset, the hypothesis about being a subset of a minimum cutset makes S itself a minimum cutset. The main problem is of course how to select a new vertex which can be added to an intermediate set S without violating the inductive hypothesis and without knowing what the minimum cutsets of G are.

The following theorem shows that under certain (strong) conditions, this can be safely done:

THEOREM 1. *Let S be a subset of some minimum cutset in a graph G, and let $v_1 \to v_2 \to \cdots \to v_k \to v_1$ be an uncut cycle in C_G^S. Suppose that v_1 has the property that any cycle in C_G^S cut by some v_i $(2 \leqq i \leqq k)$ is also cut by v_1. Then there exists a minimum cutset in G which contains $S \cup \{v_1\}$.*

Proof. By assumption any cycle in C_G^S left uncut by v_1 is also left uncut by $v_2, v_3, \cdots, v_k$, and thus

$$C_G^{S\cup\{v_1\}} = C_G^S \cap C_G^{\{v_1\}} \subseteq C_G^S \cap C_G^{\{v_i\}} = C_G^{S\cup\{v_i\}}$$

for all $2 \leqq i \leqq k$. By Lemma 1, the size of the minimum cutset containing $S \cup \{v_1\}$ is smaller than or equal to the size of the minimum cutset containing $S \cup \{v_i\}$. However, the cycle $v_1 \to v_2 \to \cdots \to v_k \to v_1$ is not cut by vertices in S, and thus any cutset containing S must also contain $S \cup \{v_j\}$ for some $1 \leqq j \leqq k$. Consequently, $S \cup \{v_1\}$ is contained in some minimum cutset in G. Q.E.D.

The problem in applying Theorem 1 is that in general, there need not exist a cycle for which one of the vertices is superior to all the others in cycle-cutting power. However, as we shall show later in the paper, this is exactly the case if the graph G is reducible.

Note that a vertex v_1, which does not satisfy the condition in Theorem 1 with respect to the initial set $S = \varnothing$, may still satisfy the weakened condition at a later stage (with respect to a bigger S), since C_G^S becomes successively smaller. (For example, when a single uncut cycle remains in C_G^S, any vertex along this cycle can be taken as v_1.) Thus even if very few vertices satisfy this condition initially, it is still possible to construct iteratively a full cutset if sufficiently many vertices become available at later stages.

4. Reducible graphs. A number of equivalent definitions of reducible graphs are known (see Hecht and Ullman (1974)). One of them is:

DEFINITION. A rooted graph G is *reducible* if the dag of G defined by α, G_d^α is the same for any DFS α of G.

The simplest example of a nonreducible graph appears in Fig. 2(a). A DFS of this graph can proceed either along $v_1 \to v_2$ or along $v_1 \to v_3$, giving the two decompositions illustrated in Fig. 2(b) and Fig. 2(c) (the backward edges are denoted by double arrows and the dag edges by single arrows).

On the other hand, the graph in Fig. 3 is reducible, since it is easy to verify that any DFS must recognize $v_3 \to v_1$, $v_4 \to v_2$, $v_5 \to v_3$ and $v_6 \to v_1$ as backward edges, and all the other edges as dag edges.

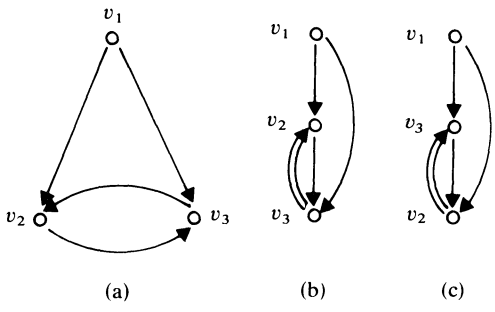


FIGURE 2

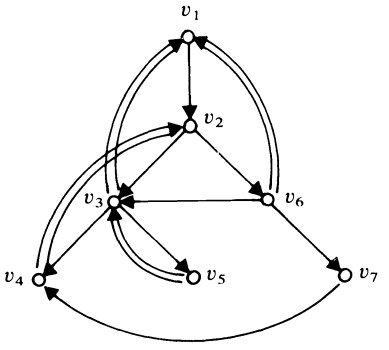


FIGURE 3

In order to check whether big graphs are reducible, some less direct methods must be used. The best known algorithm appears in Tarjan (1974), and its time complexity is slightly more than linear in the size of the graph. Some classes of graphs can be shown to contain only reducible graphs, and thus special checks are not needed for them. For example, all the graphs which can be obtained by adding to a rooted tree some edges

that always point from vertices to their tree ancestors are reducible. The dag of such a graph is the original rooted tree, and the backward edges are all the added edges.

DEFINITION. A vertex v' *dominates* another vertex v in a rooted graph $G = (V, E, r)$ if v' cuts any path $r \overset{*}{\rightarrow} v$.

One of the basic properties of reducible graphs (due to Hecht and Ullman) is:

LEMMA 2. *If $v \rightarrow v'$ is a backward edge in a reducible graph G, then v' dominates v.*

Using this lemma, it is easy to prove:

LEMMA 3. *If G is a reducible graph, then any simple cycle in G contains exactly one backward edge.*

Proof. Let C be a simple cycle in G, let P be some path from r leading into the cycle, and let v be the first vertex along P which is also along C. Proceeding from v along the cycle, let $v' \rightarrow v''$ be the first backward edge encountered (there must be at least one such edge, otherwise C would be a cycle in a dag). If $v \neq v''$, v'' does not occur along the path $r \overset{*}{\rightarrow} v \overset{*}{\rightarrow} v'$, and thus v'' does not dominate v'—a contradiction. If $v = v''$, the simple cycle closes at that point and thus cannot contain a second backward edge. Q.E.D.

Any cycle in a graph contains at least one simple cycle (just consider the first repeated occurrence of a vertex along the cycle, with respect to an arbitrary starting point). Lemma 3 provides a useful partitioning of the set of simple cycles in G in terms of their backward edges. Thus in order to check whether a given set S is a cutset, it suffices to check for any backward edge $v \rightarrow v'$ in G that all the forward dag paths from v' to v contain vertices from S.

We end this section by analyzing the possible interactions between simple cycles in a reducible graph:

THEOREM 2. *Let C_1 and C_2 be two simple cycles in a reducible graph G, which have a common vertex w. If $u \rightarrow u'$ and $v \rightarrow v'$ are the backward edges in C_1 and C_2, respectively, then either u' or v' is contained in both C_1 and C_2.* (See Fig. 4.)

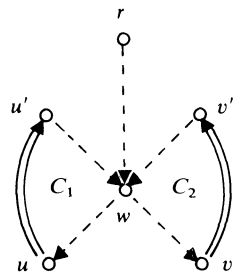


FIGURE 4

Proof. Consider an arbitrary path P in the dag G_d from the root r to w. Since this path can be extended to u and v, both u' and v' must occur along P. Without loss of generality, we can assume that u' precedes v' along P. Let P' be the path constructed in the following way:

(i) follow P from r to u';
(ii) follow C_1 from u' to w;
(iii) follow C_2 from w to v.

Since v' dominates v, it must be contained in one of these three segments. By assumption, it does not occur along segment (i) (unless $u' = v'$), and by the properties of dags it cannot occur along segment (iii) (unless $w = v'$). The result that v' occurs along both C_1 and C_2 immediately follows. Q.E.D.

5. Minimum cutsets in reducible graphs.

DEFINITION. If $v \to v'$ is a backward edge in a reducible graph G, then v' is called a *head* and v is called a *tail* in G (we also say that v' and v are *corresponding* head and tail).

DEFINITION. Let G be a reducible graph which is partially cut by a set S of vertices. Then a head v is *active* if there is some dag path from v to a corresponding tail, which is not cut by vertices from S. An active head is *maximal* if none of its proper dag descendants in G is an active head.

Example. The heads in the graph in Fig. 3 are the vertices v_1, v_2 and v_3 (note that v_1 has two corresponding tails, and that v_2 is both a head and a tail in G). When $S = \{v_3\}$, the head v_1 is active (there is an uncut path $v_1 \to v_2 \to v_6$ to one of the two corresponding tails), the head v_2 is active (the paths $v_2 \to v_3 \to v_4$ and $v_2 \to v_6 \to v_3 \to v_4$ are cut by S, but the path $v_2 \to v_6 \to v_7 \to v_4$ is still open), and the head v_3 is not active. Consequently, the only maximal active head in G is v_2.

Note than if a head v is active, then there is at least one cycle in C_G^S which contains v, but not necessarily vice versa. However, unless S is a cutset, the graph G contains at least one active head, and thus also at least one maximal active head.

The main theorem justifying the cycle cutting algorithm can now be formulated as follows:

THEOREM 3. *Let G be a reducible graph, and let S be a subset of a minimum cutset in G. If v_1 is a maximal active head in G, then $S \cup \{v_1\}$ is also a subset of a minimum cutset in G.*

Proof. Since v_1 is an active head, there is a simple cycle C_1: $v_1 \to v_2 \to \cdots \to v_k \to v_1$ in C_G^S in which $v_k \to v_1$ is the (unique) backward edge. If C_2 is any other cycle in C_G^S which is cut by some v_i $(2 \leqq i \leqq k)$, then C_1 and C_2 have a common vertex v_i. By Theorem 2, either v_1 or the head in C_2 (call it v') is contained in both cycles. If the active head v' is contained in C_1, then there is a dag path (along C_1) from v_1 to v', which contradicts the maximality of the active head v_1 (unless $v_1 = v'$). Thus v_1 must cut C_2, and we can apply Theorem 1 in order to deduce that $S \cup \{v_1\}$ is a subset of a minimum cutset in G. Q.E.D.

The basic algorithm is now a straightforward consequence of Theorem 3:

ALGORITHM A.

1. Start with $S = \varnothing$.

2. Select a maximal active head v in G with respect to the current set S. If there is none, stop; otherwise set $S \leftarrow S \cup \{v\}$ and repeat step 2.

When implementing this algorithm, it is convenient to enumerate the heads in G by a DFS, and to consider them in postorder. By the properties of DFS, all the dag descendants of a vertex v (and in particular the active heads among them) occur before v in the postorder. Any head which is found to be active at some intermediate stage in the algorithm is immediately added to S, thus ceasing to be active with respect to the new S. Furthermore, the set S can only expand, and thus a nonactive head cannot become active again at a later stage. Consequently, if heads are considered in postorder, then any head which is still active when its turn comes is a maximal active head.

Algorithm A can thus be implemented on a pointer machine in the following way:

ALGORITHM B.

Note. "top" represents the vertex which is currently at the top of the stack.

1. Set $S \leftarrow \varnothing$, push r into the (empty) stack.

2. If there is an unmarked edge top $\to v$, mark it and go to step 3, otherwise go to step 6.

3. If v has not been visited so far, push v into the stack and go to step 2.

4. If top $\rightarrow v$ is a backward edge, mark v as a head.

5. Go to step 2.

6. If v is marked as a head and is active with respect to the current set S, set $S \leftarrow S \cup \{v\}$.

7. Pop the top of the stack; if the stack is empty, halt, otherwise go to step 2.

A straightforward implementation of step 6, based directly on the definition of an active head, can be quite inefficient, but at least it shows that minimum cutsets in reducible graphs can be found in polynomial time. In the following section we optimize this "pedagogical" algorithm into a linear time algorithm.

6. The linear algorithm. The simplest way of checking whether a given head v is active is to search for uncut dag paths between v and its corresponding tails. This can be done in linear time by propagating labels from v through the dag edges, but the labeling process has to be repeated for any maximal head, thus giving a $|V| \cdot |E|$ algorithm.

In order to develop a more efficient algorithm, we structure the search in such a way that each edge is used only once, even though it may belong to dag paths between many head-tail pairs in G. Since the search must convey sufficient information in order to determine which of these paths are cut and which are still open, it seems that we need labels that are sets of heads. Unfortunately, this method leads to nonlinear algorithms even when the best known set manipulating techniques (Tarjan (1975)) are used.

As it turns out, the special structure of reducible graphs allows us to use much more economical labels. To each vertex v we attach a single number $l_S(v)$, which may change when new cutpoints are added to the current set S. Denoting by $n(v)$ (an integer between 1 and $|V|$) the position of v in the preorder (rather than postorder) sequence of vertices in G, we define:

DEFINITION. $l_S(v) = \max \{n(v') |$ there exists a backward edge $v'' \rightarrow v'$, and a dag path $v \overset{*}{\rightarrow} v''$ which is not cut by $S\}$. If no such head v' exists, then $l_S(v)$ is defined to be 0.

Example. Consider the graph in Fig. 3, in which $n(v_i) = i$ for all the vertices. When $S = \{v_5\}$, there are only two heads (v_1, v_2) whose corresponding tails are accessible from v_3 through a path which is not cut by S. Since $n(v_2) > n(v_1)$, $l_S(v_3) = 2$. When $S = \varnothing$, on the other hand, there is also an uncut dag path $v_3 \rightarrow v_5$ to the tail of the backward edge $v_5 \rightarrow v_3$, and thus $l_S(v_3) = 3$.

In reducible graphs, these labels have the following property:

THEOREM 4. *Let G be a reducible graph, let S be an arbitrary set of vertices in it, and let v be a vertex none of whose proper dag descendants is an active head. Then* $l_S(v) \leqq n(v)$.

Proof. Suppose $l_S(v) > n(v)$. By definition, $l_S(v)$ is equal to some $n(v')$, where $v'' \rightarrow v'$ is a backward edge with some uncut dag path $v \overset{*}{\rightarrow} v''$. Let $r \overset{*}{\rightarrow} v$ be the tree path from the root to v. The combined dag path $r \overset{*}{\rightarrow} v \overset{*}{\rightarrow} v''$ must contain the head v' which dominates v''. If v' occurs along the $r \overset{*}{\rightarrow} v$ tree path, its preorder number is smaller than or equal to the number of v, which contradicts the assumption that $l_S(v) = n(v') > n(v)$. Otherwise, it must occur along the uncut $v \overset{*}{\rightarrow} v''$ path; this implies that v' is an active head which is a proper dag descendant of v—again a contradiction. Q.E.D.

Let us now assume that these labels are initialized and updated (whenever S changes) by some external process, so that Algorithm B can take advantage of this extra piece of information. The following theorem shows that when successive active heads are considered and deactivated in postorder, step 6 in Algorithm B can be implemented as a simple check which requires only constant time:

THEOREM 5. *Let G be a reducible graph and let S be an arbitrary set of vertices in it. If v is a vertex such that none of its proper dag descendants is an active head, then v itself is an active head iff $l_S(v) = n(v)$.*

Proof. If $l_S(v) = n(v)$, then by definition there is some backward edge $v' \to v$ whose tail v' is accessible from v through an uncut dag path. This immediately implies that v is an active head.

On the other hand, if v is an active head, then $l_S(v) \geqq n(v)$ since $n(v)$ is one of the values maximized over in the definition of $l_S(v)$. The converse inequality $l_S(v) \leqq n(v)$ was proved (under the given conditions) in Theorem 4, and the equality follows. Q.E.D.

What remains to be done is to develop an efficient procedure for generating the labels $l_S(v)$. If we first calculate all these labels with respect to the initial set $S = \varnothing$, we may have to spend too much time updating them as S changes. Instead, we mix together the two operations of label calculation and cutpoint selection. At any intermediate stage in the process, only some of the vertices in G have labels (say, the subset L), and S is a subset of these labeled vertices. In ordered to make the process efficient, we must add new vertices to L and S according to the following rules:

(i) Vertices are added to L in postorder, i.e., a vertex is labeled only after the labels of all its dag sons are known. This order minimizes the effort involved in calculating the labels.

(ii) A vertex can be added to S only immediately after it is added to L. This order minimizes the effort involved in updating the labels, as S expands.

It is a convenient coincidence that the natural processing order in Algorithm B (which was defined for entirely different purposes) exactly satisfies these two conditions.

We now develop two procedures, one for adding a vertex to L (keeping S fixed) and one for adding a vertex to S (keeping L fixed). These "atomic" procedures preserve the correctness of the labels in the set L with respect to the cutpoints in the set S, and the overall correctness of the labeling process then follows by induction on $|L|+|S|$.

The procedure for extending L is motivated by the following theorem:

THEOREM 6. *Let G be a reducible graph and let S be an arbitrary set of vertices. Then for any vertex v in G, the following equation holds:*

$$l_S(v) = \begin{cases} 0 & \text{if } v \in S \text{ or } v \text{ does not have descendants in } G, \\ \max\left[l_s(v_1), \cdots, l_S(v_i), n(v_{i+1}), \cdots, n(v_k)\right] & \text{otherwise,} \end{cases}$$

where $v \to v_1, \cdots, v \to v_i$ are all the dag edges emanating from v, and $v \to v_{i+1}, \cdots, v \to v_k$ are all the backward edges emanating from v.

Proof. If $v \in S$ or v does not have descendants, then $l_S(v)$ must be 0 since there cannot be any uncut dag path from v to a tail of a backward edge.

If v is not in S, then clearly $l_S(v) \geqq n(v_j)$ for any vertex v_j such that $v \to v_j$ is a backward edge, since there is a trivial path from v to the tail v whose corresponding head is v_j. Similarly, if v is not in S then $l_S(v) \geqq (v_j)$ for any v_j such that $v \to v_j$ is a dag edge, since any uncut dag path from v_j to a tail can be extended to an uncut dag path from v to that tail. Consequently, $l_S(v)$ must be at least the maximum specified in the theorem.

On the other hand, $l_S(v)$ cannot exceed this maximum, since any dag path from v to a tail is either trivial, or else uses some dag edge $v \to v_j$ emanating from v. The corresponding head's number is thus bounded from above by at least one entry in the maximum. Q.E.D.

The equation in Theorem 5 shows how a new label can be computed in postorder from the known labels of its dag sons, and from the numbers of its corresponding heads. Note that while the labels of these backward sons are still unknown, they already have preorder numbers, and thus no preliminary graph traversal is needed in order to prepare these numbers. Note further that when the label of v is first computed, $v \notin S$, and thus the check $v \in S$ in the equation in Theorem 6 is superficial.

The procedure for computing these labels makes use of the fact that max is an associative operation, and thus it can be computed incrementally, taking new son values into account as they become available. The "temporary labels" thus obtained become valid labels as soon as all the sons of v are visited and we are ready to backtrack from v. The number of operations involved in adding a new vertex to L is equal to its out-degree, and thus all the $|V|$ vertices can be added to L by at most $|E|$ max operations.

We now consider the problem of updating the labels of vertices in L when a new vertex is added to S. As proved in the following theorem, at most one label can be affected if the rules of the game are observed.

THEOREM 7. *Let G be reducible graph, let L be a set of vertices for which labels have been computed in postorder, and let S be a subset of L. If v is the next vertex added to L, then adding v to S leaves all the labels in L correct with respect to the new set $S \cup \{v\}$, and changes $l_{S\cup\{v\}}(v)$ to* 0.

Proof. The fact that $l_{S\cup\{v\}}(v) = 0$ follows from the definition, since any dag path from v to a tail is cut by $S \cup \{v\}$.

If v' is any other vertex for which $l_S(v')$ is already known, then all the dag descendants of v' are in L, and thus none of them is v. Since the addition of v to S cannot affect the uncut dag paths from v' to tails, $l_{S\cup\{v\}}(v') = l_S(v')$. Q.E.D.

The final algorithm uses a single DFS of G in order to number the vertices of G in preorder, to label them in postorder, to consider successive heads in postorder, and to add new vertices to S (changing their labels to 0) when their postorder label and preorder number coincide. It uses the same skeleton as Algorithm B, and its time and space complexities are clearly linear in the size $|V|+|E|$ of G. Even though this algorithm is concise and easy to implement, its formal proof of correctness (say, using Floyd's inductive assertions method) is surprisingly subtle.

ALGORITHM C.

Note. Labels are denoted by $l(v)$, without an explicit set subscript; the algorithm keeps only one system of such labels, which correspond at any stage to the current set S.

1. Set $S \leftarrow \varnothing$; set vertex counter $c \leftarrow 1$; clear all flags; push r into the (empty) stack.
2. Set $n(top) \leftarrow c$, $c \leftarrow c+1$, $l(top) \leftarrow 0$.
3. If there is some unmarked edge $top \rightarrow v$, mark it and proceed, otherwise go to step 7.
4. If v has not been visited so far, push v into the stack and go to step 2.
5. If $top \rightarrow v$ is a backward edge, set $l(top) \leftarrow \max(l(top), n(v))$ and go to step 3.
6. Set $l(top) \leftarrow \max(l(top), l(v))$ and go to step 3.
7. If $l(top) = n(top)$, set $S \leftarrow S \cup \{top\}$ and $l(top) \leftarrow 0$.
8. Save the current top of the stack in v'; pop the stack; if the stack is empty, halt, otherwise, set $l(top) \leftarrow \max(l(top), l(v'))$; go to step 3.

Example. We demonstrate the operation of Algorithm C on the graph G in Fig. 3. Note that the DFS in which left sons are considered before right sons gives rise to preorder numbers satisfying $n(v_i) = i$ for all i.

We start by setting $S \leftarrow \varnothing$ and pushing the root v_1 into the stack. We then proceed along the dag path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4$, pushing v_2, v_3 and v_4 into the stack. The edge

$v_4 \rightarrow v_2$ is then found to be a backward edge, and thus $l(v_4) \leftarrow \max(0, n(v_2)) = 2$. Since v_4 does not have other sons, we check that $l(v_4) = 2 \neq 4 = n(v_4)$, pop it from the stack, and set $l(v_3) \leftarrow \max(0, l(v_4)) = 2$. The vertex v_5 is then pushed into the stack, and the traversal of the backward edge $v_5 \rightarrow v_3$ sets $l(v_5) \leftarrow \max(0, n(v_3)) = 3$. Popping v_5 from the stack, we update $l(v_3) \leftarrow \max(2, l(v_5)) = 3$. We then consider the backward edge $v_3 \rightarrow v_1$, which sets $l(v_3) \leftarrow \max(3, n(v_1)) = 3$. Before popping v_3 from the stack, we discover that $l(v_3) = 3 = n(v_3)$, and we thus add v_3 to S and change its label to 0.

During the rest of the process, vertex v_7 gets the label 2 calculated earlier for v_4, vertex v_6 gets the label 2 (as $\max(0, l(v_3), l(v_7), n(v_1))$), and vertex v_2 gets the label 2 (as $\max(0, l(v_3), l(v_6))$). Before backtracking from v_2 to v_1 we again discover that $l(v_2) = 2 = n(v_2)$, and thus add v_2 to S, changing its label to 0. This leads to $l(v_1) \leftarrow \max(0, l(v_2)) = 0$, and the algorithm halts after v_1 is popped from the stack.

The minimum cutset found by the algorithm is $S = \{v_3, v_2\}$. It is not uniquely minimum since $\{v_5, v_2\}$ and $\{v_3, v_6\}$ are also cutsets. All the other cutsets contain three or more vertices.

7. Algorithm C and nonreducible graphs. In some applications of cutsets (such as program verification), a small fraction of the graphs to be analyzed may be nonreducible. When applied to a nonreducible input graph, Algorithm C may mislead its user by generating (without any warning) a set S which is not a cutset, or which is an unnecessarily large cutset (note that this nonoptimality is very hard to detect, given G and S). It is thus desirable to enable Algorthim C to check its own applicability to the input graph. The obvious way of doing it is to add to Algorithm C a reducibility-checking subroutine, but this destroys its linearity.

In this section we show that by slightly augmenting Algorithm C, the reducibility-checking subroutine becomes unnecessary. We develop a condition (satisfied by all the reducible graphs as well as by some nonreducible graphs), which guarantees the correctness of the minimum cutset algorithm, and which, unlike reducibility, is easily checkable in linear time.

Algorithm C is modified into Algorithm D by adding a new step between steps 7 and 8:

7.5. If $l(top) > n(top)$, print "graph is nonreducible" and abort.

By Theorem 4, the condition in step 7.5 can never be satisfied when the input is a reducible graph, and thus erroneous messages are never printed out. What remains to be done is to show that if the input graph is nonreducible but the condition is not satisfied throughout the operation of Algorithm D, then the computed set S is still a minimum cutset.

THEOREM 8. *If Algorithm D halts successfully when applied to a graph G, then the computed set S is a cutset in G.*

Proof. If the theorem is false, there is some simple cycle $C = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \rightarrow v_1$ in G which is not cut by S (note that it may contain more than one backward edge, since G is not necessarily reducible). Without loss of generality, we may assume that v_1 has the lowest preorder number along this cycle.

Let $v_i \rightarrow v_{i+1}$ be the first backward edge after v_1 along C. The label $l(v_i)$ computed by Algorithm D is by definition at least $n(v_{i+1})$, and this value is either copied backwards or increased when the algorithm backtracks from v_i to v_1 along the dag path $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_i$ (it can decrease only when a label along this path is set to 0, but this would contradict the assumption that none of these vertices was added to S).

Consequently, the label calculated for v_1 during the operation of the algorithm satisfies

$$l(v_1) \geqq l(v_i) \geqq n(v_{i+1}) \geqq n(v_1).$$

If $l(v_1) = n(v_1)$, step 7 adds v_1 to S, thus cutting the cycle C. On the other hand, if $l(v_1) > n(v_1)$, step 7.5 causes the algorithm to abort. Since we assume that none of these situations arose, we get a contradiction to the assumption that some uncut cycle remained in G. Q.E.D.

In order to show the minimality of the cutsets which are successfully calculated by Algorithm D, we partition the vertices of G into disjoint sets according to the values of their labels. For any vertex v in G, we define

$$Q_v = \{v' | l'(v') = n(v)\}$$

where $l'(v')$ is the maximum value of the label of v' during the operation of the algorithm (i.e., for $v' \in S$ we take $l'(v') = n(v')$ rather than $l'(v) = 0$; this forces every $v' \in S$ to be in its own set $Q_{v'}$).

Example. When Algorithm D is applied to the graph G in Fig. 3, the partitioning is:

$$Q_{v_2} = \{v_2, v_4, v_6, v_7\},$$

$$Q_{v_3} = \{v_3, v_5\},$$

$$Q_{v_1} = Q_{v_4} = Q_{v_6} = Q_{v_7} = \varnothing.$$

Note that v_1 does not appear in any set, since $l'(v_1) = 0$ which is not the number of any vertex.

THEOREM 9. *For each $v \in S$, there is a simple cycle C_v in G all of whose vertices are in Q_v.*

Proof. Among all the (nonempty set of) dag paths in G which start at v and pass only through vertices in $Q_{v'}$ let P be the longest, and let v' be its endpoint. Since $v' \in Q_v$ $l'(v') = n(v)$. According to Theorem 6 (which used the reducibility of G only in order to give the notions of "dag edges" and "backward edges" a meaning which does not depend on the particular DFS order used by the algorithm), this implies that either $n(v)$ is also the label of one of the dag sons of v', or else there is a backward edge $v' \to v$ in G. The first possibility contradicts the maximality of the dag path P. The second possibility implies that P can be closed into a cycle in G all of whose vertices are in Q_v, which is the desired result. Q.E.D.

Let $S = \{v_1, \cdots, v_k\}$ be the cutset computed by Algorithm D when applied to the graph G. According to Theorem 9, with each $v_i \in S$ we can associate a cycle C_{v_i} in G, and these cycles are pairwise disjoint. Any cutset must contain at least one representative from each one of these $|S|$ cycles, and thus any cutset must contain at least $|S|$ vertices. Since S itself contains exactly $|S|$ vertices, it is a minimum cutset.

Acknowledgment. Comments by M. S. Paterson and R. E. Tarjan helped simplify an earlier version of the algorithm.

REFERENCES

A. V. AHO AND J. D. ULLMAN (1973), *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Englewood Cliffs, NJ.

R. W. FLOYD (1967), *Assigning meanings to programs*, Proc. Symp. Appl. Math., 19, pp. 19–32.

M. S. HECHT AND J. D. ULLMAN (1974), *Characterizations of reducible flow graphs*, J. Assoc. Comput. Mech., 21, pp. 367–375.

R. M. KARP (1972), *Reducibility among combinatorial problems*, Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85–104.

R. E. TARJAN (1972), *Depth first search and linear graph algorithms*, this Journal, 1, pp. 146–160.

——— (1974), *Testing Flow Graph Reducibility*, J. Comput. System Sci., 9, pp. 355–365.

——— (1975), *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mech., 22, pp. 215–225.

SIAM J. COMPUT.
Vol. 8, No. 4, November 1979

0097-5397/79/0804-0011 $01.00/0

CORRIGENDA: A COMPLETENESS CRITERION FOR SPECTRA*

T. HIKITA† AND A. NOZAKI‡

1. A result of A. V. Kuznecov has been erroneously cited in Theorem 3.1, omitting an important assumption. The correct form is: *If we assume that any* $\sim$*-incomplete spectrum is included in some* $\sim$*-maximal spectrum, then the following holds—a spectrum is* $\sim$*-complete if and only if it is not included in any* $\sim$*-maximal spectrum.*

However, this assumption follows from Proposition 3.3 (a key result of the paper) and the easily proven fact that there exists no properly increasing sequence of infinite length consisting of spectra of type first, second or third. The latter is shown by noting that, for a fixed value of k, there exist only a finite number of spectra of type second and third, and that, for a properly increasing sequence of spectra of type first, their "periods" p (see the definition of spectra of first type) must be decreasing. (Also note that Theorem 3.1 is not used in the proof of Proposition 3.3 in § 4.)

Thus, Theorem 3.1 in the original form should rather be regarded as one of the results obtained in the paper.

2. In p. 296, Reference [2], 8 (1959) should be 8 (1962).

* This Journal, 6 (1977), pp. 285–297. Received by the editors January 2, 1979.
† Department of Mathematics, Tokyo Metropolitan University, Tokyo 158, Japan.
‡ Department of Computer Science, Yamanashi University, Yamanashi 400, Japan.